R3.04 - TP n°2 :
Prise en main des doublures de test et de Mockito

Cet énoncé est disponible en ligne sur :
https://github.com/iblasquez/tuto_mockito

La page officielle du framework Mockito est accessible à l'adresse suivante https://site.mockito.org/
Une javadoc et documentation permettant de faciliter l'utilisation du framework Mockito est disponible ici : https://www.javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html

Exercice 1 : Premiers pas avec les doublures dans Mockito

Dans ce premier exercice, vous allez écrire vos premières doublures Java générées à l'aide du framework **Mockito**. Les exemples à implémenter sont ceux qui illustrent le cours **Doublures de tests**.

Commencez par **créer un projet Maven** que vous appellerez **hellodoublure**.
Pour pouvoir générer des doublures, **ajoutez une dépendance vers le framework Mockito** à votre **pom.xml**. Pour cela, ajoutez la dépendance **mockito-core** dans le bloc `<dependencies>…</dependencies>` de votre **pom.xml** « habituel » (reprendre celui du TP précédent 😉).

```
<dependency>
	<groupId>org.mockito</groupId>
	<artifactId>mockito-core</artifactId>
	<version>4.8.0</version>
	<scope>test</scope>
</dependency>
```

Pour connaître la version en cours, rendez-vous sur le site de Mockito https://site.mockito.org/
et **relevez le numéro de version indiquée en bleu à côté de** maven central
N'oubliez pas de remettre à jour votre **pom.xml**!!!

1.a : Le Stub (bouchon)

Le ***stub*** est une doublure qui fait l'objet d'une implémentation minimale.
Son implémentation tient compte du contexte exact dans lequel la doublure sera utilisée.
Le stub a ainsi pour but de fournir des réponses prédéfinies lorsque l'objet sous test (**SUT : S**ystem **U**nder **T**est) a besoin de ses services.

Le contexte (le code source disponible) :
Imaginez que vous êtes chargés de développer la partie concernant la ***messagerie*** d'un projet et qu'un autre développeur est chargé de développer les parties ***authentification*** et ***utilisateur*** de ce même projet. Le travail de votre collègue va, entre autres, consister à fournir une implémentation de la classe **User** qui exposera un service de type `getLogin` (que vous allez devoir utiliser pour implémenter la messagerie). Vous vous mettez d'accord sur la signature de cette méthode qui pour l'instant sera écrite de la manière la plus simple possible c-a-d sous la forme :
`public Object getLogin().`

Dans **src/main/java** (dans le paquetage **fr.unilim.iut**), créez et implémentez la classe **User** de la manière suivante :

```
public class User {

	public Object getLogin() {
		// TODO Auto-generated method stub
		// Rien d'implémenter pour l'instant :
		// votre collègue est en train de l'implémenter ;-)
		return null;
	}
}
```

Le problème : Au cours de votre développement, vous devez écrire un test qui fera appel à la méthode `getLogin` (qui n'existe pas encore réellement).

La solution Pour simuler cela, créez et implémentez dans **src/test/java** (dans le paquetage **fr.unilim.iut**), la classe **TestDoublures** et la méthode **test_unPremierStub** de la manière suivante :

```
import org.junit.Test;
public class TestDoublures {

	@Test
	void test_UnPremierStub() {

		User user = mock(User.class);
		when(user.getLogin()).thenReturn("alice");
		}
	}
```

Les deux lignes de code de ce test correspondent :

→ à la **création de la doublure** qui se fait par un appel à la méthode statique **`mock`**.
Rappelons qu'il est également possible de créer une doublure via l'annotation **`@Mock`**

→ au **bouchonnage de la doublure** à l'aide de la méthode statique **`when`**.
Le bouchonnage permet de spécifier le comportement que devra adopter la doublure lorsque la méthode bouchonnée. Dans notre test :
QUAND ***`getLogin`*** *est appelé sur la doublure* `user` ***ALORS*** *la doublure* ***RENVOIE*** `"alice".`

Attention !!! Notez bien que les méthodes **`mock`** et **`when`** sont des méthodes statiques comme l'indique la javadoc du framework Mockito et donc que ce code ne pourra compiler qu'après l'ajout des **`import static`** suivants :

```
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
```

Compilez ce code !!!
Vous pouvez exécuter ce code (c-a-d lancer le test), mais comme pour l'instant il n'y a aucune étape d'assertion, cela ne sert pas à grand-chose. En effet, si vous lancez ce code de test, le test sera forcément VERT !!!

