## Mini jeu - il veulent tous ta POO

---

# Mini jeu - il veulent tous ta POO

On va coder un petit jeu à la sauce Fornite en Programmation Orienté Objet

## 1. Introduction

Maintenant que tu as tous les outils de POO en main, on va te montrer comment les utiliser pour faire un petit jeu de combat du type Fortnite. Alors évidemment, je vais tout de suite calmer tes ardeurs : on va devoir se contenter de visuels bien moins ambitieux que le vrai jeu… Mais le principe sera là : des joueurs vont s'affronter dans une grande arène, s'échanger des coups, et tu ne pourras vaincre que si tu es le dernier debout ! 😎

## 2. Le projet

On va faire ce projet de façon progressive : on va déjà produire une version 1.0 en te guidant, puis on passera à une version 2.0 un peu plus recherchée et moins guidée pour conclure sur une version 3.0 où on te donnera juste les directives !

C'est un projet relativement long et complet : l'objectif est a minima que tu puisses finir la version 1.0 pour valider le projet (il faut bien sûr qu'elle marche !). Mais on est persuadé qu'en vous aidant les uns les autres, vous arriverez à aller plus loin !

## 2.1. Mise en place du repo

On commence toujours par préparer son espace de travail. Comme pour le projet d'hier, ouvre un dossier du genre `mini_jeu_POO` et donne-lui l'architecture suivante :

```
mini_jeu_POO
├── lib
│   ├── player.rb
│   └── game.rb
├── app.rb
├── README.md
├── Gemfile
├── Gemfile.lock
└── Autres fichiers (.env, .gitignore)
```

Dans le fichier `app.rb`, insère le code suivant :

```
require 'bundler'
Bundler.require

require_relative 'lib/game'
require_relative 'lib/player'

binding.pry
```

Pour rappel, ces lignes vont te permettre d'exécuter ton programme proprement depuis `app.rb` en rendant toutes les gems disponibles dans tous les fichiers (2 premières lignes) et en faisant appel aux 2 autres fichiers présents dans `\lib` (les 2 lignes suivantes).

La ligne `binding.pry` est là pour te permettre de faire des tests. Tout ce que tu vas coder dans `player.rb`, tu pourras le tester dans le terminal (avec PRY) en exécutant directement un petit `ruby app.rb`.

## 2.2. Version 1.0 : coder un combat entre 2 joueurs

Quand on s'attaque à un gros morceau comme Fortnite, il y a un max de fonctionnalités qu'on peut avoir envie de coder. Alors pour ne pas se perdre, commençons par quelque chose qui soit à la fois **relativement simple à coder et central au jeu**. Ici, j'ai choisi de te faire commencer par un combat. Tout simplement. Si on est pas capable de coder un combat entre 2 personnages, on ne saura pas coder ce jeu : commençons donc par ça !

Que faut-il pour faire un combat ? Moi j'ai en tête qu'il faut a minima :

- 2 joueurs ;
- Que chaque joueur ait un niveau de vie donné ;
- Que ce niveau de vie baisse à chaque attaque subie ;
- Si la vie atteint zéro, le personnage est mort.

Difficile de faire plus simple... Alors allons-y ! On va commencer par coder tout ça dans le fichier `player.rb` qui va donc accueillir la classe `Player` dont le but est de modéliser un joueur. Je vais te décrire chaque caractéristique d'un objet `Player`, charge à toi d'écrire le code !

### a) `Player` : attributs et `initialize`

Un joueur possède 2 attributs que tu vas mettre en `attr_accessor` : un nom `@name` (string) et un niveau de vie `@life_points` (integer).

Quand on veut créer un objet `Player`, on ne met que son nom en entrée car le niveau de vie est le même pour tout le monde au début (10 pts de vie). Écris la méthode `initialize` afin qu'on ait la réaction suivante si on lance `app.rb` et qu'on utilise PRY :

```
[1] pry(main)> player1 = Player.new("José")
=> #<Player:0x000055e2ae15e910 @life_points=10, @name="José">
[2] pry(main)> player1.name
=> "José"
```

```
[3] pry(main)> player1.life_points
=> 10
```

### b) `Player` : afficher l'état d'un joueur avec `show_state`

On sait que les joueurs vont se mettre sur la tronche et que donc leur niveau de vie va baisser. Donc on peut anticiper qu'il va falloir afficher à l'utilisateur l'état de chaque joueur pour qu'il sache ce qu'il se passe et comment le combat avance. Code une méthode `show_state` qui va afficher l'état de l'objet `Player` sur laquelle elle est exécutée : "XXXX a YYY points de vie".

