

Dernière bataille : Retour vers le projet wardcardgame !



Le client qui souhaitait vous confier le développement de différents jeux de cartes revient vers vous. Il a décidé de vous faire développer **le jeu de la bataille** (à partir d'un jeu de 52 cartes).

Un développement logiciel commence toujours par une **phase d'analyse** pendant laquelle vous devez comprendre le besoin du client et définir le **périmètre fonctionnel** de votre application (**ce que doit faire votre application**).

S'en suit une phase de **conception**, puis d'**implémentation** dans le langage de votre choix, sans oublier les **tests** pour s'assurer du bon comportement de l'application et sa non-régression lors de l'ajout de nouvelles fonctionnalités, de correction de bugs ou de remaniement de code...

1. Se familiariser avec le contexte métier et sa problématique :

Avant de se lancer dans la phase d'analyse, il faut **comprendre et avec le contexte métier**. Vous commencerez donc par vous familiariser avec **les règles du jeu de la bataille**.

(Re)Découverte des règles du jeu de cartes la bataille :

La bataille, les règles du jeu

- On distribue les **52 cartes** aux joueurs (la bataille se joue généralement à deux) qui les rassemblent **face cachée** en paquet devant eux.
- Chacun tire la carte du dessus de son paquet et **la pose face visible** sur la table.
- Celui qui a la carte la plus forte ramasse les autres cartes.
- L'as est la plus forte carte, puis roi, dame, valet, 10, etc.
- Lorsque deux joueurs posent en même temps **deux cartes de même valeur il y a « bataille »**. Lorsqu'il y a "bataille" les joueurs tirent la carte suivante et la posent, face cachée, sur la carte précédente. Puis, ils tirent une deuxième carte qu'ils posent cette fois-ci face visible et c'est cette dernière qui départagera les joueurs. Celui qui a la valeur la plus forte, l'emporte. S'il y a encore une égalité, on refait une « bataille »
- Le gagnant est celui qui remporte toutes [les cartes du paquet](#).

Ces règles sont extraites de : <https://www.momes.net/jeux/jeux-interieur/regle-des-jeux-de-cartes/la-bataille-regles-du-jeu-842140> Une vidéo est également disponible sur ce site ...

La bataille selon [https://fr.wikipedia.org/wiki/Bataille_\(jeu\)](https://fr.wikipedia.org/wiki/Bataille_(jeu))

La **bataille** est un jeu de cartes qui se joue habituellement à deux (bien que le nombre de joueurs puisse être supérieur) qui est d'une grande simplicité pour les débutants, puisqu'on peut y jouer sous la conduite exclusive du hasard (bien que la manière dont sont rangés les plis peut influencer considérablement l'issue du jeu, pour les joueurs avancés).

Une bonne pratique consiste à s'immerger dans le contexte métier.

❑ Phase d'Analyse :

En analysant les règles précédentes et en tenant compte de *l'expérience utilisateur* que vous venez de de vivre via la partie réelle que nous venons de jouer, vous avez listez pas à pas toutes les « *étapes* » qui seront nécessaires pour mener à bien une partie du jeu de la bataille dans notre logiciel.

Ces « *étapes* » ne sont autres que les **cas d'utilisation (use case)**, on parle aussi de **fonctionnalité (feature)** en développement agile, que vous devrez développer pas à pas pour livrer une application opérationnelle :

1. Préparer un jeu de cartes de 52 cartes
2. Mélanger le jeu de 52 cartes
3. Distribuer le jeu de cartes de manière équitable et aléatoire entre les joueurs
4. Tirer une carte du dessus du paquet
5. Jouer une bataille
6. Rechercher le joueur qui a la carte la plus forte
7. Ramasser les cartes
8. Déclarer le joueur vainqueur de la partie

... Dans une deuxième itération, vous avez affinés ces cas d'utilisant en détaillant plus de cas d'utilisation et/ou en faisant apparaître des termes métiers plus précis comme *levée, plis, avoir la main*, ... (termes métiers définis bien sûr dans un glossaire !)

1. Préparer un jeu de cartes de 52 cartes
2. Mélanger le jeu de 52 cartes
3. Distribuer le jeu de cartes de manière équitable et aléatoire (**entre les joueurs**)
4. **Jouer une main**
 - a. **piocher une carte sur le dessus du tas**
 - b. **éventuellement organiser une bataille**
 - c. **Déterminer le joueur qui peut lever la main** (Rechercher le joueur qui a la carte la plus forte)
5. Ramasser les cartes
 - a. **S'emparer de la main** (effectuer une levée)
 - b. **Plier la levée** (Ranger la levée au-dessous du tas)
6. **Terminer la partie** en déclarant le vainqueur de la partie

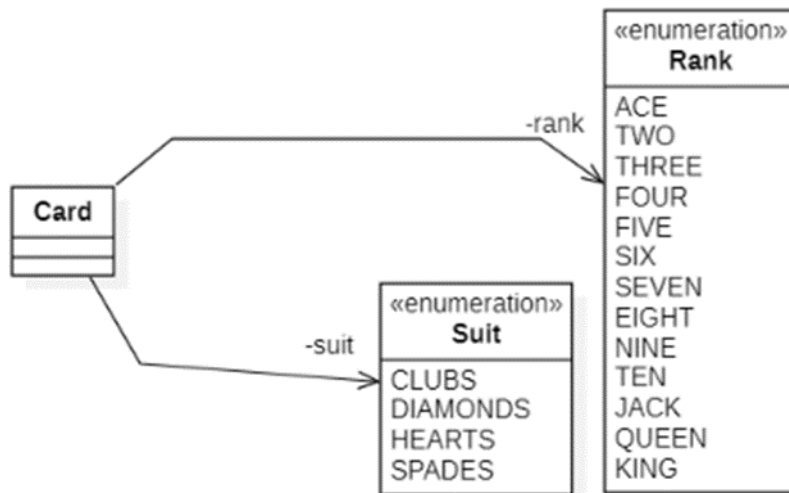
1^{ère} itération : Pas de jeu sans cartes !

❑ (première) Phase de conception

Après la phase d'analyse vient la phase de conception 😊

Le TP Pas de jeu sans carte ! vous a permis de commencer à vous focaliser sur l'**élément de base** de cette application : **la carte à jouer**, puisque **sans carte, pas de jeu à développer !** 😊

Et en vous appuyant sur la phase d'analyse menée précédemment autour de la notion de *carte*, la **phase de conception** vous a permis de proposer le diagramme de classes suivant.



❑ Mise en place du socle technique

Avant de passer à la phase d'implémentation, il est important de mettre en place le projet qui accueillera cette implémentation. Le **TP Un projet / Un workspace Mise en place du socle technique** vous a permis de mettre en place le *socle technique* (appelé aussi *stack technique*)

Ce TP a donc permis de mettre en place le socle technique en suivant les spécifications suivantes :

- le projet est nommé **wardcardgame** et se trouve dans le workspace **cardgame** (si votre IDE est Eclipse).
- le projet **wardcardgame** est un projet **maven**
- le projet **wardcardgame** est **versionné avec git**.
- le *remote* de votre projet **wardcardgame** se trouve sur votre compte **gitunilim**

❑ (première) Phase d'implémentation

Une fois le socle technique installé, il est possible de passer à la phase de conception.

