Écrire des assertions avec JUnit 5 et Hamcrest

Si vous souhaitez savoir comment travailler plus intelligemment et gagner du temps lorsque vous rédigez des tests avec JUnit 5, vous devriez jeter un œil à mon cours Introduction à JUnit 5 . Il comprend 24 leçons, 47 exercices et 13 quiz .
Cet article de blog décrit comment nous pouvons écrire des assertions avec Hamcrest lorsque nous écrivons des tests avec JUnit 5. Après avoir terminé cet article de blog, nous :

- Peut obtenir les dépendances requises avec Maven et Gradle.
- Sachez comment écrire des assertions de base avec Hamcrest.
- Comprenez comment nous pouvons combiner plusieurs matchers Hamcrest.
- Peut personnaliser le message d'erreur affiché lorsqu'une assertion échoue.

Commençons.
Cet article de blog suppose que :

- Vous pouvez créer des classes de test avec JUnit 5
- Vous pouvez écrire des tests imbriqués avec JUnit 5

Obtenir les dépendances requises

Avant de pouvoir écrire des assertions avec Hamcrest, nous devons nous assurer que la `hamcrest-library`dépendance (version 2.2) est trouvée à partir du chemin de classe. Si nous utilisons Maven, nous devons ajouter la `hamcrest-library`dépendance à la `test`portée. Nous pouvons le faire en ajoutant l'extrait suivant à la `dependencies`section de notre fichier *pom.xml :*

```
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-library</artifactId>
    <version>2.2</version>
    <scope>test</scope>
</dependency>
```

Si nous utilisons Gradle, nous devons ajouter la `hamcrest-library`dépendance à la `testImplementation`configuration des dépendances. Nous pouvons le faire en ajoutant l'extrait suivant à notre fichier *build.gradle :*

```
dependencies {
    testImplementation(
            'org.hamcrest:hamcrest-library:2.2'
    )
}
```

Après nous être assurés que la `hamcrest-library`dépendance est trouvée à partir du chemin de classe, nous pouvons écrire des assertions avec Hamcrest. Voyons comment nous pouvons le faire.

Écrire des assertions avec Hamcrest

Si vous avez utilisé Hamcrest avec JUnit 4, vous vous souviendrez probablement que vous deviez utiliser la `assertThat()`méthode de la `org.junit.Assert`classe. Cependant, l'API JUnit 5 ne dispose pas de méthode prenant un matcher Hamcrest comme paramètre de méthode. Le guide de l'utilisateur JUnit 5 explique cette décision de conception comme suit :
> Cependant, la classe de JUnit Jupiter `org.junit.jupiter.api.Assertions`ne fournit pas de `assertThat()`méthode comme celle trouvée dans `org.junit.Assert`la classe de JUnit 4 qui accepte un Hamcrest `Matcher`. Au lieu de cela, les développeurs sont encouragés à utiliser la prise en charge intégrée des matchers fournie par les bibliothèques d'assertions tierces.

En d'autres termes, si nous voulons utiliser les matchers Hamcrest, nous devons utiliser la `assertThat()`méthode de la `org.hamcrest.MatcherAssert`classe. Cette méthode prend deux ou trois paramètres de méthode qui sont décrits ci-dessous :

1. Un message d'erreur facultatif qui s'affiche lorsque notre assertion échoue.
2. La valeur ou l'objet réel.
3. Un `Matcher`objet qui spécifie la valeur attendue. Nous pouvons créer de nouveaux `Matcher`objets en utilisant les `static`méthodes d'usine fournies par la `org.hamcrest.Matchers`classe.

Lecture supplémentaire :

- Le Javadoc de la `MatcherAssert`classe
- Le Javadoc de la `Matchers`classe

Ensuite, nous examinerons quelques exemples qui démontrent comment nous pouvons écrire des assertions avec Hamcrest. Commençons par découvrir comment écrire des assertions pour `boolean`des valeurs.