Tout comme dans l'exemple ci-dessus, cette méthode doit permettre d'obtenir le résultat suivant si on lance `app.rb` et qu'on utilise PRY:

```
[1] pry(main)> player1 = Player.new("José")
=> #<Player:0x000055c8d3bcb960 @life_points=10, @name="José">
[2] pry(main)> player1.show_state
José a 10 points de vie
=> nil
```

### c) `Player` : subir une attaque avec `gets_damage`

A ce stade, on a notre base pour définir un objet `Player`. Passons maintenant à une méthode indispensable pour un combat : celle qui fait baisser le niveau de vie du joueur. En gros je veux pouvoir modéliser "le joueur stocké dans l'objet `player1` subit 5 points de dommage" en faisant un `player1.gets_damage(5)`.

Tu dois coder une méthode `gets_damage` qui prend en entrée un integer (= le nombre de dommages subit) et qui le soustraie au niveau de vie (`@life_points`) du joueur.

Une fois la soustraction faite, la méthode vérifie si `@life_points` est inférieur ou égale à zéro. Si c'est le cas, c'est que le joueur est mort : elle affiche un message "le joueur XXXX a été tué !".

Voici le fonctionnement qu'on doit obtenir en exécutant `app.rb` et en utilisant PRY :

```
[1] pry(main)> player1 = Player.new("José")
=> #<Player:0x00005648d5f9ea88 @life_points=10, @name="José">
[2] pry(main)> player1.gets_damage(5)
=> nil
[3] pry(main)> player1.show_state
José a 5 points de vie
=> nil
[4] pry(main)> player1.gets_damage(5)
le joueur José a été tué !
=> nil
[5] pry(main)> player1.show_state
José a 0 points de vie
=> nil
```

Petite aide : pour ceux qui sont complètement perdus dans l'écriture de cette première méthode, son contenu est affiché ci-dessous.

---

⚠️ **ATTENTION** ⚠️ : si tu as besoin de cette aide pour avancer, c'est que le concept d'objet et l'écriture des classes n'est pas du tout clair pour toi (pour le moment). Il est INTERDIT de copier-coller cette méthode dans ton code : je veux que tu y jettes un œil puis que tu essayes, sans la recopier, de l'écrire à nouveau dans ton code. Tu dois comprendre chaque ligne. Ensuite, lance `app.rb` avec son `binding.pry` et fais plein de tests en créant 2 ou 3 objets `Player` différents, fais des `show_state` et des `gets_damage` dessus pour bien visualiser comment tout ça marche.

Il n'y aura pas d'autre aide donc c'est important que tu saisisses MAINTENANT la logique. Quitte à te la faire expliquer par un co-moussaillon !

---

-début de l'aide-

def gets_damage(damage_received) #damage_received est un entier qu'on met en entrée de la méthode

# On soustrait l'entier en entrée au niveau de vie de l'objet sur lequel la méthode est appliquée :

@life_points = @life_points - damage_received

# Si le niveau de vie de l'objet est inférieur ou égal à zéro, le joueur est tué et on affiche un message.

```
if @life_points <= 0
  puts "le joueur #{@name} a été tué !" # l'écriture #{} permet d'insérer une variable dans un string
end
```

-fin de l'aide-

#### d) `Player` : attaquer avec `attacks`

On est à présent capable de faire baisser les points de vie d'un joueur avec `gets_damage`. Il est temps de coder une méthode `attacks` qui permette de faire qu'un joueur attaque un autre. Ainsi, si le joueur `player1` attaque le joueur `player2`, on aurait juste à taper `player1.attacks(player2)`.