Les deux TPs *Pas de jeu sans carte !* et *Un projet / Un workspace Mise en place du socle technique* ont contribué à cette phase de conception en vous permettant d'implémenter vous-même les classes métiers **Card**, **Suit** et **Rank**, ainsi qu'une classe **CardMain** où sont écrits des jeux de tests manuels permettant de vérifier et valider les implémentations précédentes. Pour ceux qui n'auraient pas eu le temps de tout implémenter, un gist a même été mis à votre disposition (<https://unil.im/cardgame>)

Au fait, avez-vous pensé à générer **hashCode** et **equals** avec l'aide de votre IDE sur **Card** ?

Si tel n'est pas le cas, empressez-vous de rattraper votre oubli avant de continuer 😊

Source → Generate hashCode() and equals()...

❑ Point sur l'actuelle architecture du projet wardcardgame (dans l'IDE)

Il est temps de rouvrir votre projet wardcardgame dans votre IDE favori !

Sous Eclipse, n'oubliez pas de changer de workspace et de vous rendre dans **cardgame** 😊

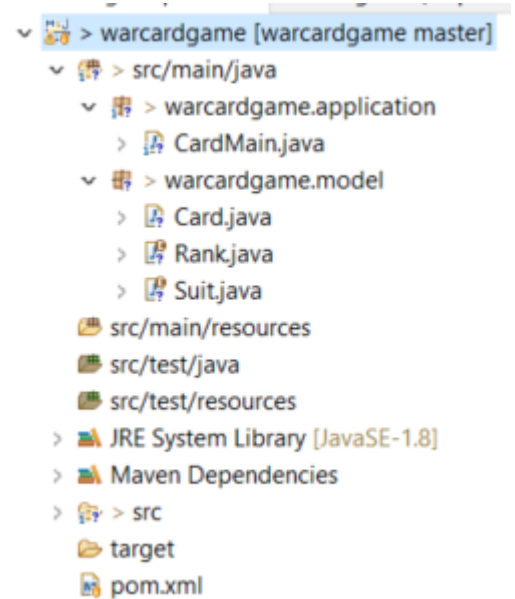
Actuellement, la structure de votre projet wardcardgame est donc la suivante :

→ **Le code de production dans src/main/java** avec :

- dans un **paquetage wardgame.application** :
la classe CardMain qui permet de **lancer l'application** (au travers d'un main, le point d'entrée du programme) et contient juste actuellement l'implémentation de jeux d'essais manuels.
- dans un **paquetage wardgame.model** :
le code **métier** au travers de **Card** et **Rank** et **Suit**

→ **Le code de test dans src/test/java**

⇒ Pour l'instant, aucun test automatisé n'a été écrit... Le code de la classe CardMain ayant permis de vérifier et valider le bon comportement du code implémenté.



❑ Point sur l'historique des commits

Et pour terminer de vous replonger dans ce projet, utiliser par exemple votre IDE ou GitKralen, pour visualiser l'historique git de vos commits !

*Cette première itération de développement sur le projet wardcardgame s'est focalisé sur le concept au cœur du projet la **carte** !*

Sur quel concept allons-nous focaliser la deuxième itération ?

*Pour jouer, il faut plusieurs cartes : intéressons-nous maintenant au **paquet de cartes** !!!*

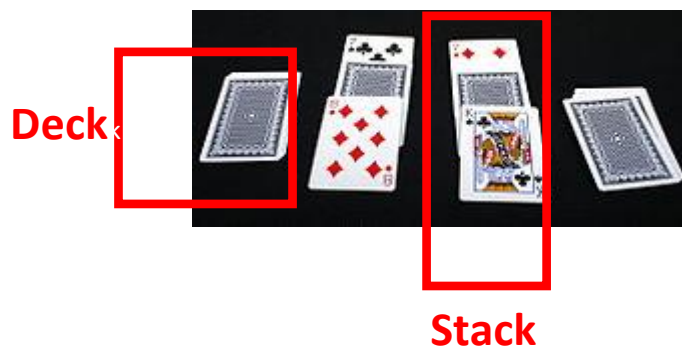
2^{ème} itération : Pas de jeu sans *paquet* de cartes !

❑ Rapide analyse et délimitation du périmètre fonctionnel de la 2^{ème} itération

A la (re)lecture des règles du jeu et suite à une *quick design session* avec vos collègues (une rapide séance collaborative de discussion et de réflexion), vous avez identifié trois paquets de cartes différents qui sont utilisés pour jouer à la bataille et qui ont des rôles (responsabilités) différentes.

Pour garder une trace de votre réflexion commune et aligner tous les développeurs sur la compréhension commune de ces concepts, vous avez complété le glossaire de la manière suivante :

- ➔ **Le jeu de cartes initial (standard deck)** est le paquet de cartes initial constitués de 52 cartes. Il est créé une fois en début de partie et n'est utilisé qu'une seule fois en début de partie pour distribuer les cartes de manière équitable et aléatoire entre les joueurs
- ➔ **Le paquet de cartes d'un joueur (deck)** est le paquet de cartes utilisé par un joueur tout au long de la partie. Il change d'état à chaque tour, puisqu'à chaque tour au moins une carte est retirée (plus s'il y a une bataille). Si le joueur gagne le tour, des cartes sont également ajoutées à ce paquet. Le nombre de cartes présentes dans ce paquet permet de déterminer si le joueur est éliminé (si le paquet est vide) ou si le joueur a gagné (si le paquet contient les 52 cartes du jeu de cartes initial).
- ➔ **La main (ou tas) d'un joueur (stack) :** est un nouveau paquet propre à chaque tour. Au début de chaque tour, la main doit être vide, puis une ou plusieurs cartes sont posées (suivant si une bataille doit avoir lieu). Les cartes sont ensuite retirées de la main pour être redistribuées au joueur gagnant le tour.



Délimitation du périmètre fonctionnel de la 2^{ème} itération : Cette rapide phase d'analyse autour de la notion de *paquets de carte*, vous a amené à identifier 3 types de paquets de cartes (**standard deck**, **deck** et **stack**). Pour cette itération, l'équipe choisit de livrer uniquement la première étape de l'application, à savoir :

Etape 1 : Mise en place (et distribution) du matériel :

- ⇒ 1. Préparer un jeu de cartes de 52 cartes
- ⇒ 2. Mélanger le jeu de 52 cartes
- ⇒ 3. Distribuer le jeu de cartes de manière équitable et aléatoire entre les joueurs

La livraison de cette étape nécessite de se focaliser uniquement sur la conception et l'implémentation du **standard deck** et de commencer à faire apparaître un **deck** prêt à être utilisé !