Affirmation de valeurs booléennes

Si nous voulons vérifier qu'une `boolean`valeur est `true`, nous devons créer notre matcher Hamcrest en appelant la `is()`méthode de la `Matchers`classe. En d'autres termes, nous devons utiliser cette assertion :

```
1
2import org.junit.jupiter.api.DisplayName;
3import org.junit.jupiter.api.Nested;
4import org.junit.jupiter.api.Test;
5
6import static org.hamcrest.Matchers.is;
7import static org.hamcrest.MatcherAssert.assertThat;
8
9@DisplayName("Write assertions for booleans")
dixclass BooleanAssertionTest {
11
12    @Nested
13    @DisplayName("When boolean is true")
14    class WhenBooleanIsTrue {
15
16        @Test
17        @DisplayName("Should be true")
18        void shouldBeTrue() {
19            assertThat(true, is(true));
20        }
21    }
}
```

Si nous voulons vérifier qu'une `boolean`valeur est `false`, nous devons créer notre matcher Hamcrest en appelant la `is()`méthode de la `Matchers`classe. En d'autres termes, nous devons utiliser cette assertion :

```
1import org.junit.jupiter.api.DisplayName;
2import org.junit.jupiter.api.Nested;
3import org.junit.jupiter.api.Test;
4
5import static org.hamcrest.Matchers.is;
6import static org.hamcrest.MatcherAssert.assertThat;
7
8@DisplayName("Write assertions for booleans")
9class BooleanAssertionTest {
```

```
dix
 11      @Nested
 12      @DisplayName("When boolean is false")
 13      class WhenBooleanIsFalse {
 14
 15          @Test
 16          @DisplayName("Should be false")
 17          void shouldBeFalse() {
 18              assertThat(false, is(false));
 19          }
 20      }
 21  }
```

Passons à autre chose et découvrons comment vérifier qu'un objet est `null`ou non `null`.

Affirmer qu'un objet est nul ou n'est pas nul

Si nous voulons vérifier qu'un objet est `null`, nous devons créer notre matcher Hamcrest en appelant la `nullValue()`méthode de la `Matchers`classe. En d'autres termes, nous devons utiliser cette assertion :

```
  1
  2 import org.junit.jupiter.api.DisplayName;
  3 import org.junit.jupiter.api.Nested;
  4 import org.junit.jupiter.api.Test;
  5
  6 import static org.hamcrest.MatcherAssert.assertThat;
  7 import static org.hamcrest.Matchers.nullValue;
  8
  9 @DisplayName("Writing assertions for objects")
dix class ObjectAssertionTest {
 11
 12      @Nested
 13      @DisplayName("When object is null")
 14      class WhenObjectIsNull {
 15
 16          @Test
 17          @DisplayName("Should be null")
 18          void shouldBeNull() {
 19              assertThat(null, nullValue());
 20          }
 21      }
 21  }
```

Si nous voulons vérifier qu'un objet ne l'est pas `null`, nous devons créer notre matcher Hamcrest en appelant la `notNullValue()`méthode de la `Matchers`classe. En d'autres termes, nous devons utiliser cette assertion :

```
  1
  2 import org.junit.jupiter.api.DisplayName;
  3 import org.junit.jupiter.api.Nested;
  4 import org.junit.jupiter.api.Test;
  5
  6 import static org.hamcrest.MatcherAssert.assertThat;
  7 import static org.hamcrest.Matchers.notNullValue;
  8
  9 @DisplayName("Writing assertions for objects")
dix class ObjectAssertionTest {
 11
 12      @Nested
```

```
    @DisplayName("When object is not null")
    class WhenObjectIsNotNotNull {

        @Test
        @DisplayName("Should not be null")
        void shouldNotBeNull() {
            assertThat(new Object(), notNullValue());
        }
    }
}
```

Nous découvrirons ensuite comment vérifier que deux objets (ou valeurs) sont égaux ou non.