Code cette méthode en respectant ces contraintes :

- La méthode prend donc en entrée un objet `Player` qui est le joueur subissant l'attaque ;
- La méthode commence par annoncer "le joueur [nom de player1] attaque le joueur [nom de player2]" avec un `puts` ;

- Ensuite on doit calculer les dommages que `player1` va faire subir à `player2`. Pour des raisons que tu comprendras plus tard, on va créer une méthode à part s'appelant `compute_damage` qui va faire ce calcul. Et dans la pure tradition des jeux de rôle, les dommages seront aléatoires car égaux au résultat d'un lancé de dé (= un chiffre au hasard entre 1 et 6). Voici le code de notre méthode `compute_damage` que tu vas mettre juste à la suite de `attacks` :

  ```
  def compute_damage
  return rand(1..6)
  end
  ```

- Maintenant, dans `attacks`, fais appel à `compute_damage` et stocke le résultat dans une variable.

- Fais subir les dégâts à l'autre `Player` en utilisant ces dommages et la méthode `gets_damage`.

- À présent `puts` une phrase qui explique ce qui vient de se passer : "il lui inflige XXXX points de dommages"

Voici le fonctionnement qu'on doit obtenir en exécutant `app.rb` et en utilisant PRY :

```
[1] pry(main)> player1 = Player.new("Josiane")
=> #<Player:0x000055ebac183358 @life_points=10, @name="Josiane">
[2] pry(main)> player2 = Player.new("José")
=> #<Player:0x000055ebac11cb80 @life_points=10, @name="José">
[3] pry(main)> player1.attacks(player2)
Josiane attaque José
il lui inflige 5 points de dommages
=> nil
[4] pry(main)> player2.show_state
```

```
José a 5 points de vie
=> nil
[5] pry(main)> player1.show_state
Josiane a 10 points de vie
=> nil
```

### e) Orchestrons un combat !

C'est bon, on a tous les ingrédients pour que nos joueurs se trucident bien comme il faut ! Les joueurs peuvent attaquer, subir des dégâts et on peut donner des infos sur le terminal via plein de `puts`. Il ne nous reste qu'à mettre en place notre premier combat.

On va tout coder dans `app.rb` qui va orchestrer le combat et faire appel à la classe `Player`. Petite exception à la règle : pas besoin de créer une classe `App` dans ce fichier.

Je vais à nouveau te guider pas à pas. N'oublie pas de vérifier que chaque étape fonctionne bien en exécutant `app.rb` avec un petit `binding.pry` à la fin pour confirmer que tout marche avant de passer à l'étape d'après.

Allez c'est parti :

1. **À ma droite "Josiane"** : crée un `Player` répondant à ce doux prénom et stocké dans la variable `player1`.
2. **À ma gauche "José"** : crée un autre `Player` répondant à ce joli prénom et stocké dans la variable `player2`.
3. **Présentons les deux combattants** : affiche dans le terminal l'état de chaque combattant grâce à des `puts` et des `show_state`. Juste avant, affiche un petit `puts "Voici l'état de chaque joueur :"`.
4. **Fight !** Indique que le combat commence avec un `puts "Passons à la phase d'attaque :"`.
5. **Josiane aura l'honneur d'attaquer la première** : fais attaquer `player2` par `player1` avec la méthode `attacks`.
6. **José ne va pas se laisser faire** : fais l'attaque inverse.

Si tu exécutes le code en l'état, tu devrais avoir un premier round d'attaque entre nos deux gladiateurs. C'est déjà un bon début ! Mais José et Josiane doivent à présent combattre jusqu'à la mort 👇. Ils doivent donc s'attaquer jusqu'à ce que l'un des deux n'ait plus de point de vie. Es-tu déjà en mesure de voir comment faire ?

Allez, sur cette première étape je t'aide : il faut faire une boucle `while`. En effet, on ne sait pas combien de tours le combat va durer vu que les dommages sont aléatoires (il faut entre 2 et 10 coups pour venir à bout des 10 points de vie).

Tu vas donc créer une boucle qui commence au point 3 de la liste ci-dessus (la présentation de l'état de chaque combattant doit avoir lieu au début de la boucle) et finie au point 6. Cette boucle doit tourner jusqu'à ce qu'un des `Player` voit son niveau de vie être inférieur ou égal à zéro. Es-tu en mesure d'écrire cette condition d'arrêt ?

Aide si tu n'y arrives pas : while player1.life_points > 0 && player2.life_points > 0

Et voilà tu dois maintenant pouvoir faire tourner plusieurs combats sans souci et faire des paris sur qui, de José ou Josiane, en sortira les pieds devants. Hum… En fait si tu fais tourner plusieurs combats, tu devrais identifier un petit souci.