❑ Rapide phase de conception

Durant la quick design session, vous avez pris les notes suivantes reflétant les discussions de votre équipe de développement :

❑ Il y a **le paquet de cartes initial** celui qui contient toutes les 52 cartes, qui devra être mélangé et mis à la disposition de l'arbitre afin que ce dernier puisse le répartir équitablement entre les joueurs.

Remarque :

- la distribution est une responsabilité de l'arbitre
- mais le mélange est une responsabilité du paquet : le paquet est donc capable de se mélanger lui-même 😊
-

Nom du paquet	Standard Deck Paquet de cartes contenant les 52 cartes mélangées
Responsabilités du paquet	→ Mettre à disposition un nouveau jeu de 52 cartes mélangées

❑ **Chaque joueur dispose aussi d'un paquet de cartes** qui lui permet de jouer tout au long de la partie.

Le paquet de cartes sera constitué à partir d'un paquet de cartes provenant du jeu initial.

Lorsque la partie sera lancée, le joueur pourra piocher une carte dans ce paquet de cartes, en l'occurrence celle du haut et replacer en bas une carte ou les cartes qu'il a gagnées.

Le joueur pourra récupérer un nouveau paquet de cartes s'il a gagné une bataille, ou simplement une carte, c-a-d qu'il pourra ajouter des cartes à son paquet.

Il peut également être intéressant de savoir si le paquet est vide, dans ce cas-là le joueur sera éliminé de la partie

Il est important de savoir combien il reste de cartes dans le paquet pour que l'arbitre puisse déterminer

- si le joueur dispose encore d'assez de cartes (au moins deux pour procéder à une bataille)
- si le joueur ne dispose pas d'assez de cartes, il sera éliminé de la partie
- ou s'il a gagné en ayant récupéré toutes les cartes 😊

Il peut également être intéressant de pouvoir vider le paquet de toutes ses cartes (pour être sûr en début de partie de commencer avec un paquet vide avant que l'arbitre procède à la distribution du jeu initial).

Nom du paquet

Responsabilités du paquet
uniquement nécessaires à
la livraison de l'étape 1 ...

Deck	
Paquet de cartes du joueur	
	<ul style="list-style-type: none">→ ajouter une carte (en dessous) = > au début→ connaître le nombre de cartes (restantes) dans le paquet→ tirer la première carte du paquet (en dessous)→ éventuellement ajouter un paquet de cartes (en dessous) = > si le temps de développement le permet, sinon on fera avec ajouter une carte pour commencer 😊
	... les autres services seront identifiés et implémentés plus tard 😊

Remarque : le standard deck doit également être capable d'ajouter une carte et connaître le nombre de cartes pour pouvoir constituer et vérifier que le jeu comporte bien 52 cartes !!!!

Question : En tenant compte de tout ce qui précède et en ne représentant uniquement les services nécessaires à la bonne implémentation de l'étape 1, proposez un diagramme de classes mettant en œuvre les deux concepts de paquets de cartes : **standard deck** et **deck**.

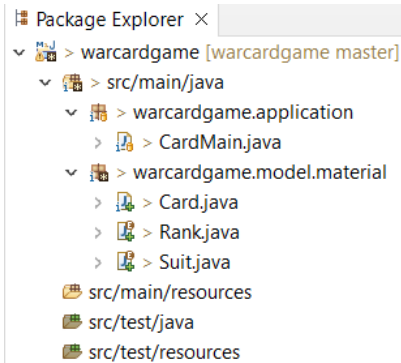
La dernière remarque montre que ces deux concepts ont des responsabilités communes. Etablissez une hiérarchie d'héritage entre ces deux concepts (et uniquement entre ces deux concepts, aucune classe abstraite n'est demandée pour le moment 😊)

❏ Phase d'implémentation et de tests 😊

Après les phases d'analyse et de conception vient la phase d'implémentation !

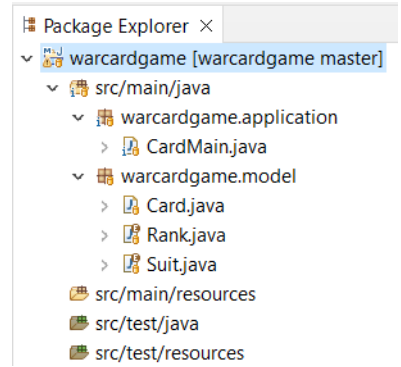
⇒ **Préliminaire** : réorganisation de la structure du projet warcardgame.

Pour l'instant, votre projet est structuré de la manière suivante ci-contre.

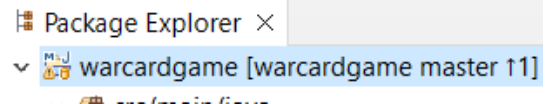


Faites apparaître un paquetage **warcardgame.model.material** dans lequel vous regrouperez les classes métiers : Card, Rank et Suit afin que le projet soit structuré de la manière ci-contre. N'oubliez pas de modifier les import pour CardMain pour qu'il n'y ait plus aucune erreur de compilation dans votre projet !

Effectuez un commit pour marquer ce changement avec un message du genre : **add warcardgame.model.material**



Remarque : Comme votre projet est lié à un remote, juste après votre commit, l'entête du projet dans votre IDE devient :



Le **↑1** qui vient d'apparaître à côté du nom de la branche master signifie que vous avez commitez 1 fois de plus en local que sur le remote, c-a-d que le local est en avance (**↑**) de 1 commit par rapport au remote.

Si vous décidez maintenant de **pousser (push)** sur le remote (soit en ligne de commande, soit avec gitkraken), vous pouvez constater que cette information disparaît. Quand il n'y a aucune information du genre (**↑x**) ou (**↓y**) : cela signifie que votre dépôt local et votre remote sont synchronisés !!!

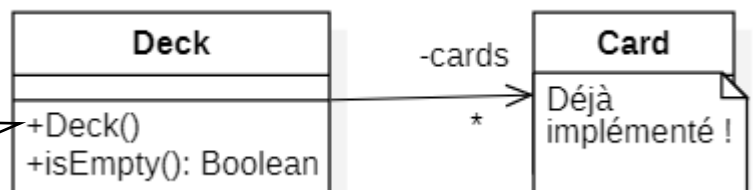
Vous avez donc compris que quand vous verrez (**↓y**), cela signifiera que vous êtes en retard sur le remote de y commits. Si vous travaillez sur une branche personnelle, vous pourrez tirer (pull) sans problème pour resynchroniser, si vous travaillez sur le **main**, soyez prudent !!!

1. Un paquet de cartes n'est autre qu'une collection de cartes

- Dans le code de production src/main/java**, commencez par créer une classe **Deck** dans un paquetage : **warcardgame.model.material.deck**
En effet, nous commençons par la classe Deck, puisque dans le diagramme de classes réalisé à la page précédente, vous avez dû identifier la classe Deck comme classe mère de l'héritage 😊.