Affirmer que deux objets ou valeurs sont égaux

Si nous voulons vérifier que la valeur (ou l'objet) attendue est égale à la valeur (ou l'objet) réelle, nous devons créer notre matcher Hamcrest en appelant la `equalTo()`méthode de la `Matchers`classe. Par exemple, si nous voulons comparer deux `Integer`objets, nous devons utiliser cette assertion :

```

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.equalTo;

@DisplayName("Writing assertions for objects")
class ObjectAssertionTest {

    @Nested
    @DisplayName("When two objects are equal")
    class WhenTwoObjectsAreEqual {

        @Nested
        @DisplayName("When objects are integers")
        class WhenObjectsAreIntegers {

            private final Integer ACTUAL = 9;
            private final Integer EXPECTED = 9;

            @Test
            @DisplayName("Should be equal")
            void shouldBeEqual() {
                assertThat(ACTUAL, equalTo(EXPECTED));
            }
        }
    }
}
```

Si nous voulons vérifier que la valeur (ou l'objet) attendue n'est pas égale à la valeur (ou l'objet) réelle, nous devons créer notre matcher Hamcrest en appelant la `not()`méthode de la `Matchers`classe. Par exemple, si nous voulons comparer deux `Integer`objets, nous devons utiliser cette assertion :

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;
```

```
4
5import static org.hamcrest.MatcherAssert.assertThat;
6import static org.hamcrest.Matchers.not;
7
8@DisplayName("Writing assertions for objects")
9class ObjectAssertionTest {
dix
11    @Nested
12    @DisplayName("When two objects aren't equal")
13    class WhenTwoObjectsAreNotEqual {
14
15        @Nested
16        @DisplayName("When objects are integers")
17        class WhenObjectsAreIntegers {
18
19            private final Integer ACTUAL = 9;
20            private final Integer EXPECTED = 4;
21
22            @Test
23            @DisplayName("Should not be equal")
24            void shouldNotBeEqual() {
25                assertThat(ACTUAL, not(EXPECTED));
26            }
27        }
28    }
29}
```

Passons à autre chose et découvrons comment écrire des assertions pour les références d'objets.

Affirmation de références d'objet

Si nous voulons nous assurer que deux objets font référence au même objet, nous devons créer notre matcher Hamcrest en appelant la `sameInstance()`méthode de la `Matchers`classe. En d'autres termes, nous devons utiliser cette assertion :

```
1import org.junit.jupiter.api.DisplayName;
2import org.junit.jupiter.api.Nested;
3import org.junit.jupiter.api.Test;
4
5import static org.hamcrest.MatcherAssert.assertThat;
6import static org.hamcrest.Matchers.sameInstance;
7
8@DisplayName("Writing assertions for objects")
9class ObjectAssertionTest {
dix
11    @Nested
12    @DisplayName("When two objects refer to the same object")
13    class WhenTwoObjectsReferToSameObject {
14
15        private final Object ACTUAL = new Object();
16        private final Object EXPECTED = ACTUAL;
17
18        @Test
19        @DisplayName("Should refer to the same object")
20        void shouldReferToSameObject() {
            assertThat(ACTUAL, sameInstance(EXPECTED));
```

```
        }
    }
}

```

Si nous voulons nous assurer que deux objets ne font pas référence au même objet, nous devons inverser l'attente spécifiée par la `sameInstance()`méthode en utilisant la `not()`méthode de la `Matchers`classe. En d'autres termes, nous devons utiliser cette assertion :

```

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.not;
import static org.hamcrest.Matchers.sameInstance;

@DisplayName("Writing assertions for objects")
class ObjectAssertionTest {

    @Nested
    @DisplayName("When two objects don't refer to the same object")
    class WhenTwoObjectsDoNotReferToSameObject {

        private final Object ACTUAL = new Object();
        private final Object EXPECTED = new Object();