Utilisation du bouchonnage :

A partir de maintenant, on peut donc utiliser ***`getLogin`*** n'importe où dans la suite du test.
A chaque fois que ***`getLogin`*** sera appelé, il renverra `"alice"`.
Pour illustrer ce fait, ajoutez après le bouchonnage la ligne de code suivante :

```
System.out.println(user.getLogin());
```

Compilez et exécutez !
Vous devriez voir s'afficher `alice` dans la console ….

Pour vérifier ce « bon » comportement, nous allons maintenant faire appel à **`getLogin`** dans l'étape d'assertion du test. En fin de test, ajoutez l'étape d'***Assertion*** suivante :

```
assertEquals(user.getLogin(), "alice");
```

en n'oubliant pas l'**`import static`** pour *`assertEquals`* !

Compilez et exécutez !
Le test doit passer au VERT !!!

Suite à cette assertion, ajouter l'assertion suivante :

```
assertEquals(user.getLogin(), "bob");
```

Compilez et exécutez !
La barre de tests est désormais AU ROUGE en raison de cette seconde assertion.
Le test est en échec car la valeur attendue est `bob` et c'est toujours `alice` qui est renvoyée par `getLogin`!

Commentez ou supprimez cette dernière assertion qui fait échouer le test.
Relancez le test pour continuer sur une barre VERTE !!!

Remarque : Dans cet exemple, nous avons utilisé la méthode bouchonnée dans l'étape d'assertion car nous voulions illustrer le « bon » comportement du bouchon… Dès lors qu'une méthode est bouchonnée, elle va pouvoir être utilisée n'importe où dans le test (et notamment dans l'étape ***Arrange*** du pattern **AAA**) dès que l'objet sous test aura besoin du service bouchonné pour implémenter correctement le comportement à tester ….

1.b : Le Mock (objet Factice)

Du stub au mock, il n'y a qu'un pas puisqu'un mock permet en plus de bouchonner un comportement, de mettre en place une **vérification comportementale** c-a-d de vérifier quelle fonction a été appelée, avec quel(s) argument(s), combien de fois ….

Dans la classe `TestDoublures`, ajoutez la méthode de test suivante :

```
@Test
void test_UnPremierMock() {

	User user = mock(User.class);
	when(user.getLogin()).thenReturn("alice");

	System.out.println(user.getLogin());
}
```

Pour l'instant, tout comme un stub, la doublure est créée (*mock*), bouchonnée (*when*) et utilisée dans le test (utilisation simulée par un affichage).

→ En fin de test, ajoutez l'instruction suivante :

```
verify(user).getLogin();
```

Cette instruction permet de vérifier que la méthode **`getLogin`** a bien été appelée (1 fois) sur l'objet **`user`** : elle permet donc de mettre en place une vérification comportementale.

Compilez et exécutez le test !
Le test doit passer au VERT !!!

→ Commentez l'instruction `System.out.println`

Compilez et exécutez le test !
Le test passe AU ROUGE !!!
Et le message d'erreur indique : ***Wanted but not invoked : user.getLogin()***

Décommentez l'instruction `System.out.println`
Compilez et exécutez le test !
Le test doit repasser au VERT !!!

→ ***Transformez*** l'instruction :

```
verify(user).getLogin();
```

En

```
verify(user, times(2)).getLogin();
```

en n'oubliant pas l' `import static org.mockito.Mockito.times;`

Compilez et exécutez le test !
Le test doit être ROUGE !!!

Ajoutez une nouvelle instruction `System.out.println(user.getLogin());`

Compilez et exécutez le test !
Le test doit être VERT maintenant !!!