### f) Corrigeons ce petit bug

Et oui, si jamais José (`player2`) venait à être tué en premier, tu vas vite te rendre compte qu'il peut malgré tout porter une dernière attaque sur Josiane (et potentiellement la tuer aussi). Pas terrible : ça serait considéré comme un bug par un utilisateur extérieur ! On doit gérer ce cas.

Juste après l'attaque de Josiane (`player1`) sur José (`player2`), rajoute un petit `if` qui teste si `player2` est mort (ses points de vie sont négatifs ou nuls). Si c'est le cas, il nous faut sortir immédiatement de la boucle `while` : c'est faisable grâce à la ligne `break`.

### Résultat attendu sur le terminal

Voici un exemple de ce que le terminal doit afficher si tu lances un combat en exécutant `app.rb` :

```
Voici l'état de nos joueurs :
Josiane a 10 points de vie
José a 10 points de vie

Passons à la phase d'attaque :
Josiane attaque José
il lui inflige 4 points de dommages
José attaque Josiane
il lui inflige 3 points de dommages

Voici l'état de nos joueurs :
Josiane a 7 points de vie
José a 6 points de vie

Passons à la phase d'attaque :
Josiane attaque José
il lui inflige 1 points de dommages
José attaque Josiane
il lui inflige 4 points de dommages

Voici l'état de nos joueurs :
Josiane a 3 points de vie
José a 5 points de vie

Passons à la phase d'attaque :
Josiane attaque José
il lui inflige 1 points de dommages
José attaque Josiane
il lui inflige 6 points de dommages
le joueur Josiane a été tué !
```

Comme tu peux le voir, j'ai inséré quelques sauts de ligne en plus pour rendre le tout plus lisible. N'hésite pas à mettre en page un peu mieux si tu te sens de rajouter des `puts` en plus, des lignes de "----------" ou autres.

## 2.3. Version 2.0 : créer un nouveau type de joueur

Nous allons maintenant créer un nouveau type de joueur qui va disposer de plus de compétences et de plus de choix : il sera joué par un humain. Je vais un peu moins t'accompagner dans cette version-là afin de te laisser te creuser la tête et t'obliger à chercher les solutions sur internet ou parmi la communauté.

### a) Poser les bases de la nouvelle classe

Commence par compléter ton fichier `player.rb` en rajoutant, à la suite de la classe `Player`, une nouvelle classe que nous appellerons `HumanPlayer`. Les bonnes pratiques en Ruby voudraient qu'on découpe cela en 2 fichiers différents mais ici, on va faire simple et efficace car ce programme est relativement petit.

Cette classe doit hériter de `Player` car nous voulons qu'un objet `HumanPlayer` ait la même base : les attributs `name` et `life_points` ainsi que toutes les méthodes que nous avons codées.

### b) `HumanPlayer` : l'attribut `@weapon_level` et modification de `initialize` et `show_state`

Il y aura 2 choses qui vont différencier un `HumanPlayer` d'un `Player`. Tout d'abord, on va donner un petit avantage à notre joueur humain vis-à-vis des bots : il disposera de 100 points de vie. Ensuite, il aura la possibilité d'améliorer l'arme dont il dispose pour augmenter les dégâts qu'il inflige à ses adversaires.

Commence par définir en `attr_accessor` la variable `weapon_level` qui définira, sous forme d'integer, le niveau de l'arme du `HumanPlayer`.

Maintenant tu vas définir une nouvelle méthode `initialize` qui prend toujours uniquement le `name` en entrée mais, à la différence de celle des `Player` :

- Elle donne une valeur de 100 à `@life_points`.
- Elle fixe `@weapon_level = 1`.

Et vu qu'on modifie la nature de cet objet, autant modifier la façon de présenter le joueur. Tu vas changer la méthode `show_state` afin qu'elle affiche une phrase du genre "XXX a YYY points de vie et une arme de niveau ZZZ".