        @Test
        @DisplayName("Should not refer to the same object")
        void shouldNotReferToSameObject() {
            assertThat(ACTUAL, not(sameInstance(EXPECTED)));
        }
    }
}
```

Si vous souhaitez accéder à du matériel à jour qui vous aide à travailler plus intelligemment et à gagner du temps lorsque vous rédigez des tests avec JUnit 5, vous devriez jeter un œil à mon cours Introduction à JUnit 5 . Il comprend 24 leçons, 47 exercices et 13 quiz.

Nous découvrirons ensuite comment vérifier que deux tableaux sont égaux.

Affirmer que deux tableaux sont égaux

Si nous voulons vérifier que deux tableaux sont égaux, nous devons créer notre matcher Hamcrest en appelant la `equalTo()`méthode de la `Matchers`classe. Par exemple, si nous voulons vérifier que deux `int`tableaux sont égaux, nous devons utiliser cette assertion :

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

import static org.hamcrest.Matchers.equalTo;
import static org.hamcrest.MatcherAssert.assertThat;

@DisplayName("Write assertions for arrays")
class ArrayAssertionTest {

    @Nested
```

```
    @DisplayName("When arrays contain integers")
    class WhenArraysContainIntegers {

        final int[] ACTUAL = new int[]{2, 5, 7};
        final int[] EXPECTED = new int[]{2, 5, 7};

        @Test
        @DisplayName("Should contain the same integers")
        void shouldContainSameIntegers() {
            assertThat(ACTUAL, equalTo(EXPECTED));
        }
    }
}
```

Deux tableaux sont considérés comme égaux si :

- Ils sont tous les deux `null`ou vides.
- Les deux tableaux contiennent les « mêmes » objets ou valeurs. Pour être plus précis, JUnit 5 itère les deux tableaux un élément à la fois et garantit que les éléments trouvés à partir de l'index donné sont égaux.

Passons à autre chose et découvrons comment nous pouvons écrire des assertions pour les listes.

Affirmation de listes

Si nous voulons écrire une assertion qui vérifie que la taille d'une liste est correcte, nous devons créer notre matcher Hamcrest en appelant la `hasSize()`méthode de la `Matchers`classe. Par exemple, si l'on veut vérifier que la taille d'une liste est de 2, il faut utiliser cette assertion :

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

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

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.hasSize;

@DisplayName("Writing assertions for lists")
class ListAssertionTest {

    @Nested
    @DisplayName("When we write assertions for elements")
    class WhenWeWriteAssertionsForElements {

        private Object first;
        private Object second;

        private List<Object> list;

        @BeforeEach
        void createAndInitializeList() {
            first = new Object();
```

```
            second = new Object();

            list = Arrays.asList(first, second);
        }

        @Test
        @DisplayName("Should contain two elements")
        void shouldContainTwoElements() {
            assertThat(list, hasSize(2));
        }
    }
}
```

Si nous voulons vérifier que la liste contient uniquement les éléments attendus dans l'ordre donné, nous devons créer notre matcher Hamcrest en appelant la `contains()`méthode de la `Matchers`classe. Par exemple, si nous voulons vérifier que notre liste contient les bons éléments dans l'ordre donné, nous devons utiliser cette assertion :

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

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

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.contains;
dix
@DisplayName("Writing assertions for lists")
class ListAssertionTest {

    @Nested
    @DisplayName("When we write assertions for elements")
    class WhenWeWriteAssertionsForElements {

        private Object first;
        private Object second;

        private List<Object> list;

        @BeforeEach
        void createAndInitializeList() {
            first = new Object();
            second = new Object();

            list = Arrays.asList(first, second);
        }

        @Test
        @DisplayName("Should contain the correct elements in the given order")
        void shouldContainCorrectElementsInGivenOrder() {
            assertThat(list, contains(first, second));
        }
    }
}
```

```
37
38
```

Si nous voulons vérifier que la liste contient uniquement les éléments attendus dans n'importe quel ordre, nous devons créer notre matcher Hamcrest en appelant la `containsInAnyOrder()`méthode de la `Matchers`classe. Par exemple, si nous voulons vérifier que notre liste contient les bons éléments dans n'importe quel ordre, nous devons utiliser cette assertion :

```
1
2
3  import org.junit.jupiter.api.BeforeEach;
4  import org.junit.jupiter.api.DisplayName;
5  import org.junit.jupiter.api.Nested;
6  import org.junit.jupiter.api.Test;
7
8  import java.util.Arrays;
9  import java.util.List;
dix
11 import static org.hamcrest.MatcherAssert.assertThat;
12 import static org.hamcrest.Matchers.containsInAnyOrder;
13
14 @DisplayName("Writing assertions for lists")
15 class ListAssertionTest {
16
17     @Nested
18     @DisplayName("When we write assertions for elements")
19     class WhenWeWriteAssertionsForElements {
20
21         private Object first;
22         private Object second;
23
24         private List<Object> list;
25
26         @BeforeEach
27         void createAndInitializeList() {
28             first = new Object();
29             second = new Object();
30
31             list = Arrays.asList(first, second);
32         }
33
34         @Test
35         @DisplayName("Should contain the correct elements in any order")
36         void shouldContainCorrectElementsInAnyOrder() {
37             assertThat(list, containsInAnyOrder(second, first));
38         }
39     }
40 }
```