L'instanciation se fera dans le constructeur :

```
public Deck() {  
    this.cards = new ArrayList<>();  
}
```



N'oubliez pas également de générer **hashCode** et **equals** avec l'aide de votre IDE :

Source → **Generate hashCode() and equals()...**

Pour vous faciliter la tâche et vous permettre de recopier les codes de test donnés,
Cet énoncé est également disponible sur :
<https://github.com/iblasquez/enseignement-but1-developpement>

- b. Dans le code de test src/test/java, créer une classe **DeckTest** dans un paquetage : **warcardgame.model.material.deck** dans laquelle vous implémenterez le premier test suivant :

```
@Test
void the_deck_is_empty_at_the_beginning() {
    Deck deck = new Deck();
    assertTrue(deck.isEmpty());
}
```

Afin de vérifier et de valider le code de production précédente, faites-en sorte que votre test passe AU VERT !

2. Une carte peut être ajoutée au paquet ...

- a. ⇒ **Dans le code de production**, ajoutez à la classe **Deck** la méthode suivante :

```
public Boolean put(Card card)
```

⇒ **Dans le code de test**, faites passer AU VERT dans la classe **DeckTest** le test suivant de manière à vérifier votre implémentation !

```
@Test
void the_deck_is_not_empty_when_a_card_put_in_the_deck() {
    Deck deck = new Deck();
    Boolean isPut = deck.put(new Card(Rank.ACE, Suit.HEARTS));

    assertTrue(isPut);
    assertFalse(deck.isEmpty());
}
```

- b. **Qualité du code de test** : A l'image du code de production, le code de test peut également être refactorisé pour faciliter l'écriture et la lisibilité des tests.
Sélectionnez la partie de code : `new Card(Rank.ACE, Suit.HEARTS)`
Puis à l'aide d'un clic droit, choisissez le menu **Refactor** → **Extract Constant** ...
et nommez cette constante : `ACE_OF_HEARTS`

- c. **Connaitre le nombre de cartes présentes dans le paquet** : Nous savons que le paquet n'est pas vide, mais pour le test précédent soit complet, il serait également intéressant de vérifier que le paquet contient bien une carte.

⇒ **Dans le code de production**, ajoutez à la classe **Deck** la méthode suivante :

```
public Integer remainingCards()
```

Remarque à propos du nommage de cette méthode : lors de la quick design session, vous aviez décidé avec votre équipe de développement que vous souhaiteriez connaître le nombre de cartes restantes dans le paquet de cartes.

⇒ **Dans le code de test**, ajoutez une troisième assertion dans les tests précédentes et faites passer AU VERT ces tests de manière à vérifier votre implémentation !

```
@Test
void the_deck_is_not_empty_when_a_card_put_in_the_deck() {
    Deck deck = new Deck();
    Boolean isPut = deck.put(ACE_OF_HEARTS);

    assertTrue(isPut);
    assertFalse(deck.isEmpty());
    assertEquals(1, deck.remainingCards());
}
```

⇒ **Refactoring du nom de votre code.**

Le nom de votre méthode de test peut « être revisitée » pour qu'elle soit plus explicite c-a-d qu'elle illustre mieux le comportement testée. Renommez votre méthode de test :

```
the_deck_is_not_empty_when_a_card_put_in_the_deck()
....en...
deck_has_one_card_when_a_card_put_in_an_empty_deck()
```

***Assurez-vous que tous vos tests passent AU VERT
avant de continuer et passer à l'écriture d'un autre test !!***

3. Deux cartes successives peuvent être ajoutée au paquet ...

⇒ Dans le code de test, implémentez dans la classe **DeckTest** ce nouveau test qui surcharge la méthode **put** précédente

```
@Test
void deck_has_2_cards_when_two_cards_put_in_an_empty_deck() {
    Deck deck = new Deck();
    Boolean isPut = deck.put(ACE_OF_HEARTS, KING_OF_DIAMONDS);

    assertTrue(isPut);
    assertFalse(deck.isEmpty());
    assertEquals(2, deck.remainingCards());
}
```

En ayant au préalable, déclarez :

```
private static final Card KING_OF_DIAMONDS = new Card(Rank.KING, Suit.DIAMONDS);
```

⇒ Dans le code de production, ajoutez à la classe **Deck** la méthode suivante pour faire compiler le test précédent et implémentez cette méthode avant de faire passer le test précédent AU VERT !

```
public Boolean put(Card card1, Card card2);
```

4. Trois cartes et plus successives peuvent être ajoutée au paquet ...

Dans le code de test, on pourrait continuer à écrire d'autres tests du même genre en surchargeant la méthode **put** :

→ pour trois cartes:

```
@Test
void deck_has_3_cards_when_3_tiles_put_in_an_empty_deck() {
    Deck deck = new Deck();
    Boolean isPut = deck.put(ACE_OF_HEARTS, KING_OF_DIAMOND, QUEEN_OF_CLUBS);
    //...
}
```

→ pour quatre cartes:

```
@Test
void deck_has_4_cards_when_4_tiles_put_in_an_empty_deck() {
    Deck deck = new Deck();
    Boolean isPut = deck.put(ACE_OF_HEARTS, KING_OF_DIAMOND, QUEEN_OF_CLUBS,
                            TEN_OF_SPADES);
    //...
}
```

→ pour n cartes ...

Ce qui donnerait dans le code de production, beaucoup (trop) de surcharges de la méthode **put** !

```
public Boolean put(Card card);
public Boolean put(Card card1, Card card2);
public Boolean put(Card card1, Card card2, Card card3);
public Boolean put(Card card1, Card card2, Card card3, Card card4);
//...
```



Vous la sentez la mauvaise odeur dans ce code ?
Heureusement
la notion **d'arguments variables (varargs)**
va venir à notre rescousse
pour nettoyer un peu ce code !!!

Elle permet en effet de passer un **nombre non défini**
d'arguments d'un même type à une méthode

Par exemple, dans le cas de la méthode put, on souhaite passer un nombre n de cartes (pas nécessairement 1, ou 2, mais un **nombre arbitraire** de cartes en fonction des cas qui vont se présenter).

Java propose le concept de **varargs** (...) pour implémenter facilement cette multiplicité de n , sans avoir à surcharger une méthode en fonction du nombre de cartes.

Dans le code de production, dans la classe Deck, mettez pour le moment les deux méthodes put surchargées en commentaires et écrivez la nouvelle méthode put suivante à la place :

```
public Boolean put(Card...otherCards) {  
    if (otherCards.length == 0)  
        return false;  
  
    for (Card card : otherCards) {  
        if (card != null)  
            this.cards.add(card);  
    }  
    return true;  
}
```

Compilez et exécutez les tests. Si les test continuent de passer AU VERT, c'est parfait : cela signifie que cette méthode remplace bien les deux précédentes, vous pouvez donc supprimer les méthodes en commentaires de votre code !

