

# Langage Orienté Objet / Java

## *Partie 2 – Héritage, Random et E/S*

### The Learning Souls Game

Ce TP doit être réalisé à partir du code créé dans la partie précédente

#### 1. Classe Character



Dans la partie 1, nous avons créé la classe **Monster** en procédant par copier/coller de la classe **Hero** de manière à en dupliquer les mécanismes. Comme cela a été souligné précédemment, il s'agit d'une très mauvaise approche. En effet, si l'on veut par exemple modifier un de ces mécanismes communs, il est pour l'instant nécessaire d'apporter la modification dans les 2 copies du code... sans en oublier.... et sans se tromper. On peut aussi imaginer que plusieurs autres futures classes aient besoin de ces mécanismes. L'approche par copier/coller entraînerait la création d'autant de duplicatas, et rendrait le programme tout entier de plus en plus difficile à maintenir. Il faut retenir que d'une manière générale, en informatique, l'utilisation du copier/coller pour reproduire les mêmes mécanismes doit fortement questionner, car elle est souvent le symptôme d'un problème de conception.

Dans l'approche **Objet**, la mise en commun de mécanisme doit être réalisée par la **généralisation** et/ou la **spécialisation**, en utilisant le mécanisme d'**héritage**.

Les classes **Hero** et **Monster** ont ici des **membres** (**attributs** et **méthodes**) dupliqués du fait qu'elles sont en réalité des spécialisations d'un même concept que l'on peut nommer « personnage » (character en anglais).

1.1. Créez la classe **characters.Character** en y intégrant toutes les parties communes des classes **Hero** et **Monster**.

1.2. Faites hériter **Hero** et **Monster** de **Character**, et nettoyez les du code superflu.

-  Pour afficher dynamiquement le nom de la classe (le « [ Hero ] » et « [ Monster ] »), vous utiliserez un appel à **getClass().getSimpleName()**. Cherchez ces méthodes dans la documentation officielle Java. Comment cela fonctionne-t-il ?
-  Il faudra certainement passer des membres de **private** à **protected**. Pourquoi ?

- 1.3. Transformez la méthode **toString()** en retirant les « \t » (qui provoquent des alignements peu fiables) et en gérant les alignements grâce à la méthode (**static**) **format** de la classe **String**

 Utilisez des formats tels que « %-20s »

NB : comme vous l'avez certainement noté, grâce à notre meilleure conception Objet et au mécanisme d'héritage, la modification de **toString()** n'est faite qu'à un seul endroit (dans la classe **Character**), et fonctionne de manière cohérente pour les sous-classes que sont **Hero** et **Monster** !

## 2. Classe Dice



Notre jeu a besoin d'un moyen pour laisser une part au hasard. Par exemple, lorsqu'un personnage porte un coup, un nombre tiré aléatoirement en déterminera la précision. Pour ce faire, et comme cela existe classiquement dans les RPG, nous allons donner un dé (**Dice**) à chaque personnage. Lorsqu'une action aura besoin d'un tirage aléatoire, le personnage pourra ainsi lancer son dé.

Chaque personnage possède son (ou ses) propres dés : nous allons donc créer une classe **Dice** qui permettra de fournir à chacun une (ou plusieurs) instance(s) de dé.

Chaque dé possède un nombre fixe de faces qui est défini lors de son instanciation (donc dans le constructeur). Tirer un nombre aléatoire correspond à faire rouler (**roll**) le dé.

Les dés que nous allons créer seront un peu particuliers puisque leur 1ere face portera le chiffre 0, et la face la plus « grande » portera le nombre correspondant au nombre de faces -1. En d'autres termes, un dé à 50 faces fournira des nombres entre 0 et 49.


- 2.1. Créez la classe **lsg.helpers.Dice** avec les attributs d'instance :

 **faces** : entier qui contiendra le nombre de faces du dé.  
 **random** : de type **java.util.Random** et qui servira à générer les séries de nombres aléatoires.

- 2.2. Créez un constructeur à 1 paramètre permettant de fixer le nombre de faces du dé.

- 2.3. Créez la méthode d'instance **roll()** qui retourne un entier correspondant au résultat du lancer de dé.

- 2.4. Créez une méthode **main** qui nous servira simplement à tester le dé :






 on crée un dé à 50 faces, on le lance 500 fois, on conserve le plus petit tirage et le plus grand, et on vérifie que la plus petite valeur tirée est bien 0, et que la plus grande est bien 49.