Si nous voulons nous assurer qu'une liste contient l'élément donné, nous devons créer notre matcher Hamcrest en appelant la `hasItem()`méthode de la `Matchers`classe. Par exemple, si nous voulons vérifier que notre liste contient les `Object`éléments stockés dans le champ appelé `first`, nous devons utiliser cette assertion :

```
1 import org.junit.jupiter.api.BeforeEach;
2 import org.junit.jupiter.api.DisplayName;
3 import org.junit.jupiter.api.Nested;
4 import org.junit.jupiter.api.Test;
```

```
5
6import java.util.Arrays;
7import java.util.List;
8
9import static org.hamcrest.MatcherAssert.assertThat;
dix import static org.hamcrest.Matchers.hasItem;
11
12@DisplayName("Writing assertions for lists")
13class ListAssertionTest {
14
15    @Nested
16    @DisplayName("When we write assertions for elements")
17    class WhenWeWriteAssertionsForElements {
18
19        private Object first;
20        private Object second;
21
22        private List<Object> list;
23
24        @BeforeEach
25        void createAndInitializeList() {
26            first = new Object();
27            second = new Object();
28
29            list = Arrays.asList(first, second);
30        }
31
32        @Test
33        @DisplayName("Should contain a correct element")
34        void shouldContainCorrectElement() {
35            assertThat(list, hasItem(first));
36        }
37    }
38}
```

Si nous voulons nous assurer qu'une liste ne contient pas d'élément, nous devons inverser l'attente spécifiée par la `hasItem()`méthode en utilisant la `not()`méthode de la `Matchers`classe. En d'autres termes, nous devons utiliser cette assertion :

```
1import org.junit.jupiter.api.BeforeEach;
2import org.junit.jupiter.api.DisplayName;
3import org.junit.jupiter.api.Nested;
4import org.junit.jupiter.api.Test;
5
6import java.util.Arrays;
7import java.util.List;
8
9import static org.hamcrest.MatcherAssert.assertThat;
dix import static org.hamcrest.Matchers.hasItem;
11import static org.hamcrest.Matchers.not;
12
13@DisplayName("Writing assertions for lists")
14class ListAssertionTest {
15
16    @Nested
17    @DisplayName("When we write assertions for elements")
```

```
    class WhenWeWriteAssertionsForElements {

        private Object first;
        private Object second;

        private List<Object> list;

        @BeforeEach
        void createAndInitializeList() {
            first = new Object();
            second = new Object();

            list = Arrays.asList(first, second);
        }

        @Test
        @DisplayName("Should not contain an incorrect element")
        void shouldNotContainIncorrectElement() {
            assertThat(list, not(hasItem(new Object())));
        }
    }
}
```

Si nous voulons vérifier que deux listes sont profondément égales, nous devons créer notre matcher Hamcrest en invoquant la `equalTo()`méthode de la `Matchers`classe. Par exemple, si l'on veut vérifier que deux `Integer`listes sont profondément égales, il faut utiliser cette assertion :

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

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

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.equalTo;
dix
@DisplayName("Writing assertions for lists")
class ListAssertionTest {

    @Nested
    @DisplayName("When we compare two lists")
    class WhenWeCompareTwoLists {

        private final List<Integer> ACTUAL = Arrays.asList(1, 2, 3);
        private final List<Integer> EXPECTED = Arrays.asList(1, 2, 3);