En savoir un peu plus sur les varargs :

Les **varargs** sont présentées brièvement dans <https://www.baeldung.com/java-varargs> qui rappelle les points suivants :

- **varargs are arrays so we need to work with them just like we'd work with a normal array.**

Les varargs seront donc traités comme **de simples tableaux** : vous pouvez retrouver cette information dans l'IDE en passant la souris sur `otherCards`, le message suivant devrait s'afficher :

```
public Boolean put(Card... otherCards) {  
    if (otherCards.length == 0)  
        ...  
}
```

Card[] otherCards - warcardgame.model.material.deck.Deck.put(Card...)

- *Each method can **only** have **one varargs parameter***
- *The varargs argument must be the **last parameter***

⇒ Dans le code de test ajoutez le code suivant et vérifiez que le code que vous venez d'implémenter fait également bien passer ce test AU VERT (et oui pour couvrir tout le code de cette méthode c'est mieux de tester le `assertTrue` et le `assertFalse` 😊)

```
@Test
void return_false_when_no_cards_put_in_the_deck() {
    Deck deck = new Deck();
    Boolean isPut = deck.put();

    assertFalse(isPut);
}
```

⚡ Interruption momentanée du projet warcardgame ⚡



A partir de maintenant, nous allons manipuler un(et même plusieurs) paquet(s) de carte, c'est-à-dire que nous ne nous intéresserons pas à un seul objet (de type `Card`), mais une **collection d'objets** (de type `Card`).

Ecrire des tests avec JUnit5 pour un seul objet est assez simple avec `assertEquals` et `assertTrue`

Ecrire des tests avec JUnit5 sur des collections d'objets est un peu plus complexe...

C'est pourquoi lorsque vous devez écrire des tests unitaires sur des collections d'objets, une bonne pratique consiste à utiliser le framework `AssertJ` que vous allez découvrir maintenant dans un petit projet « bac à sable »



Rendez-vous dans l'annexe pour découvrir et prendre en main `assertJ`

Vous réaliserez ce projet « bac à sable » dans le même workspace que le projet warcardgame ...

« Reprise du projet warcardGame »



Dans la suite, nos tests vont faire appel aux services offerts par AssertJ.

Avant de continuer, il vous faut donc ajouter dans votre **pom.xml** une dépendance vers ce framework.

Comme vu dans la prise en main d'AssertJ en annexe, rendez-vous dans la documentation d'AssertJ dans la rubrique **2.4 Quick start** et plus particulièrement dans **Maven** pour copier le bloc dependency à ajouter dans votre pom.xml. (<https://assertj.github.io/doc/#maven>)

Après avoir copié/collé ce bloc (entre les balises <dependency> et </dependency>) et sauvé votre pom.xml, n'oubliez pas de le updater (sous Eclipse) pour que ces modifications soient bien prises en compte 😊

5. Une carte a été ajoutée au paquet, mais le paquet contient-il bien la carte ajoutée et pas une autre ?

⇒ **Dans le code de test**, implémentez le test suivant, afin que son passage AU VERT permette de valider le comportement attendu.

```
@Test
void deck_contains_the_right_cards_put_in_an_empty_desk() {
    Deck deck = new Deck();
    deck.put(ACE_OF_HEARTS, KING_OF_DIAMONDS, QUEEN_OF_CLUBS, TEN_OF_SPADES);

    Collection<Card> cards = deck.cards();

    assertThat(cards).containsExactly(ACE_OF_HEARTS, KING_OF_DIAMONDS, QUEEN_OF_CLUBS, TEN_OF_SPADES)
        .containsSequence(ACE_OF_HEARTS, KING_OF_DIAMONDS, QUEEN_OF_CLUBS, TEN_OF_SPADES);
}
```

⇒ **Dans le code de production**, implémentez la méthode utilisée dans l'étape **Act** de ce test c-a-d :

```
public Collection<Card> cards()
```

6. Retirer d'un seul coup toutes les cartes d'un paquet :

⇒ **Dans le code de test**, implémentez ce nouveau test qui permet de vérifier qu'une fois toutes les cartes retirées du paquet (d'un seul coup), le paquet se retrouve bien vide (nettoyé) 😊

```
@Test
void deck_is_empty_when_it_is_cleared() {
    Deck deck = new Deck();
    deck.put(ACE_OF_HEARTS, KING_OF_DIAMONDS, QUEEN_OF_CLUBS, TEN_OF_SPADES);

    deck.clear();
    assertTrue(deck.isEmpty());
    assertThat(deck.remainingCards()).isZero();
}
```

⇒ **Dans le code de production**, implémentez **en une seule ligne** la méthode suivante afin de faire Passer le test précédent AU VERT :

```
public void clear()
```

7. Et si on permettait également d'ajouter une collection de cartes d'un seul coup au paquet ?

Les **arguments variables** (*varargs*) permettent d'ajouter plusieurs cartes successivement une à une dans le paquet dans une même méthode (à *n* paramètres d'entrée de type Card)

Dans certains cas d'usage, il se peut qu'il soit plus pratique d'ajouter directement d'un seul coup toute une collection de cartes (méthode à 1 paramètre de type Collection<Card>) 😊

⇒ Dans le code de test, implémentez les deux tests suivants :

```
@Test
void a_collection_of_cards_can_be_put_once_in_the_deck() {
    Deck deck = new Deck();
    Collection<Card> newCards = Arrays.asList(ACE_OF_HEARTS, KING_OF_DIAMONDS, QUEEN_OF_CLUBS);
    deck.put(newCards);

    assertThat(deck.remainingCards()).isEqualTo(3);
    assertThat(deck.cards()).containsExactly(ACE_OF_HEARTS, KING_OF_DIAMONDS, QUEEN_OF_CLUBS)
        .containsSequence(ACE_OF_HEARTS, KING_OF_DIAMONDS, QUEEN_OF_CLUBS);
}
```

```
@Test
void return_false_when_cards_collection_of_deck_does_not_change_() {
    Deck deck = new Deck();
    Collection<Card> newCards = new ArrayList<>();
    Boolean isPut = deck.put(newCards);

    assertFalse(isPut);
}
```

⇒ Dans le code de production, implémentez judicieusement (en un nombre minimum de lignes de code) la méthode suivante afin de faire passer le test précédent AU VERT :

```
public Boolean put(Collection<Card> otherCards)
```

8. Et bien sûr pour jouer, il est indispensable de pouvoir tirer la première carte du paquet !

⇒ Dans le code de test, implémentez les trois tests suivants :

```
@Test
void the_deck_is_empty_an_card_is_out_the_desk_when_the_only_card_drawn() {
    Deck deck = new Deck();
    deck.put(ACE_OF_HEARTS);

    Card cardDrawn = deck.draw();
    assertTrue(deck.isEmpty());
    assertThat(cardDrawn).isEqualTo(ACE_OF_HEARTS);
}
```

```

@Test
void first_card_of_the_deck_is_out_when_drawn() {
    Deck deck = new Deck();
    deck.put(KING_OF_DIAMONDS, ACE_OF_HEARTS, QUEEN_OF_CLUBS);

    Card cardDrawn = deck.draw();
    assertThat(cardDrawn).isEqualTo(KING_OF_DIAMONDS);
    assertThat(deck.remainingCards()).isEqualTo(2);
}

```

```

@Test
void return_null_when_drawn_with_an_empty_deck() {
    Deck deck = new Deck();

    Card cardDrawn = deck.draw();
    assertThat(cardDrawn).isNull();
}

```


⇒ Dans le code de production, implémentez la méthode suivante afin de faire passer les tests précédents AU VERT :

public Card draw()

Vérifiez que tous vos tests passent AU VERT et faites une **rétro-conception** de la classe **Deck** pour vérifier que vous avez bien implémenté la classe ci-contre.