Exemple d'exécution du **main** dans la console :

12 36 46 7 48 42 49 30 0 14 13 48 41 6 22 12 25 24 14 2 33 ...
Min : 0
Max : 49

### 3. Classe Weapon

3.1. Créez la classe **lsg.weapons.Weapon** avec les attributs :

-  **name** : une chaîne contenant le nom « humanisé » de l'arme.
-  **minDamage** : un entier déterminant les dommages minimums qui seront causés par l'arme.
-  **maxDamage** : un entier déterminant les dommages maximums pouvant être causés par l'arme. (Ce montant sera atteint si le personnage porte un coup avec une précision de 100%)
-  **stamCost** : le coût en force pour une frappe avec l'arme
-  **durability** : le nombre de coups pouvant être portés avec l'arme avant qu'elle ne casse.

3.2. Créez un constructeur à 5 paramètres permettant de fixer la valeur des attributs lors de l'instanciation.

3.3. Créez les getters correspondant aux attributs

3.4. Créez un setter privé pour **durability**.


NB1 : Le passage par un setter pour modifier la durabilité en interne facilitera une éventuelle future mise en place d'un mécanisme d'écouteurs sur cet attribut.

NB2 : les autres attributs n'ont a priori pas vocation à être modifiés.

3.5. Créez la méthode **use()** qui a pour effet de décrémenter la **durability** de 1.

3.6. Créez la méthode **isBroken()** qui renvoie un booléen indiquant si l'arme est cassée (**durability** inférieure ou égale à 0).

3.7. Surchargez la méthode **toString()** pour fournir une représentation de l'arme dans la console.

-  Exemple avec une arme portant le nom de « Basic Sword » :  
Basic Sword (min:5 max:10 stam:20 dur:100)

### 4. Classe Sword

Créez la classe **lsg.weapons.Sword** dont les instances porteront le nom « **Basic Sword** » et qui aura les statistiques initiales indiquées dans l'exemple précédent.

## 5. Classe Character

Le calcul des dégâts correspondants à un coup porté par un personnage avec une arme dépend des statistiques de l'arme, du personnage, et d'un tirage aléatoire.

- ✎ Un coup porté avec une arme cassée fera toujours 0 points de dégâts.
- ✎ Quand l'arme n'est pas cassée, la valeur des dégâts portés se situe toujours entre le **minDamage** et le **maxDamage** de l'arme.

Un lancer de dé du personnage permet de déterminer la précision de l'attaque (entre 0% et 100%), et donc les dégâts additionnels au delà de **minDamage**, sans que le total **minDamage + dégât additionnels** puissent dépasser **maxDamage**.

Ainsi, une attaque avec une précision de 0% (un lancer de dé à 0) générera **minDamage** de dégâts. Une attaque avec une précision de 100% (un lancer à 100) générera des dégâts additionnels portant le total à **maxDamage**. Une précision intermédiaire générera des dégâts additionnels proportionnels à la valeur tirée.

- ✎ Lorsqu'un personnage porte un coup avec une arme, il consomme la force correspondant à l'arme utilisée : sa **stamina** décroît donc en conséquence après le coup.

Lorsqu'un personnage porte un coup avec une arme sans avoir la **stamina** nécessaire, le montant des dégâts résultant de l'attaque est amoindri en proportion.

Ainsi, une attaque avec des dégâts calculés à 10 (en fonction de l'arme et suite au lancer de dé) pour un personnage avec assez de **stamina** par rapport au cout de l'arme générera effectivement 10 points de dégâts.

Par contre, cette même attaque portée par un personnage ne possédant que la moitié de la **stamina** nécessaire à l'utilisation de l'arme ne générera finalement que 5 points (la moitié) des dégâts calculés. NB : porter une attaque sans avoir assez de **stamina** a pour effet de mettre la **stamina** du personnage à 0.

D'après ce mode de calcul, un personnage avec 0 de **stamina** générera forcément 0 points de dégâts, quelle que soit l'arme.

- 5.1. Faites en sorte que chaque personnage créé possède automatiquement un dé à 101 faces (pour effectuer des tirages au sort entre 0 et 100).

5.2. Créez la méthode **attackWith(Weapon weapon)** qui renvoie un entier correspondant aux dégâts portés par le personnage avec l'arme passée en paramètre selon les règles de calcul évoquées ci-avant.