        @Test
        @DisplayName("Should contain the same elements")
        void shouldContainSameElements() {
            assertThat(ACTUAL, equalTo(EXPECTED));
        }
    }
}
```

Deux listes sont considérées comme égales si :

- Ils sont tous les deux `null`ou vides.
- Les deux listes contiennent les « mêmes » objets ou valeurs. Pour être plus précis, JUnit 5 parcourt les deux listes un élément à la fois et garantit que les éléments trouvés à partir de l'index donné sont égaux.

Nous découvrirons ensuite comment écrire des assertions pour les cartes.

Affirmation de cartes

Si nous voulons vérifier qu'une carte contient la clé donnée, nous devons créer notre matcher Hamcrest en appelant la `hasKey()`méthode de la `Matchers`classe. En d'autres termes, nous devons utiliser cette assertion :

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

import java.util.HashMap;
import java.util.Map;

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.hasKey;

@DisplayName("Writing assertions for maps")
class MapAssertionTest {

    private static final String KEY = "key";
    private static final String VALUE = "value";

    private Map<String, String> map;

    @BeforeEach
    void createAndInitializeMap() {
        map = new HashMap<>();
        map.put(KEY, VALUE);
    }

    @Nested
    @DisplayName("When we check if the map contains the given key")
    class WhenWeCheckIfMapContainsGivenKey {

        @Test
        @DisplayName("Should contain the correct key")
        void shouldContainCorrectKey() {
            assertThat(map, hasKey(KEY));
        }
    }
}
```

Si nous voulons vérifier qu'une carte ne contient pas la clé donnée, nous devons inverser l'attente spécifiée par la `hasKey()`méthode en utilisant la `not()`méthode de la `Matchers`classe. En d'autres termes, nous devons utiliser cette assertion :

```
import org.junit.jupiter.api.BeforeEach;
```

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

import java.util.HashMap;
import java.util.Map;

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.hasKey;
import static org.hamcrest.Matchers.not;

@DisplayName("Writing assertions for maps")
class MapAssertionTest {

    private static final String INCORRECT_KEY = "incorrectKey";
    private static final String KEY = "key";
    private static final String VALUE = "value";

    private Map<String, String> map;

    @BeforeEach
    void createAndInitializeMap() {
        map = new HashMap<>();
        map.put(KEY, VALUE);
    }

    @Nested
    @DisplayName("When we check if the map contains the given key")
    class WhenWeCheckIfMapContainsGivenKey {

        @Test
        @DisplayName("Should not contain the incorrect key")
        void shouldNotContainIncorrectKey() {
            assertThat(map, not(hasKey(INCORRECT_KEY)));
        }
    }
}
```

Si nous voulons nous assurer qu'une carte contient la bonne valeur, nous devons créer notre matcher Hamcrest en appelant la `hasEntry()`méthode de la `Matchers`classe. En d'autres termes, nous devons utiliser cette assertion :

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

import java.util.HashMap;
import java.util.Map;

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.hasEntry;

@DisplayName("Writing assertions for maps")
class MapAssertionTest {

```

```
    private static final String KEY = "key";
    private static final String VALUE = "value";

    private Map<String, String> map;

    @BeforeEach
    void createAndInitializeMap() {
        map = new HashMap<>();
        map.put(KEY, VALUE);
    }

    @Nested
    @DisplayName("When we check if the map contains the correct value")
    class WhenWeCheckIfMapContainsCorrectValue {

        @Test
        @DisplayName("Should contain the correct value")
        void shouldContainCorrectValue() {
            assertThat(map, hasEntry(KEY, VALUE));
        }
    }
}
```

Passons à autre chose et découvrons comment nous pouvons combiner plusieurs matchers Hamcrest.

Combiner les Matchers Hamcrest

Nous pouvons maintenant écrire des assertions de base avec Hamcrest. Cependant, nous devons parfois combiner plusieurs matchers Hamcrest. En fait, nous l'avons déjà fait lorsque nous avons inversé l'attente d'un matcher Hamcrest en invoquant la `not()`méthode de la `Matchers`classe.