#### c) `HumanPlayer` : augmenter les dommages causés

C'est cool que les `HumanPlayer` aient une arme avec un `@weapon_level`, mais il faut que ça serve en combat ! On va faire de `@weapon_level` un multiplicateur des dégâts. Tu vas créer, dans `HumanPlayer`, une nouvelle méthode `compute_damage` pour refléter cela :

```
def compute_damage
  rand(1..6) * @weapon_level
end
```

Du coup tu comprends mieux pourquoi, dans la classe `Player`, je t'avais fait créer une méthode `compute_damage` toute seule. C'était pour anticiper le fait que la ligne `rand(1..6)` allait devoir prendre en compte l'existence d'un `@weapon_level` dans la classe `HumanPlayer`. Grâce à ça, on a juste à modifier la méthode très courte `compute_damage` et non pas la méthode `attacks` entière.

Si tu suis bien, une fois que tu as fait ce travail, les joueurs de type `Player` pourront infliger des dégâts compris entre 1 et 6 alors que les joueurs de type `HumanPlayer` infligeront des dégâts compris entre [@weapon_level] et [6 x @weapon_level].

#### d) `HumanPlayer` : chercher une nouvelle arme

Une nouvelle fonctionnalité spécifique aux `HumanPlayer` sera la possibilité pour lui d'aller chercher une nouvelle arme, plus puissante. Pour cela, tu vas coder, dans la classe `HumanPlayer`, une méthode `search_weapon` qui va faire les choses suivantes :

- Elle va commencer par lancer un "dé" dont le résultat sera compris entre 1 et 6 (tu sais faire ça maintenant non?).
- Ce lancé de dé sera égal au niveau de la nouvelle arme trouvée. Annonce le résultat de la recherche à l'utilisateur en affichant un message du genre "Tu as trouvé une arme de niveau XXX".
- Maintenant, cherche à savoir si ça vaut le coup pour le joueur `Human Player` de la garder… Utilise un `if` pour comparer le niveau de cette nouvelle arme avec celle qu'il possède déjà (`@weapon_level`).
- Si l'arme trouvée est d'un niveau strictement supérieur, il la garde. Son `@weapon_level` prend alors la valeur de la nouvelle arme et tu affiches un message du genre "Youhou ! elle est meilleure que ton arme actuelle : tu la prends."
- Si l'arme trouvée est égale ou moins bien que son arme actuelle, tu ne changes rien et ne fais qu'afficher un petit "M@*#$... elle n'est pas mieux que ton arme actuelle..."

#### e) `HumanPlayer` : chercher un pack de points de vie

Une autre fonctionnalité qu'auront les `HumanPlayer` : ils pourront partir à la recherche d'un pack de points de vie afin de faire remonter leur niveau de vie. De façon assez similaire à la méthode `search_weapon`, tu vas coder une méthode `search_health_pack` qui va se comporter comme suit :

- Elle commence également par lancer un "dé" dont le résultat sera compris entre 1 et 6. En fonction du résultat, voilà ce qu'elle devra faire :
- Si le résultat est égal à 1, le joueur n'a rien trouvé et on retourne simplement le string "Tu n'as rien trouvé... ".
- Si le résultat est compris entre 2 (inclus) et 5 (inclus), le joueur a trouvé un pack de 50 points de vie. On va donc augmenter sa vie de 50 points mais sans qu'elle puisse dépasser 100 points. Puis on va retourner le string "Bravo, tu as trouvé un pack de +50 points de vie !".
- Si le résultat est égal à 6, le joueur a trouvé un pack de 80 points de vie. On va donc augmenter sa vie de 80 points mais sans qu'elle puisse dépasser 100 points. Puis on va retourner le string "Waow, tu as trouvé un pack de +80 points de vie !".

#### f) Combat interactif sur `app_2.rb` : toi contre José et Josiane

OK, tous les ingrédients sont prêts: il ne nous reste plus qu'à mettre le tout en musique pour enfin jouer ton premier interactif ! Cela va se faire au tour-par-tour et à chaque fois tu auras le choix entre plusieurs actions afin de te défaire de tes 2 adversaires.