- ✎ Utilisez **Math.round** pour faire des arrondis (cf. doc officielle)
- ✎ Testez votre méthode dans le main de **LearningSoulsGame** en instanciant un **Hero**, et en le faisant frapper en boucle avec une instance de **Sword**.

Exemple :

[ Hero ]	Ynovator	LIFE:	100	STAMINA:	50	(ALIVE)
attaque avec Basic Sword (min:5 max:10 stam:20 dur:100) > 8						
[ Hero ]	Ynovator	LIFE:	100	STAMINA:	30	(ALIVE)
attaque avec Basic Sword (min:5 max:10 stam:20 dur:99) > 6						
[ Hero ]	Ynovator	LIFE:	100	STAMINA:	10	(ALIVE)
attaque avec Basic Sword (min:5 max:10 stam:20 dur:98) > 4						
[ Hero ]	Ynovator	LIFE:	100	STAMINA:	0	(ALIVE)
attaque avec Basic Sword (min:5 max:10 stam:20 dur:97) > 0						
[ Hero ]	Ynovator	LIFE:	100	STAMINA:	0	(ALIVE)
attaque avec Basic Sword (min:5 max:10 stam:20 dur:96) > 0						

## 6. Classe LearningSoulsGame

Reprenez votre test en instanciant cette fois un **Hero**, une **Sword** et un **Monster**.

- ✎ Lancez plusieurs attaques avec le héros.
- ✎ Lancez plusieurs attaques avec le monstre.
- ✎ Que remarquez-vous au niveau de la durabilité de l'épée ? Pourquoi ?

## 7. Classe Character, attack()

7.1. Pour que chaque **Character** utilise sa propre arme, créez un attribut **weapon** de type **Weapon** qui fournira une référence vers l'arme équipée, ainsi que les accesseurs correspondants.

7.2. Créez une méthode publique **attack()** sans paramètre qui correspond à une attaque avec l'arme équipée, puis passez la méthode **attackWith** précédemment définie en **private**.

7.3. Testez le tout sur des **Hero** et **Monster** dans le **main** de **LearningSoulsGame**.

## 8. Classe Character, getHitWith

Lorsqu'un personnage reçoit des dégâts (en étant la cible d'une attaque, en tombant d'une falaise, ...), il perd des points de vie. Le nombre de points de vie retirés dépend des dégâts reçus mais ne sera pas forcément égal : les dégâts pourront par exemple (plus tard dans notre projet) être atténués par une armure.

La méthode **public int getHitWith(int value)** calcule le nombre de points de vie (PV) effectivement retirés au personnage en fonction d'un montant de dégâts reçus. Cette méthode retourne le nombre de PV retirés de manière à par exemple pouvoir les afficher à l'écran.

Dans une première version (nos héros n'ont pas encore d'armure), le nombre de PV retirés (et retourné) sera effectivement égal au montant des dégâts passés en paramètres, sauf dans le cas où le personnage n'a plus assez de vie. En effet, nous ne voulons pas que la vie d'un personnage puisse avoir un montant négatif donc, lorsque le montant de dégâts est supérieur à la vie restante, la vie tombe à 0 et le nombre retourné correspond au nombre de PV qui restaient.

8.1. Ecrivez la méthode **public int getHitWith(int value)** en utilisant un *opérateur ternaire* plutôt qu'un **if** pour calculer le nombre de PV retirés.

8.2. Créez une nouvelle arme **lsg.weapons.ShotGun** telle que :  
ShotGun (min:6 max:20 stam:5 dur:100)

8.3. Testez vos méthodes dans **LearningSoulsGame** en laissant un **Hero** ayant équipé un **ShotGun** attaquer un **Monster** jusqu'à ce qu'il meure, ou jusqu'à ce que le héros n'ait plus assez de **stamina** pour utiliser efficacement son arme.