Ensuite, nous examinerons deux exemples qui démontrent comment nous pouvons combiner les matchers Hamcrest lorsque nous écrivons des assertions pour un `Person`objet. Le code source de la `Person`classe se présente comme suit :

```
public class Person {

    private String firstName;
    private String lastName;

    public Person() {}

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public void setLastName(String lastName) {
```

```
        this.lastName = lastName;
    }
}

```

Comme nous pouvons le voir, si nous voulons vérifier qu'une personne porte le nom correct, nous devons nous assurer que l' `Person`objet affirmé porte le nom et le prénom corrects. Lorsque nous écrivons cette assertion, nous devons créer le matcher Hamcrest qui est passé à la `assertThat()`méthode en utilisant ces Hamcrest Matchers :

- La `allOf()`méthode de la `Matchers`classe renvoie un matcher Hamcrest qui s'attend à ce que l'objet affirmé corresponde à **tous** les matchers Hamcrest spécifiés.
- La `hasProperty()`méthode de la `Matchers`classe renvoie un matcher qui nous permet d'écrire des assertions pour les propriétés de l'objet affirmé.
- La `equalTo()`méthode de la `Matchers`classe renvoie un matcher qui permet de vérifier que la valeur réelle de la propriété est égale à la valeur attendue.

Après avoir rédigé notre assertion, elle se présente comme suit :

```


import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.allOf;
import static org.hamcrest.Matchers.equalTo;
import static org.hamcrest.Matchers.hasProperty;
dix
@DisplayName("Combine multiple assertions")
class CombineAssertionsTest {

    private static final String FIRST_NAME = "Jane";
    private static final String LAST_NAME = "Doe";

    private Person person;

    @BeforeEach
    void createPerson() {
        person = new Person();
        person.setFirstName(FIRST_NAME);
        person.setLastName(LAST_NAME);
    }

    @Test
    @DisplayName("Should have the correct name")
    void shouldHaveCorrectName() {
        assertThat(person, allOf(
                hasProperty("firstName", equalTo(FIRST_NAME)),
                hasProperty("lastName", equalTo(LAST_NAME))
        ));
    }
}
```

D'un autre côté, si nous voulons vérifier qu'une personne a le prénom ou le nom correct, nous devons créer le matcher Hamcrest qui est transmis à la `assertThat()`méthode en utilisant ces Hamcrest Matchers :

- La `anyOf()`méthode de la `Matchers`classe renvoie un matcher Hamcrest qui s'attend à ce que l'objet affirmé corresponde à **n'importe quel** matcher Hamcrest spécifié.
- La `hasProperty()`méthode de la `Matchers`classe renvoie un matcher qui nous permet d'écrire des assertions pour les propriétés de l'objet affirmé.
- La `equalTo()`méthode de la `Matchers`classe renvoie un matcher qui permet de vérifier que la valeur réelle de la propriété est égale à la valeur attendue.

Après avoir rédigé notre assertion, elle se présente comme suit :

```
1
2
3  import org.junit.jupiter.api.BeforeEach;
4  import org.junit.jupiter.api.DisplayName;
5  import org.junit.jupiter.api.Test;
6
7  import static org.hamcrest.MatcherAssert.assertThat;
8  import static org.hamcrest.Matchers.anyOf;
9  import static org.hamcrest.Matchers.equalTo;
dix import static org.hamcrest.Matchers.hasProperty;
11
12 @DisplayName("Combine multiple assertions")
13 class CombineAssertionsTest {
14
15     private static final String FIRST_NAME = "Jane";
16     private static final String LAST_NAME = "Doe";
17
18     private Person person;
19
20     @BeforeEach
21     void createPerson() {
22         person = new Person();
23         person.setFirstName(FIRST_NAME);
24         person.setLastName(LAST_NAME);
25     }
26
27     @Test
28     @DisplayName("Should have correct first name or last name")
29     void shouldHaveCorrectFirstNameOrLastName() {
30         assertThat(person, anyOf(
31                 hasProperty("firstName", equalTo(FIRST_NAME)),
32                 hasProperty("lastName", equalTo(LAST_NAME))
33         ));
34     }
35 }
```

De nombreuses méthodes d'usine fournies par la `Matchers`classe peuvent prendre un matcher Hamcrest (ou des matchers) comme paramètre de méthode. C'est pourquoi je vous recommande de jeter un œil à sa documentation lorsqu'il semble que vous ne puissiez pas écrire l'assertion dont vous avez besoin. Il y a de fortes chances que vous puissiez l'écrire en combinant plusieurs matchers Hamcrest.