Ces exemples montrent que le mock est un objet simulé dont le comportement est décrit spécifiquement pour un test unitaire dans un test unitaire.
Avec un mock, l'assertion du test unitaire est réalisée, comme nous venons de le montrer, via un appel à **`verify`**…

Pour tester les différentes options de vérification comportementale possible qu'offre la méthode `verifiy`, vous pouvez terminer cet exercice en implémentant et testant l'exemple suivant (extrait de la rubrique **Verifying exact number of invocations / at `least x / never`** de la javadoc : https://www.javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html#exact_verification)

```
@Test
void test_OptionsVerification() {

        LinkedList<String> mockedList = mock(LinkedList.class);

         mockedList.add("once");

         mockedList.add("twice");
         mockedList.add("twice");

         mockedList.add("three times");
         mockedList.add("three times");
         mockedList.add("three times");

        //following two verifications work exactly the same - times(1) is used by default
         verify(mockedList).add("once");
         verify(mockedList, times(1)).add("once");

         //exact number of invocations verification
         verify(mockedList, times(2)).add("twice");
         verify(mockedList, times(3)).add("three times");

         //verification using never(). never() is an alias to times(0)
         verify(mockedList, never()).add("never happened");

         //verification using atLeast()/atMost()
         verify(mockedList, atLeastOnce()).add("three times");
         verify(mockedList, atLeast(2)).add("three times");
         verify(mockedList, atMost(5)).add("three times");
}
```

Exercice 2 : Ecrivez votre première doublure …

Créez un nouveau projet Maven que vous appellerez **firstdoublure** et **ajoutez une dépendance vers Mockito** au **pom.xml** de ce projet.

Imaginez que vous êtes en train de travailler sur une application dans laquelle on souhaite maintenant créer des utilisateurs dont le mot de passe est correctement haché.

Vous n'êtes pas un spécialiste de la cryptographie et vous décidez donc de déléguer la partie sur le hachage à un de vos collègues qui sera chargé de développer un composant capable de vous fournir un mot de passe correctement haché à partir d'un texte passé en paramètre.

Vous vous mettez d'accord sur l'interface de ce composant :

```
public interface HashProvider {
    String hash(String text);
}
```

Pour vous, ce composant n'est donc pas disponible pour le moment, seule son l'interface est connue. Vous devez donc **commencer par ajouter l'interface `HashProvider`** dans le code source de votre projet (**src/main/java**)

Votre travail consiste ensuite à développer un service qui va permettre de créer un utilisateur.
Vous allez donc déclarer, à votre tour, une interface **`UserService`** qui propose le service que vous êtes en train de développer.
Dans **src/main/java**, ajoutez donc maintenant l'interface **`UserService`** suivante :

```
public interface UserService {
     User createUser(String firstname, String password);
}
```

Pour que ce code puisse compiler, il faut disposer de la classe **`User`** que vous vous empressez d'implémenter le plus simplement possible.
Dans **src/main/java**, vous ajoutez donc la classe **`User`** suivante :

```
public class User {

     private final String firstName;
     private final String hashedPassword;

     public User(String firstName, String hashedPassword) {
          this.firstName = firstName;
          this.hashedPassword = hashedPassword;
     }

     public String firstName() {
          return firstName;
     }

     public String hashedPassword() {
          return hashedPassword;
     }
}
```

Vérifiez que votre code compile bien ! … avant de continuer 😉

Il ne vous reste plus qu'à implémenter le service **createUser** qui vous a été demandé.
… Et vous l'implémentez le plus simplement possible dans la classe **UserServiceImpl** en faisant appel au **HashProvider** que votre collègue est en train d'écrire de son côté …

```
public class UserServiceImpl implements UserService {

    private final HashProvider hashProvider;

    public UserServiceImpl(HashProvider hashProvider) {
        this.hashProvider = hashProvider;
    }

      @Override
      public User createUser(String firstName, String password) {
            String hashedPassword = hashProvider.hash(password);
            return new User(firstName, hashedPassword);
      }
}
```

Ajoutez la classe **UserServiceImpl** dans votre **src/main/java !**

Vérifiez que votre code compile ! Pour mieux comprendre la conception que vous venez d'implémenter, **à l'aide de votre IDE, procédez à un petit reverse sur le code métier (src/main/java) pour visualiser le diagramme de classes correspondant**.

Il ne vous reste plus qu'à tester votre implémentation!