9. Commitez !

Il ne vous reste plus qu'à commiter les classes **Deck** et **DeckTest** avec un message du genre :
« add Deck and DeckTest »





















































 Deck
<div> <div></div> <div>cards: List<Card></div> </div>
<ul style="list-style-type: none"> Deck() isEmpty(): boolean remainingCards(): Integer cards(): Collection<Card> clear(): void put(otherCards: Card[]): Boolean put(otherCards: Collection<Card>): Boolean draw(): Card hashCode(): int equals(obj: Object): boolean

... Et maintenant intéressons-nous au StandardDeck !

❑ Rapide phase de conception

Le jeu de cartes initial (ou standard deck) est le paquet de cartes initial constitués de 52 cartes. Il est créé une fois en début de partie et n'est utilisé qu'une seule fois en début de partie pour distribuer les cartes de manière équitable et aléatoire entre les joueurs

Example set of 52 playing cards; 13 of each suit: clubs, diamonds, hearts, and spades

	Ace	2	3	4	5	6	7	8	9	10	Jack	Queen	King
Clubs													
Diamonds													
Hearts													
Spades													

- ⇒ Le standard deck **EST-DONC** un deck qui lors de son instanciation devra :
- être capable fournir un jeu initial de 52 cartes (similaire à celui-ci-dessus)

Modélisez la phrase précédente sous forme de diagramme de classes 😊 !

❏ Phase d'implémentation et de tests du StandardDeck 😊

Dans le package `warcardgame.model.material.deck`, créez la classe **StandardDeck** suivante, qui hérite de **Deck**, et qui permet, à l'instanciation, de créer 52 cartes et de les regrouper dans **cards**.

```
public class StandardDeck extends Deck{

    public StandardDeck() {
        super();
        this.putAllCardsTogether();
    }
}
```

A l'aide de votre IDE, éliminez l'erreur de compilation en créant une méthode privée **putAllCardsTogether**

⇒ **Code de test** : récupérez le code de la classe `StandardDeckTest` dans le fichier gist suivant :

<https://unil.im/decktest>

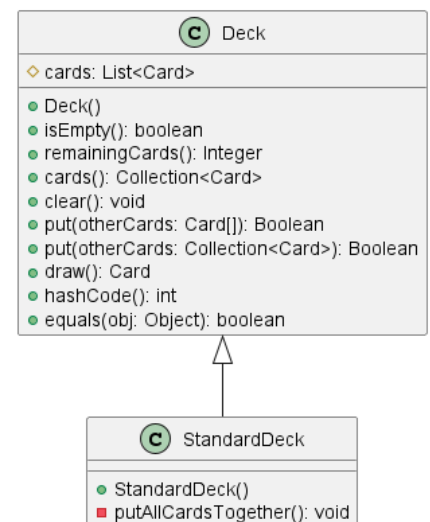
(<https://gist.github.com/iblasquez/1c5d96d4abac617dd2a86994ab14ac72>)

et ajoutez ce fichier au même niveau que le fichier de test `DeckTest`

⇒ **Code de production** : Implémentez avec un code minimum la méthode **putAllCardsTogether** de manière à faire passer AU VERT tous les tests du fichier `StandardDeckTest`.

Remarque : 3 lignes de code suffisent à implémenter cette méthode 😊 A vous de jouer !

⇒ **Rétroconception** : Procédez à une petite rétro-conception sur les classes du paquetage `warcardgame.model.material.deck` afin de vérifier que le design de votre code est actuellement conforme au diagramme de classes ci-contre.



⇒ **Commitez !** Il ne vous reste plus qu'à commiter les classes `Deck` et `StandardDeck` car avec l'héritage l'attribut `cards` a dû être modifié de *private* en **protected** 😊) avec un message du genre : « **add StandardDeck and StandardDeckTest** »

Annexe :

Découverte et prise en main de l'API *fluent* AssertJ

Les bonnes pratiques d'écriture d'un test recommandent d'utiliser le pattern AAA.

Le 3^{ème} A du pattern AAA est indispensable dans l'écriture d'un test automatisé car il fait référence à l'**Assertion**, une étape qui consiste à comparer le résultat attendu avec le résultat obtenu afin que le framework de tests automatisés puissent rendre son *verdict* : soit le test passe (et il est VERT), soit le test échoue (et il est ROUGE).

Pour l'instant, vous avez implémenté l'étape d'assertion en utilisant les méthodes de la classe **Assertions** du paquetage **org.junit.jupiter.api.Assertions** de Junit5 à l'image des méthodes qui proposent essentiellement des méthodes **assertEquals** (ou **assertNotEquals**) surchargées prenant en premier paramètre le résultat attendu et en second paramètre le résultat actuel :

static void	assertEquals (Long [Ⓢ] expected, Long [Ⓢ] actual)
static void	assertEquals (Long [Ⓢ] expected, Long [Ⓢ] actual, String [Ⓢ] message)
static void	assertEquals (Long [Ⓢ] expected, Long [Ⓢ] actual, Supplier [Ⓢ] <String [Ⓢ] > messageSupplier)
static void	assertEquals (Object [Ⓢ] expected, Object [Ⓢ] actual)
static void	assertEquals (Object [Ⓢ] expected, Object [Ⓢ] actual, String [Ⓢ] message)
static void	assertEquals (Object [Ⓢ] expected, Object [Ⓢ] actual, Supplier [Ⓢ] <String [Ⓢ] > messageSupplier)
static void	assertEquals (Short [Ⓢ] expected, short actual)
static void	assertEquals (Short [Ⓢ] expected, short actual, String [Ⓢ] message)
static void	assertEquals (Short [Ⓢ] expected, short actual, Supplier [Ⓢ] <String [Ⓢ] > messageSupplier)
static void	assertEquals (Short [Ⓢ] expected, Short [Ⓢ] actual)

Extrait de la javadoc de Junit5, pour visualiser toutes les services offerts par la classe Assertions, rendez-vous à l'adresse suivante 😊

<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html>

Maintenant que vous maîtrisez mieux cette étape d'assertion, nous vous proposons de découvrir une nouvelle API qui vous permettra d'écrire l'étape d'assertion de manière plus lisible, et améliorera ainsi la qualité de votre code de test. En effet, écrire des tests sur des collections peut s'avérer fastidieux et complexe avec les méthodes de bases de JUnit, alors qu'avec l'API AssertJ, l'écriture et la lecture des tests deviendra plus aisée, plus fluide (« fluente »), c'est pour cela qu'on qualifie souvent **AssertJ de *fluent* API**

AssertJ est disponible à l'adresse suivante : <https://assertj.github.io/doc/>

Comme l'indique la page d'accueil, AssertJ propose plusieurs modules, nous nous focaliserons pour commencer uniquement sur le **core module**.