Nous découvrirons ensuite comment fournir un message d'erreur personnalisé qui s'affiche lorsque notre assertion échoue.

Fournir un message d'erreur personnalisé

Comme nous nous en souvenons, si nous voulons spécifier un message d'erreur personnalisé qui s'affiche lorsque notre assertion échoue, nous devons transmettre ce

message comme premier paramètre de méthode de la `assertThat()`méthode. Nous pouvons créer ce message d'erreur en utilisant l'une de ces deux options :

- Si le message d'erreur n'a pas de paramètres, nous devons utiliser un `String`littéral.
- Si le message d'erreur a des paramètres, nous devons utiliser la `static format()`méthode de la `String`classe.

Par exemple, si nous voulons créer un message d'erreur qui s'affiche lorsque la liste affirmée ne contient pas l'élément donné, nous devons créer une assertion qui ressemble à ceci :

```
1
2
3  import org.junit.jupiter.api.BeforeEach;
4  import org.junit.jupiter.api.DisplayName;
5  import org.junit.jupiter.api.Nested;
6  import org.junit.jupiter.api.Test;
7
8  import java.util.Arrays;
9  import java.util.List;
dix
11 import static org.hamcrest.MatcherAssert.assertThat;
12 import static org.hamcrest.Matchers.hasItem;
13
14 @DisplayName("Writing assertions for lists")
15 class ListAssertionTest {
16
17     @Nested
18     @DisplayName("When we write assertions for elements")
19     class WhenWeWriteAssertionsForElements {
20
21         private Object first;
22         private Object second;
23
24         private List<Object> list;
25
26         @BeforeEach
27         void createAndInitializeList() {
28             first = new Object();
29             second = new Object();
30
31             list = Arrays.asList(first, second);
32         }
33
34         @Test
35         @DisplayName("Should contain a correct element")
36         void shouldContainCorrectElementWithCustomErrorMessage() {
37             assertThat(String.format(
38                             "The list doesn't contain the expected object: %s",
39                             first
40                     ),
41                     list,
42                     hasItem(first)
43             );
44         }
45     }
46 }
```

Il est bon de comprendre que le message d'erreur personnalisé ne remplace pas le message d'erreur par défaut affiché si une assertion échoue. Il s'agit simplement d'un préfixe ajouté au message d'erreur par défaut du matcher Hamcrest utilisé. Au début, cela semble un peu bizarre, mais c'est en fait très utile une fois qu'on s'y est habitué.

Nous pouvons désormais écrire des assertions de base avec Hamcrest, combiner plusieurs correspondants Hamcrest et fournir un message d'erreur personnalisé qui s'affiche lorsqu'une assertion échoue.
Résumons ce que nous avons appris de cet article de blog.

Résumé

Cet article de blog nous a appris quatre choses :

- Avant de pouvoir écrire des assertions avec Hamcrest, nous devons nous assurer que la `hamcrest-library`dépendance est trouvée à partir du chemin de classe.
- Si nous voulons écrire des assertions avec Hamcrest, nous devons utiliser la `assertThat()`méthode de la `org.hamcrest.MatcherAssert`classe.
- Si nous souhaitons fournir un message d'erreur personnalisé qui s'affiche lorsqu'une assertion échoue, nous devons transmettre ce message d'erreur comme premier paramètre de méthode de la `assertThat()`méthode.
- Certaines méthodes de la `Matchers`classe peuvent prendre un ou plusieurs matchers Hamcrest comme paramètre de méthode. Nous pouvons combiner plusieurs matchers Hamcrest en utilisant ces méthodes.