Dans **src/test/java**, créez une classe **UserServiceImplTest** et commencez à l'implémenter comme suit avec les **import** qui permettent de faire compiler le code en ne laissant qu'une erreur sur l'instruction à compléter (celle avec **new** UserServiceImpl(???))

```
public class UserServiceImplTest {

   @Test
   void should_create_user_with_hashed_password() {

           UserService userService = new UserServiceImpl(???);

           User user = userService.createUser("Bob", "secret");

           assertEquals(user.firstName(), "Bob");
           assertEquals(user.hashedPassword(), "???");
    }
}
```

Cette classe doit vous permettre au travers de la méthode **should_create_user_with_hashed_password** de tester que votre service fonctionne correctement c-a-d que l'utilisateur est bien créé avec un mot de passe correctement haché (différent de celui saisi).

Mais que mettre dans les **???** puisque le composant qui est censé fournir le mot de passe crypté n'est pas encore disponible puisque votre collègue est justement en train de le développer …
La solution à ce problème consiste bien sûr à utiliser une doublure de **HashProvider** pour simuler le comportement attendu …

A vous de jouer !!!
Complétez ce test avec une doublure de type `mock` et faites le passer au VERT !!!

Exercice 3 : Kata dice

Ce tutoriel s'inspire du Kata Designing a Game of Dices *de* *Sébastien Mosser* *et Simon Urli*

Ce kata consiste à implémenter **un jeu de dé** à deux joueurs. Chaque joueur aura droit à deux lancers. Le vainqueur est le joueur qui aura obtenu la plus grande valeur lors d'un de ses lancers.
En cas d'égalité les joueurs rejouent (en relançant deux fois le dé).
Après 5 égalités, le jeu se termine sans vainqueur.

Ce kata va être découpé en plusieurs **fonctionnalités** de manière à ce qu'il puisse être implémenté petit pas par petit pas.

Nous n'adopterons pas une approche TDD (Test Driven Development) durant ce kata (pas de *test first*).
Les **tests unitaires** seront écrits **après le code**.
Des **mocks** pourront être utilisés.
Une fois les tests écrits, une phase de **refactoring** pourra être envisagée pour améliorer la qualité du code….

Spécifications fonctionnelles :

Les différentes fonctionnalités à implémenter pour mener à bien ce kata sont donc :

1. **Etre capable de lancer un dé**
   → *critère d'acceptation* : le dé a 6 faces et retourne un nombre aléatoire entre 1 et 6.

2. **Ajouter un joueur**
   Associer un lancer de dé à un joueur.
   → *critère d'acceptation* : un joueur a un nom et expose la valeur obtenue par son propre dé

3. **Garder la valeur maximale entre de deux lancers**
   Le joueur lance deux fois le dé et la valeur maximale est conservée.
   → *critère d'acceptation* : le dé est lancé juste deux fois et seule la valeur maximale est conservée.

4. **Jouer aux dés :** Notre jeu de dés se joue à deux.
   Le vainqueur est celui qui obtient la valeur maximale après avoir lancé ses deux dés.
   En cas d'égalité les joueurs rejouent (en relançant deux fois le dé).
   Après 5 égalités, le jeu se termine sans vainqueur.
   → *critère d'acceptation* : le jeu donne le vainqueur en tenant compte des règles précédentes

Mise en place du projet :

Créer un projet maven que vous appellerez `gameofdices`.
Configurer le **pom.xml** afin d'utiliser vos versions habituelles de **Java** et de **JUnit** et d'ajouter le framework **Mockito** comme dépendance à ce projet.

Fonctionnalité n°1 : Lancer un dé

1.a Code de production (src/main/java)

Créer une classe **Dice** dans le paquetage **gameofdices** de **src/main/java**.
Cette classe, responsable de lancer le dé, propose ce service au travers de la méthode **roll**.
Implémentez, dans un premier temps, la classe **Dice** de la manière suivante :

```
public class Dice {

   private final static int FACES = 6;
   private Random rand;

   public Dice(Random rand) { this.rand = rand; }

   public int roll() {
      int result = rand.nextInt(FACES) + 1;
      if (result < 1 || result > FACES) {
         throw new RuntimeException("Dice returns an incompatible value");
          }
      return result;
      }
   }
```