AssertJ - fluent assertions java library

Version 1.0

1. AssertJ Overview

AssertJ is composed of several modules:

- A [core module](#) to provide assertions for JDK types (String, Iterable, Stream, Path, File, Map...)
- A [Guava module](#) to provide assertions for Guava types (Multimap, Optional...)
- A [Joda Time module](#) to provide assertions for Joda Time types (DateTime, LocalDateTime)
- A [Neo4j module](#) to provide assertions for Neo4j types (Path, Node, Relationship...)
- A [DB module](#) to provide assertions for relational database types (Table, Row, Column...)
- A [Swing module](#) provides a simple and intuitive API for functional testing of Swing user interfaces

❑ Pour découvrir **AssertJ**, nous vous proposons de commencer par **créer un nouveau projet maven** que vous appellerez **helloassertj** avec pour commencer un **pom** habituel (celui avec la dépendance à JUnit5).

❑ Vous allez ensuite créer une classe de tests JUnit 5 nommée **AssertJTest** dans `src/test/java/helloAssertj` à l'aide de **New→JUnit Test Case** qui contiendra un premier test écrit avec JUnit :

```
@Test
void test_hello_with_assertEquals_Junit5() {
    String hello = "hello AssertJ !";
    assertEquals("hello AssertJ !", hello);
}
```

Implémentez ce test et exécutez-le afin qu'il passe AU VERT 😊

❑ Avant de pouvoir écrire un test plus *fluent* faisant appel aux services offerts par AssertJ, il faut commencer par ajouter la **dépendance à cette API** dans votre **pom.xml**

Pour effectuer cela, rendez-vous donc dans la documentation d'AssertJ dans la rubrique **2.4 Quick start** et plus particulièrement dans **Maven** pour copier le bloc dependency à ajouter dans votre pom.xml.

(<https://assertj.github.io/doc/#maven>)

Après avoir copié/collé ce bloc (entre les balises `<dependency>` et `</dependency>`) et sauvegardé votre pom.xml, n'oubliez pas de le updaten (sous Eclipse) pour que ces modifications soient bien prises en compte 😊

2.4. Quick start

2.4.1. Supported Java Versions

2.4.2. Get AssertJ Core

Maven

Gradle

Other build tools

❑ Vous pouvez maintenant écrire votre premier test avec AssertJ :

```
@Test
void test_hello_with_assertThat_AssertJ() {
    String hello = "hello AssertJ !";
    assertThat(hello).isEqualTo("hello AssertJ !");
}
```

Sans oublier l'import qui va bien :

```
import static org.assertj.core.api.Assertions.*;
```

Implémentez ce test et exécutez-le afin qu'il passe AU VERT 😊

Remarques :

⇒ Il est important de noter que **AssertJ** ne remplace pas **JUnit**.

Vous avez absolument besoin d'utiliser **JUnit** pour pouvoir écrire une classe de tests et lancer vos tests unitaire. **AssertJ** va vous fournir des services (méthodes) supplémentaires pour écrire des assertions de manière plus fluide 😊

⇒ La version JUnit 5 : **assertEquals("hello AssertJ !", hello);**
devient avec AssertJ : **assertThat(hello).isEqualTo("hello AssertJ !");**

Pour la comparaison simple d'un objet, utiliser JUnit5 ou AssertJ, la différence entre **assertEquals** et **assertThat** peut paraître minime dans cet exemple. Toutefois utiliser **assertThat** permet d'écrire un code « plus sûr » dans le sens où il permet de comprendre directement que le premier paramètre correspond au résultat obtenu et le deuxième paramètre au résultat attendu. Quand vous écrivez un test avec **assertEquals**, n'avez-vous pas un doute si le premier paramètre à passer doit être le résultat attendu ou le résultat obtenu ?

Par contre, le framework **AssertJ** va devenir une vraie « pépite » lorsque vous devrez écrire des tests sur des collections de manière fluide comme nous allons le voir...

❑ AssertJ et les collections :

⇒ Implémentez le test suivant et exécutez-le afin de vérifier qu'il passe AU VERT 😊

```
@Test
void test_List_String() {
    List<String> list = Arrays.asList("1", "2", "3");
    assertThat(list).contains("1");
}
```

⇒ Ajoutez dans le test précédent l'assertion suivante et exécutez-le à nouveau pour vérifier qu'il passe toujours AU VERT !

```
assertThat(list).doesNotContain("5");
```

⇒ Ajoutez une à une à une chaque assertion et exécutez le test après ajout de chque assertion pour vérifier que ce test passe bien au VERT 😊

```
@Test
void test_List_String() {
    List<String> list = Arrays.asList("1", "2", "3");

    assertThat(list).contains("1");
    assertThat(list).doesNotContain("5");
    assertThat(list).isNotEmpty();
    assertThat(list).startsWith("1");
    assertThat(list).containsSequence("2", "3");
    assertThat(list).doesNotContainSequence("3", "2");
    assertThat(list).containsExactly("1", "2", "3");
}
```

⇒ En fait, lorsqu'un test contient plusieurs assertions sur le même objet, la bonne pratique avec AssertJ consiste à « chaîner » ces assertions pour rendre la lecture du test plus fluid. Ecrivez donc maintenant le test suivant à la suite du précédent et exécutez-le pour vérifier qu'il passe bien AU VERT 😊

```
@Test
void test_List_String_best_practice()
{
    List<String> list = Arrays.asList("1", "2", "3");

    assertThat(list).contains("1")
        .doesNotContain("5")
        .isNotEmpty()
        .startsWith("1")
        .containsSequence("2", "3")
        .doesNotContainSequence("3", "2")
        .containsExactly("1", "2", "3");
}
```

⇒ Implémentez également le test suivant et exécutez-le pour vérifier qu'il passe AU VERT 😊

```
@Test
void test_Character()
{
    Character someCharacter = 'i';

    assertThat(someCharacter)
        .isNotEqualTo('a')
        .inUnicode()
        .isGreaterThanOrEqualTo('b')
        .isLowerCase();
}
```

❏ Documentation sur le site officiel de AssertJ

A partir de la première page du site officiel d'AssertJ : <https://assertj.github.io/doc/>