Le fichier qui va orchestrer tout cela va s'appeler `app_2.rb` afin que tu gardes intacte ta version 1.0 sur `app.rb`. Commence donc par créer ce fichier (toujours à la racine de ton dossier) et fais-le commencer par les mêmes 4 lignes de code que ton `app.rb` de base :

```
require 'bundler'
Bundler.require

require_relative 'lib/game'
require_relative 'lib/player'
```

Maintenant, voici la liste de ce que tu dois coder dans `app_2.rb`. Fais-le au fur et à mesure, et ne passe à l'étape suivante que si ton code marche :

- **Accueil** : Commence par afficher dans le terminal, au lancement de `app_2.rb`, un petit message de démarrage du jeu. Laisse libre cours à ton imagination mais voici un exemple basique :

  ```
  ------------------------------------------------
  |Bienvenue sur 'ILS VEULENT TOUS MA POO' !      |
  |Le but du jeu est d'être le dernier survivant !|
  ------------------------------------------------
  ```

- **Initialisation du joueur**: ensuite, le jeu va demander à l'utilisateur son prénom et créer un `HumanPlayer` portant ce prénom.

- **Initialisation des ennemis** : le jeu va maintenant créer nos deux combattants préférés, "Josiane" et "José".

  Comme nous savons qu'à terme (version 3.0) il y aura plus de 2 ennemis, on va mettre en place une astuce pour manipuler facilement un groupe d'ennemis : le jeu va créer un array `enemies` qui va contenir les deux objets `Player` que sont José et Josiane. Tu verras plus tard l'usage qu'on va en faire.

- **Le combat** : tout comme dans la version 1.0, on peut maintenant lancer le combat ! Tu vas ouvrir une boucle `while` qui ne doit s'arrêter que si le joueur de l'utilisateur (`HumanPlayer`) meurt ou si les 2 joueurs "bots" (`Player`) meurent. Cette condition d'arrêt n'est pas triviale à écrire mais je te propose d'essayer ! Sinon la réponse est disponible plus bas. Laisse la boucle `while` vide pour le moment, on la codera juste après.

- **Fin du jeu** : maintenant, si on sort de cette boucle `while`, c'est que la partie est terminée. Donc juste en dessous du `end` de la boucle, on va préparer un petit message de fin. Le jeu doit afficher "La partie est finie" et ensuite soit afficher "BRAVO ! TU AS GAGNE !" si le joueur humain est toujours en vie, ou "Loser ! Tu as perdu !" s'il est mort.

Très bien, on a l'ossature globale du jeu version 2.0 mais il reste à coder le combat ! Voici ce que tu vas mettre dans la boucle `while` :

- Tout d'abord, on te donne la condition d'arrêt de la boucle en partant de l'hypothèse que tu as stocké le `HumanPlayer` dans la variable `user` et les deux `Player` dans les variables `player1` et `player2`.

  ```
  while user.life_points >0 && (player1.life_points > 0 || player2.life_points >0)
  ...
  end
  ```

- La première chose qu'on va faire à chaque tour de combat, c'est afficher l'état du `HumanPlayer` : rajoute cela.

- Ensuite, on va proposer un menu qui doit ressembler à cela :

  ```
  Quelle action veux-tu effectuer ?
  a - chercher une meilleure arme
  s - chercher à se soigner
  attaquer un joueur en vue :
  ```

```
0 - Josiane a 10 points de vie
1 - José a 10 points de vie
```

- Évidemment la partie "Josiane a 10 points de vie" et "José a 10 points de vie" devra se mettre à jour quand ils vont recevoir des dégâts. Tu dois donc utiliser la méthode `show_state` dans cette partie du menu pour afficher l'état réel de chaque `Player` que l'utilisateur combat.

- Une fois cela fait, on laisse l'utilisateur effectuer une saisie. Et en fonction de la saisie, on va :

- si l'utilisateur tape "a", exécuter sur son `HumanPlayer` la méthode qui le fait partir à la recherche d'une arme ;

- si l'utilisateur tape "s", exécuter sur son `HumanPlayer` la méthode qui le fait partir à la recherche d'un pack de soin ;

- si l'utilisateur tape "0", faire attaquer Josiane par son `Human Player` ;

- si l'utilisateur tape "1", faire attaquer José par son `Human Player` ;

- C'est maintenant au tour des ennemis de riposter ! Tu peux l'indiquer en affichant en console un petit `puts "Les autres joueurs t'attaquent !"`

- On va faire que les 2 `Player` attaquent le `HumanPlayer`. Mais au lieu d'écrire 2 lignes quasiment identiques en mode `player1.attacks(user)` et `player2.attacks(user)`, je veux que tu utilises l'array `enemies` contenant les 2 objets `Player`. L'idée est de faire une boucle `each` sur cet array pour ensuite exécuter la méthode `attacks` sur chaque objet. Pourquoi ? Tout simplement car on anticipe là le fait qu'il y aura bientôt 10 ou 20 ou 30 `Player` : on va pas se taper 10 ou 20 ou 30 lignes de `playerX.attacks(user)` !