Quelques remarques sur ce code :
→ dans le cadre de ce kata, nous considérons qu'**un dé à 6 faces uniquement** sera utilisé.
→ un objet de type **Random** sera fourni à la création du dé pour prendre en compte le côté *reproductible* du lancer.
→ Une exception de type `RuntimeException` est utilisée. Une `RuntimeException` soulève normalement une erreur de programmation. Ici, ce devrait donc plutôt être une classe d'exception personnalisée montrant l'intention métier qui devrait être levée. Comme ce kata se focalise sur les tests et les mocks (plutôt que sur les exceptions), nous nous contenterons pour le moment d'une `RuntimeException` , même si cette dernière fait apparaître une certaine dette technique 😉

1.b Code de test (src/test/java)

Pour vérifier le critère d'acceptation « le dé a 6 faces et retourne un nombre aléatoire entre 1 et 6 » , plusieurs scénarios de tests doivent être envisagés pour valider ce « bon » comportement :

→ pour tester que la méthode `roll` fonctionne *normalement* et que chaque lancer de dé renvoie bien une valeur valide entre 1 et 6, la méthode de test suivante peut être implémentée :

```
@Test
void rollReturnsAValue() {
      theDice = new Dice(new Random());
      for (int i = 0; i < 100; i++) {
            int result = theDice.roll();
            assertTrue(result >= 1);
            assertTrue(result <= 6);
      }
}
```

→ pour tester que la méthode `roll` est **capable de renvoyer une exception** si la valeur tirée au sort se trouve être en dehors de [1-6], deux méthodes scénarios doivent alors être envisagés :

- un premier scénario pour tester le *bon* comportement en cas de **valeur trop grande**

```
@Test
void identifyBadValuesGreaterThanNumberOfFaces() {
      ???
      theDice = new Dice(???);
      theDice.roll();
      assertThrows(RuntimeException.class, () -> theDice.roll());
}
```

- un second scénario pour tester le *bon* comportement en cas de **valeur trop petite**

```
@Test
void identifyBadValuesLesserThanOne() {
    ???
    theDice = new Dice(???);
    theDice.roll();
    assertThrows(RuntimeException.class, () -> theDice.roll());
}
```

Que mettre dans les ??? : Dans ces ceux derniers tests, on ne peut pas se contenter pour l'initialisation du dé (**???**) d'un simple tirage aléatoire (objet de type **Random**). Il est indispensable de s'assurer que le tirage pourra répondre aux besoins du test c-a-d forcer l'objet aléatoire à retourner une valeur cohérente avec ce que l'on souhaite tester (une valeur >= 7 pour le premier test est une valeur <=-1 pour le second) pour bien pouvoir « tester » le déclenchement des exceptions.

Travail à faire :
→ Ecrire une classe de test **`DiceTest`** qui reprend l'implémentation de ces trois tests.

```
public class DiceTest {

      Dice theDice;

      @Test
      void rollReturnsAValue() {
            //…à compléter avec l'implémentation donnée ci-dessus
      }

      @Test
      void identifyBadValuesGreaterThanNumberOfFaces() {
            //…à compléter avec l'implémentation donnée ci-dessus
      }

      @Test
      void identifyBadValuesLesserThanOne() {
            //…à compléter avec l'implémentation donnée ci-dessus
      }
}
```

→ Dans les méthodes `identifyBadValuesGreaterThanNumberOfFaces` et `identifyBadValuesLesserThanOne`, complétez les **???** par des doublures de **Random** qui bouchonne la méthode **nextInt** (pour n'importe quelle valeur entière) avec une valeur propice au déclenchement des exceptions.
→ **Compilez et exécutez les tests :** ils doivent passer AU VERT !!!

Fonctionnalité n°2 : Ajouter un joueur

2.a Code de production (src/main/java)

Cette fonctionnalité vise à faire apparaître un joueur (classe **`Player`**) dans notre implémentation.
La classe **`Player`** devra, au minimum, être composée :

- d'un constructeur avec le **nom** du joueur et le **dé** associé au joueur.
- d'un attribut permettant de connaître la dernière valeur mémorisée suite à un lancer de dé (**`lastValue`**), valeur initialisée par défaut à **-1** (tant que le dé n'a pas encore été lancé).
- d'une méthode pour de jouer (**`play`**) permettant de lancer le dé et mémoriser la valeur obtenue suite à ce lancer.