⇒ rendez-vous dans la partie **2.1 What is AssertJ Core ?** pour visualiser un bel ensemble d'exemples d'assertions **AssertJ** autour des collections qui vous montre toute la puissance d'utilisation de ce framework :

```
// entry point for all assertThat methods and utility methods (e.g. entry)
import static org.assertj.core.api.Assertions.*;

// basic assertions
assertThat(frodo.getName()).isEqualTo("Frodo");
assertThat(frodo).isNotEqualTo(sauron);

// chaining string specific assertions
assertThat(frodo.getName()).startsWith("Fro")
    .endsWith("do")
    .isEqualToIgnoringCase("frodo");

// collection specific assertions (there are plenty more)
// in the examples below fellowshipOfTheRing is a List<TolkienCharacter>
assertThat(fellowshipOfTheRing).hasSize(9)
    .contains(frodo, sam)
    .doesNotContain(sauron);

// as() is used to describe the test and will be shown before the error message
assertThat(frodo.getAge()).as("check %s's age", frodo.getName()).isEqualTo(33);

// exception assertion, standard style ...
assertThatThrownBy(() -> { throw new Exception("boom!"); }).hasMessage("boom!");
// ... or BDD style
Throwable thrown = catchThrowable(() -> { throw new Exception("boom!"); });
assertThat(thrown).hasMessageContaining("boom");

// using the 'extracting' feature to check fellowshipOfTheRing character's names
assertThat(fellowshipOfTheRing).extracting(TolkienCharacter::getName)
    .doesNotContain("Sauron", "Elrond");

// extracting multiple values at once grouped in tuples
assertThat(fellowshipOfTheRing).extracting("name", "age", "race.name")
    .contains(tuple("Boromir", 37, "Man"),
        tuple("Sam", 38, "Hobbit"),
        tuple("Legolas", 1000, "Elf"));

// filtering a collection before asserting
assertThat(fellowshipOfTheRing).filteredOn(character -> character.getName().contains("o"))
    .containsOnly(aragorn, frodo, legolas, boromir);

// combining filtering and extraction (yes we can)
assertThat(fellowshipOfTheRing).filteredOn(character -> character.getName().contains("o"))
    .containsOnly(aragorn, frodo, legolas, boromir)
    .extracting(character -> character.getRace().getName())
    .contains("Hobbit", "Elf", "Man");

// and many more assertions: iterable, stream, array, map, dates, path, file, numbers, predicate, optional ...
```

⇒ **Les parties 2.5 Core Assertion Guide et 2.6 Extending Assertions** expliquent à l'aide de nombreux exemples les possibilités offertes par ce framework. Laissez toujours un onglet sur votre navigateur ouvert sur cette partie (<https://assertj.github.io/doc/#assertj-core-assertions-guide>) qui pourra grandement vous aider dans l'écriture de vos tests AssertJ 😊

⇒ sinon, comme indiqué au début de la partie **2.AssertJ**, la javadoc assertj-core est consultable à l'adresse suivante : <https://www.javadoc.io/doc/org.assertj/assertj-core> (un conseil, laissez également cet onglet ouvert 😊)

❏ Zoom sur AssertJ et l'extraction des propriétés des objets d'une collection

Les méthodes **extracting** (et **flatExtracting**) permettent d'extraire une ou plusieurs propriétés (attributs) des objets d'une collection soit via une fonction soit via le nom de la propriété.

En effet, face à une liste d'objet complexe, il est parfois plus simple de tester un sous-ensemble des propriétés des objets de la liste plutôt que les objets en entier.

⇒ Avant d'écrire un nouveau test, nous allons ajouter du code de production. Dans **src/main/java**, ajoutez donc les classes suivantes :

```
public enum Sport {
    Handball, Judo;
}

public class SportMan {
    private final String firstName;
    private final String lastName;
    private final Sport sport;

    public SportMan(String firstName, String lastName, Sport sport) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.sport = sport;
    }

    public String whoAmI() {
        return this.firstName + " " + this.lastName + " - " + this.sport;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public Sport getSport() {
        return sport;
    }

    // ne pas oublier de générer hashCode et equals 😊
}
```


⇒ Revenir dans le code de test (**src/test/java**) et ajoutez une classe de test **SportManTest**.
Récupérez le contenu de la classe **SportManTest** à partir du gist suivant : <https://unil.im/testassertj>
(URL réelle : <https://gist.github.com/iblasquez/ad3a030b508f534e32c8bab358bba87>)

Exécutez la classe **SportManTest** pour vérifier que tous les tests passent AU VERT 😊
Prendre le temps de lire le code de la classe **SportManTest** et de comprendre les tests écrits, en vous aidant si nécessaire de la documentation donnée par le site officiel de **AssertJ** présentée précédemment. Vous pouvez modifier les tests ou en ajouter d'autres si le cœur vous en dit 😊

**Vous venez de faire une rapide introduction au framework AssertJ
qui nous suffira pour continuer cette séance de TP.
Fermer le projet helloassertj et revenez au projet warcardgame 😊**

Pour aller plus loin ...

Si, plus tard, vous souhaitez continuer à prendre en main, ce framework, voici quelques liens qui pourraient vous être utiles :

❑ le tutoriel de Vogella sur AssertJ disponible à l'adresse suivante :
<https://www.vogella.com/tutorials/AssertJ/article.html>

❑ Retour vers Tolkien ...

✓ Dans le projet **helloassertj** :

→ dans le **code de production** (**src/main/java**), un package **helloassertj.tolkien**

Dans lequel, vous ajouterez l'énumération **Race** et la classe **Tolkien** disponible dans le gist suivant : <https://unil.im/tolkien>
(<https://gist.github.com/iblasquez/8bdd3e28a08ae3e2dfec6c84cab156e5>)

→ dans le **code de test** (**src/test/java**), dans un package **helloassertj.tolkien**, ajoutez le fichier test **TolkienChecksWithAssertJTests** également disponible dans le même gist que précédemment. Exécutez ce fichier de test afin de constater que les tests échouent bien 😊

✓ En utilisant les services offerts par **AssertJ**, faites passer AU VERT les tests UN à UN, de manière à ce que tous les tests de ce fichier de tests passent AU VERT !

✓ Une fois que tous vos tests passent AU VERT, améliorez la qualité de code de votre fichier test en organisant les imports afin de garder uniquement les imports nécessaires au bon fonctionnement du code.

❑ Pour en savoir plus sur AssertJ

→ **Site officiel** :

<https://assertj.github.io/doc/>
<https://www.javadoc.io/doc/org.assertj/assertj-core>

→ **Tutoriels** :

<https://www.baeldung.com/introduction-to-assertj>
<https://www.vogella.com/tutorials/AssertJ/article.html>
<https://www.petrikainulainen.net/programming/testing/junit-5-tutorial-writing-assertions-with-assertj/>
<https://pieces-of-code.com/guide/quickstart/assertj.htm> (en français)