- Ha oui, un petit dernier truc : il ne faut pas qu'un `Player` puisse attaquer s'il est mort… Donc rajoute un petit `if` dans ta boucle `each`.

Super ! Tu es arrivé au bout de la version 2.0 de ton Fornite-like ! Lance plusieurs combats, fais plein de tests et compare les résultats avec tes fellow moussaillons. N'hésite pas à mettre des petits `gets.chomp` ici et là qui auront pour seul objectif de faire une petite pause dans l'affichage du texte du jeu sur le terminal. Ça aidera à la lecture et à suivre ce qu'il se passe.

## 2.4. Version 3.0 : un jeu mieux organisé et plus d'ennemis

Bravo, tu as fini la version 2.0 ! Si tout est fait proprement jusque là, tu valides sans souci le projet. Maintenant, on va augmenter un peu le niveau car tu as besoin de challenge. Fini le pas à pas : on va te donner des instructions plus générales, charge à toi de t'en sortir seul !

Commence par créer un fichier `app_3.rb` pour garder l'historique de tes versions 1.0 et 2.0. Ce fichier doit commencer avec les mêmes 4 lignes de `require` que les 2 autres.

### a) Première étape : rapatrier plein de choses de `app_2.rb` vers une nouvelle classe `Game`

Notre fichier `app_2.rb` gère beaucoup trop de chose et il est trop long pour que ce soit acceptable : on doit ranger son contenu dans des méthodes et des classes dédiées. On va donc créer une classe `Game` dans le fichier `game.rb` qui aura pour rôle de stocker les informations du jeu et effectuer chaque étape.

Voici ce que tu dois faire dans la classe `Game` (80 % du travail consiste à rapatrier du code depuis `app_2.rb`) :

- Crée la classe `Game` qui aura 2 `attr_accessor` : un `@human_player` de type `HumanPlayer` et un array `@enemies` qui contiendra des `Player`.
- Un objet `Game` s'initialise ainsi : `my_game = Game.new("Wolverine")`. Il crée automatiquement 4 `Player` qu'il met dans `@enemies` et un `HumanPlayer` portant (dans cet exemple) le nom "Wolverine".
- Écris une méthode `kill_player` qui prend un objet `Player` en entrée et le supprime de `@enemies`. Cette méthode permet d'éliminer un adversaire tué.
- Écris une méthode `is_still_ongoing?` qui retourne true si le jeu est toujours en cours et false sinon. Le jeu continue tant que le `@human_player` a encore des points de vie et qu'il reste des `Player` à combattre dans

l'array `@enemies`.
- Écris une méthode `show_players` qui va afficher 1) l'état du joueur humain et 2) le nombre de joueurs "bots" restant
- Écris une méthode `menu` qui va afficher le menu de choix (juste l'afficher, pas plus). On a les mêmes choix que dans la version 2.0 à la seule différence qu'il y a plus de 2 ennemis à combattre et que si un ennemi est mort, on ne doit plus proposer de l'attaquer.
- Écris une méthode `menu_choice` qui prend en entrée un string. Cette méthode va permettre de faire réagir le jeu en fonction du choix, dans le menu, de l'utilisateur. Par exemple, si l'entrée est "a", le `@human_player` doit aller chercher une arme. Si l'entrée est "0", on le fait attaquer l'ennemi présenté au choix "0", etc. Pense à faire appel, dans cette méthode, à la méthode `kill_player` si jamais un `Player` est tué par le joueur humain !
- Écris une méthode `enemies_attack` qui va faire riposter tous les ennemis vivants. Ils vont attaquer à tour de rôle le joueur humain.
- Écris une méthode `end` qui va effectuer l'affichage de fin de jeu. Tu sais, la partie "le jeu est fini" puis "Bravo..." ou "Loser..."

### b) `app_3.rb` en chef d'orchestre

Maintenant tu vas compléter `app_3.rb` pour qu'il puisse coordonner le jeu.

Fais commencer le jeu par l'habituel message de bienvenue, demande son nom à l'utilisateur et utilise-le pour initialiser un objet `Game` que tu vas stocker dans une variable `my_game` et rappeler tout du long de ton code.