Implémentez dans **src/main/java** la classe **`Player`** de la manière suivante :

```
public class Player {

	private String name;
	private Dice dice;
	private int lastValue = -1;

	public Player(String name, Dice dice) {
		this.name = name;
		this.dice = dice;
	}

	public void play() {
		this.lastValue = dice.roll();
	}

	public int getLastValue() {
		return lastValue;
	}

}
```

2.b Code de test (src/test/java)

Pour vérifier le critère d'acceptation « *un joueur a un nom et expose la valeur obtenue par son propre dé*», deux scénarios doivent être envisagés pour valider l'exposition de la « bonne » valeur de dé :

→ lors de l'initialisation du joueur, la dernière valeur doit être la valeur par défaut (-1)
→ après avoir joué, la dernière valeur doit être différente de la valeur par défaut (c-a-d différente de -1).

Tester le code de classe `Player` revient donc à écrire la classe `PlayerTest` suivante avec les **`import`** qui vont bien 😉

```
public class PlayerTest {

	Player player;

	@Test
	void lastValueNotInitialized() {
		player = new Player("John Doe", new Dice(new Random()));
		assertEquals(player.getLastValue(), -1);
	}

	@Test
	void lastValueInitialized() {
		player = new Player("John Doe", new Dice(new Random()));
		player.play();
		assertNotEquals(player.getLastValue(), -1);
	}

}
```

Implémentez cette classe dans **src/test/java**.
Compilez et exécutez les tests : ils doivent passer AU VERT !!!

2.c Une petite phase de refactoring ?

Mais au fait, comment a été choisie la valeur **-1** ? … de manière arbitraire !
-1 est donc un code smell de type Magic Number qui contribue à la dette technique…
Un petit refactoring s'impose !...

En effet, savez-vous que Java8 a introduit la classe Optional qui permet d'encapsuler un objet dont la valeur peut être **définie ou null** ?

Pour pouvoir utiliser cette classe assurez-vous que votre **pom.xml** soit bien configuré avec (au minimum) la version **1.8** de Java :

```
<properties>
	<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
	<maven.compiler.source>1.8</maven.compiler.source>
	<maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

Une fois cette configuration effectuée, il est possible d'utiliser la classe **Optional** pour supprimer le Magic Number ***-1*** et permettre à l'attribut **lastValue** de contenir une valeur ou pas…
Pour ce faire, il suffit de de transformer le type primitif ***int*** en ***Optional<Integer>***.

Modifiez la classe **Player** afin de faire apparaître ***Optional*** :

```
public class Player {

	private String name;
	private Dice dice;
	private Optional<Integer> lastValue = Optional.empty();

	public Player(String name, Dice dice) {
		this.name = name;
		this.dice = dice;
	}

	public void play() {
		this.lastValue = Optional.of(dice.roll());
	}

	public Optional<Integer> getLastValue() {
		return lastValue;
	}

}
```

Compilez ce code.

Exécutez les tests : la barre de tests est désormais AU ROUGE !!!
Normal, la valeur **–1** n'a plus de raison d'être dans les assertions des tests.
Le but des assertions est de savoir si le dé a déjà été lancé ou pas, c-a-d si une valeur est connue ou pas c-a-d **si une valeur est bien présente ou non dans lastValue**. D'après la javadoc d'Optional, c'est la méthode isPresent qui : «*Return true if there is a value present, otherwise false*. »

Modifiez la classe **PlayerTest** en faisant apparaître ***isPresent***.

```
public class PlayerTest {
	Player player;

	@Test
	void lastValueNotInitialized() {
		player = new Player("John Doe", new Dice(new Random()));
		assertFalse(player.getLastValue().isPresent());
	}