### Exemple de trace d'exécution :

NB : l'arme donnée à Rick a été instanciée ainsi : new Weapon("ShotGun", 6, 20, 5, 100)

[ Hero ]	Rick	LIFE: 100	STAMINA: 50	(ALIVE)
[ Monster ]	Zombie	LIFE: 10	STAMINA: 10	(ALIVE)
!!! Rick attack Zombie with Shotgun (9) !!! -> Effective DMG: 00009 PV				
[ Hero ]	Rick	LIFE: 100	STAMINA: 45	(ALIVE)
[ Monster ]	Zombie	LIFE: 1	STAMINA: 10	(ALIVE)
!!! Rick attack Zombie with Shotgun (11) !!! -> Effective DMG: 00001 PV				
[ Hero ]	Rick	LIFE: 100	STAMINA: 40	(ALIVE)
[ Monster ]	Zombie	LIFE: 0	STAMINA: 10	(DEAD)

## 9. Classe Claw

Le combat précédent n'était pas très équitable : nous allons rééquilibrer la partie en créant une arme digne de nos monstres.

9.1. Créez la classe **Claw** (griffe...) avec les statistiques suivantes :

Bloody Claw (min:50 max:150 stam:5 dur:100)

## 10. Classe LearningSoulsGame

Les combats de notre héros vs les monstres étant amenés à devenir de plus en plus longs et complexes, nous allons organiser et étendre la classe **LearningSoulsGame** en créant différentes méthodes facilitant le maintien et le déclenchement de tâches répétitives sur son « plateau » de jeu. Dans cette première version, le jeu opposera simplement un héros et un monstre.






10.1. Créez 2 attributs d'instance **hero** et **monster** destinés à recevoir nos 2 protagonistes.

10.2. Créez un attribut d'instance **scanner** du type **java.util.Scanner** qui sera utilisé pendant la partie pour lire les entrées faites au clavier par le joueur.

```
Scanner scanner = new Scanner(System.in) ;
```

10.3. Ecrivez une méthode d'instance **refresh** qui affiche dans la console et à la suite les statistiques de **hero**, puis celles de **monster**.

10.4. Créez une méthode d'instance **fight1v1** qui permet au héros et au monstre de lancer chacun leur tour une attaque sur l'autre personnage.

-  La partie démarre par l'affichage des statistiques.
-  A chaque tour, le joueur est invité à appuyer sur la touche **ENTREE** pour lancer l'action suivante : cf. la documentation officielle de **Scanner** et la méthode **String nextLine()**.
-  Le combat démarre par une attaque du héros.
-  Chaque attaque est affichée dans la console, puis les (nouvelles) statistiques sont affichées.
-  Le combat se termine lorsqu'un des personnages est mort. Un message indique alors qui a remporté le combat.


*NB : tout ceci n'est encore qu'une version beta et on pourra remarquer qu'il y aura une boucle infinie si les 2 protagonistes tombent à cours de **stamina** avant la fin du combat...*

10.5. Créez une méthode d'instance **init** qui

 instancie **hero** avec une **Sword**


 instance **monster** avec une **Claw**

10.6. Créez une méthode d'instance **play\_v1** qui

 initialise la partie (**init**)

 lance le **fight1v1**

✎ Testez **play\_v1** sur un **LearningSoulsGame** qu'on instanciera dans le **main**.

 *En option: vous pouvez lancer plusieurs combats et éventuellement changer les statistiques de nos classes (pv, armes, ...) pour tenter d'équilibrer le jeu...*

Ex. d'exécution :

[ Hero ]	Ynovator	LIFE:	100	STAMINA:	50	(ALIVE)
[ Monster ]	Monster_1	LIFE:	10	STAMINA:	10	(ALIVE)
Hit enter	key for next move >					
Ynovator	attacks Monster_1 with Basic Sword		(ATTACK:7   DMG : 7)			
[ Hero ]	Ynovator	LIFE:	100	STAMINA:	30	(ALIVE)
[ Monster ]	Monster_1	LIFE:	3	STAMINA:	10	(ALIVE)
Hit enter	key for next move >					
Monster_1	attacks Ynovator with Bloody Claw (ATTACK:78   DMG : 78)					
[ Hero ]	Ynovator	LIFE:	22	STAMINA:	30	(ALIVE)
[ Monster ]	Monster_1	LIFE:	3	STAMINA:	5	(ALIVE)
Hit enter	key for next move >					
Ynovator	attacks Monster_1 with Basic Sword (ATTACK:6   DMG : 3)					
[ Hero ]	Ynovator	LIFE:	22	STAMINA:	10	(ALIVE)
[ Monster ]	Monster_1	LIFE:	0	STAMINA:	5	(DEAD)
--- Ynovator WINS !!! ---						