Inspire-toi du contenu de `app_2.rb` mais cette fois-ci, tu ne dois faire que des appels de méthode sur ton objet `my_game`. Il contient toute l'information du jeu en cours : le personnage de ton utilisateur et ses ennemis. Par contre tu peux mettre les `gets.chomp` directement dans le code de `app_3.rb`

### c) Quelques fonctionnalités en plus

On aimerait bien que notre Fornite-like permette de combattre 10 ou 20 ennemis sur une même partie... Mais s'ils débarquent tous dès le début et s'acharnent sur notre pauvre `HumanPlayer`, même avec ses habilités spéciales il va pas durer longtemps.

On va mettre en place un système où les ennemies vont débarquer au compte-goutte, un peu comme dans les jeux où on tombe sur eux au hasard de tes déplacements sur la carte. Pour ça, on va suivre à la fois le nombre d'ennemis toujours présents dans le jeu ( = ennemis restant à éliminer) et le nombre d'ennemis qui sont "en vue" (= ennemis qu'on peut attaquer et qui peuvent nous attaquer en retour). Voilà comment on va faire dans la classe `Game` :

- Rajoute deux attributs : `@players_left` qui est un integer qu'on initialize à 10 et `@enemies_in_sight` qui vient remplacer `@enemies` et qui est un array vide au départ. Tu l'as compris car j'ai bien nommé mes variables (prends en de la graine) : `@players_left` représente le nombre de joueur restant dans le jeu (= nombre restant à éliminer pour gagner) et `@enemies_in_sight` est un array de `Player` qui sont ceux en vue (= qu'on peut attaquer et qui vont nosu attaquer en retour).
- Modifie la méthode `is_still_ongoing?` pour que le jeu continue tant que notre `HumanPlayer` est toujours en vie et qu'il n'est pas le dernier vivant.

- Crée une méthode `new_players_in_sight` qui va avoir pour rôle de rajouter des ennemis en vue. Voici les règles de fonctionnement de cette méthode :

- Si tous les joueurs du jeu sont déjà "en vue", on ne doit pas en rajouter. Dans ce cas, cela signifie que le nombre d'objets `Player` dans `@enemies_in_sight` est égal à l'integer `@players_left`. Affiche alors un message d'info du type "Tous les joueurs sont déjà en vue".

- La méthode va lancer un dé à 6 faces et va réagir en fonction de ce résultat aléatoire :

- Si le dé vaut 1, aucun nouveau joueur adverse n'arrive (afficher un message informant l'utilisateur).

- Si le dé vaut entre 2 et 4 inclus, un nouvel adversaire arrive en vue. Il faut alors créer un `Player` avec un nom aléatoire du genre "joueur_1234" ou "joueur_6938" (ou ce que tu veux) et injecter ce `Player` dans le

array `@enemies_in_sight` . Affiche un message informant l'utilisateur de ce qui se passe.

- Si le dé vaut 5 ou 6, cette fois c'est 2 nouveaux adversaires qui arrivent en vue. De même qu'au-dessus, il faut les créer et les rajouter au jeu. Rajoute toujours un message informant l'utilisateur.

- Et maintenant, il faut que cette méthode `new_players_in_sight` soit appelée dans ton `app_3.rb` juste avant l'affichage du menu à l'utilisateur. Cela permet d'ajouter, petit à petit, des adversaires en vue !

Voilà, une fois que tu auras fait ça, tu pourras essayer de sortir vivant d'un combat contre 10, 20 voire 100 adversaires ! N'hésite pas à pimper l'affichage pour l'utilisateur et à joueur sur les paramètres (la vie de chaque adversaire, ta vie, la taille des packs de vie qu'on peut trouver, etc.) pour trouver les réglages qui sont les plus fun !

# 3. Rendu attendu

Un repo GitHub contenant l'ensemble de ton programme qui a l'architecture dossier décrite en 2.1

Le repo doit contenir chaque version de ton jeu en fonction de ce que tu auras réussi à faire : `app.rb`, `app_2.rb` et `app_3.rb`. En effet, on va demander à tes correcteurs de tester chaque version.

Ce projet va durer trois jours et devra être rendu mardi prochain.
Retour à la page précédente
Proposer une amélioration du projet