	@Test
	void lastValueInitialized() {
		player = new Player("John Doe", new Dice(new Random()));
		player.play();
		assertTrue(player.getLastValue().isPresent());
	}
}
```

Exécutez les tests : ils doivent maintenant passer AU VERT !!!
Attention à bien modifier également les `assertEquals` par des `assertTrue` et `assertFalse` 😉

Remarque : Pour en savoir un peu plus sur **Optional**, vous pouvez, à l'occasion, consulter :
→ la javadoc sur **Optionnal (**https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html**)**
→ Cet article : **apprendre à utiliser la classe Optional<T> pour éviter d'utiliser explicitement null** (https://www.developpez.com/actu/134958/Java-apprendre-a-utiliser-la-classe-Optional-lt-T-gt-pour-eviter-d-utiliser-explicitement-null-par-Gugelhupf) ou celui-là **Optional en Java 8** (http://www.touilleur-express.fr/2014/11/07/optional-en-java-8/) par exemple.

Fonctionnalité n°3 : Garder la valeur maximale entre deux lancers

3.a Code de production (src/main/java)

Via cette fonctionnalité, le joueur doit pouvoir lancer deux fois le dé et la valeur maximale doit pouvoir être conservée.
Pour implémenter ce comportement, il suffit juste de modifier la méthode **play** en faisant apparaître 2 appels à **roll** et une recherche de valeur maximale.

Modifiez la méthode **play** de la classe `Player` de la manière suivante :

```
public void play() {
   int firstRoll = dice.roll();
   int secondRoll = dice.roll();
   this.lastValue = Optional.of(Math.max(firstRoll, secondRoll));
   }
```

Compilez !

3.b Code de test (src/test/java)

Pour vérifier le critère d'acceptation « *le dé est lancé juste deux fois et seule la valeur maximale est conservée*», deux scénarios doivent être envisagés :
→ le premier devra vérifier que le dé est lancé juste deux fois (**throwDiceOnlyTwice**)
→ le second devra vérifier que c'est bien la valeur maximale qui est conservée (**keepTheMaximum**)

3.b.1 Pour vérifier que le dé est lancé juste deux fois(throwDiceOnlyTwice),

il faut mettre en place une **vérification comportementale** afin de comptabiliser le nombre d'appels à la méthode à la méthode **roll** lors de l'exécution de ce scénario de test. Ce type de vérification peut facilement être mis en place sur une doublure à l'aide de la méthode **verify** du framework **Mockito**.
La méthode **roll** appartenant à la classe **Dice**, le scénario devra être écrit à partir d'un **mock** de **Dice**.

Implémentez avec un `mock` la méthode de test **throwDiceOnlyTwice** dans la classe **PlayerTest**.

```
@Test
void throwDiceOnlyTwice() {
      Dice dice = mock(Dice.class);
      player = new Player("John Doe", dice);
      player.play();
      verify(dice, times(2)).roll();
}
```

Exécutez les tests : ils doivent passer AU VERT !!!

3.b.2 Pour vérifier que c'est bien la valeur maximale qui est conservée (keepTheMaximum), il faut mettre en place un **bouchonnage** afin de pouvoir contrôler les valeurs renvoyées par la méthode **roll** et s'assurer que seule la valeur maximale est conservée.
Un tel test pourrait s'écrire de la manière suivante.

```
@Test
void keepTheMaximum() {
	Dice dice = mock(Dice.class);
	player = new Player("John Doe", dice);

	when(dice.roll()).thenReturn(2).thenReturn(5);
	player.play();
	assertEquals(player.getLastValue().get(), new Integer(5));

	when(dice.roll()).thenReturn(6).thenReturn(1);
	player.play();
	assertEquals(player.getLastValue().get(), new Integer(6));
}
```

Remarque : Deux assertions sont écrites dans ce test :
- la première pour vérifier un scénario où la valeur issue du second lancer (**5**) est la valeur maximale.
- la seconde pour vérifier un scénario où la valeur issue du premier lancer (**6**) est la valeur maximale.

Implémentez ce test dans la classe **PlayerTest**

Exécutez les tests : ils doivent passer AU VERT !!!

Fonctionnalité n°4 : Jouer aux dés

4.a Code de production (src/main/java)

Il ne reste plus qu'à implémenter le **jeu** de dés (**Game**) à proprement parler.
Rappelons que :
- ce jeu se joue à *deux joueurs*
- ce jeu *désigne le vainqueur* d'une partie en tenant compte des règles suivantes :
  Le vainqueur est celui qui obtient la valeur maximale après avoir lancé ses deux dés.
  En cas d'égalité les joueurs rejouent (en relançant deux fois le dé).
  Après 5 égalités, le jeu se termine sans vainqueur.

Un tel comportement peut être implémentée dans une classe **Game** de la manière suivante :

```
import java.util.Optional;

public class Game {

	private Player left;
	private Player right;

	public Game(Player left, Player right) {
		this.left = left;
		this.right = right;
	}
```

```
	public Optional<Player> play() {
		int counter = 0;
		while (counter < 5) {
			left.play();
			int leftValue = left.getLastValue().get();
			right.play();
			int rightValue = right.getLastValue().get();

			if (leftValue > rightValue) {
				return Optional.of(left);
			} else if (rightValue > rightValue) {
				return Optional.of(right);
			}

			counter++;
		}
		return Optional.empty();
	}
}
```

Remarque : Comme le jeu peut se terminer avec ou sans vainqueur, un `Optional<Player>` est utilisé comme type de retour de la méthode `play`.

Implémentez la classe **`Game`** dans **src/main/java**

Compilez ! .. et faites en sorte que ce code compile… Il ne reste plus qu'à le tester…

4.b Code de test (src/test/java)

Pour vérifier le critère d'acceptation « *le jeu donne le vainqueur en tenant compte des règles*», deux scénarios doivent être envisagés :

→ un premier scénario où **un vainqueur peut être désigné**

→ un second scénario où, après 5 égalités, il n'y aura **aucun vainqueur.**

4.b.1 Pour vérifier que le jeu est bien capable de désigner un vainqueur, il faut mettre en place un **bouchonnage** sur la classe **Player** afin de pouvoir contrôler les valeurs renvoyées par la méthode **getLastValue** et s'assurer que le « bon » joueur est renvoyé.

Ecrire dans ***src/test/main*** une classe **GameTest** qui décrit dans le comportement d'un tel test :

```
public class GameTest {
	Game game;
	@Test
	public void andTheWinnerIs() {

		Player player1 = mock(Player.class);
		when(player1.getLastValue()).thenReturn(Optional.of(new Integer(5)));

		Player player2 = mock(Player.class);
		when(player2.getLastValue()).thenReturn(Optional.of(new Integer(2)));

		game = new Game(player1, player2);
		assertEquals(player1, game.play().get());
	}
}
```

Après avoir implémenté et fait compiler **GameTest** avec les `import` qui vont bien,
exécutez les tests : ils doivent passer AU VERT !!!

4.b.2 Pour vérifier que le jeu est bien capable de ne désigner aucun vainqueur après avoir pris en compte 5 égalités il faut :

- commencer par mettre en place un **bouchonnage** sur la classe **Dice** afin que le dé renvoie toujours une seule et même valeur utilisée par le premier ET le second joueur, ce qui revient à ce que les joueurs utilisent le même dé bouchonné 😉

```
Dice single = mock(Dice.class);
when(single.roll()).thenReturn(1);

Player player1 =  new Player("John", single);
Player player2 =  new Player("Jane", single);
```

- s'assurer que chaque joueur va jouer 5 fois c-a-d que la méthode **play** va être appelée 5 fois pour chaque joueur (pour simuler les 5 égalités), ce qui revient à mettre en place une vérification comportementale via la méthode **verify** sur une doublure de **Player**.
Dans ce test, nous ne souhaitons pas bouchonner (forcer) le comportement d'un joueur, mais ***seulement l'espionner*** pour juste savoir si la méthode **play** est bien appelée 5 fois.
Pour (d)écrire ce scénario, un objet de type **Spy** semble donc plus adapté à cette situation qu'un objet de type **Mock**, ce qui revient à implémenter la méthode de test de la manière suivante :

```
@Test
void noWinnerAfter5Attempts() {
	Dice single = mock(Dice.class);
	when(single.roll()).thenReturn(1);

	Player player1 =  spy(new Player("John", single));
	Player player2 =  spy(new Player("Jane", single));

	game = new Game(player1,player2);
	assertFalse(game.play().isPresent());
	verify(player1, times(5)).play();
	verify(player2, times(5)).play();
}
```

Implémentez le test `noWinnerAfter5Attempts` dans la classe **GameTest**.
Exécutez les tests : ils doivent passer AU VERT !!!