TP 6 : Le combat des pokemons

Dans ce TP nous allons finalement réaliser le combat entre les pokemons. Dans une première étape, nous allons utiliser l'héritage pour définir quelques types d'attaques. Les pokemons pourront désormais ajouter et utiliser des attaques.

Liens utiles : https://pokemondb.net/pokedex/all .

https://pokemondb.net/move

Contexte

L'idée de ce TP est de vous montrer un deuxième type d'héritage, notamment comment faire des classes hériter d'une classe abstraite. Nous allons commencer sur une première superclasse abstraite `Attaque`. Puis nous allons créer deux sousclasses concrètes d'`Attaque`, notamment `AttaquePhysique` et `AttaqueSpeciale`. En fonction de si une attaque est plutôt une attaque physique ou une attaque spéciale, nous allons faire des attaques individuelles (modélisées par des sousclasses) hériter soit d'`AttaquePhysique`, soit d'`AttaqueSpeciale`.

Même si nous devons simplifier beaucoup la dynamique des attaques, nous allons avoir besoin de prendre en compte plus de caractéristiques des pokémons de ce qu'on fait actuellement. On devra donc retravailler la classe `Pokemon`. Nous allons utiliser les nouvelles classes pour permettre un combat entre deux joueurs qui partagent le même écran. Notamment, nous n'allons pas pouvoir implémenter une intelligence artificielle qui fera un choix au nom du pokemon adversaire. Finalement, le combat ne sera qu'entre deux pokemons -- même si un joueur aura plus de pokemons à sa disposition, ceux-ci ne pourront pas reprendre le combat d'un pokemon qui s'est évanoui.

Préambule

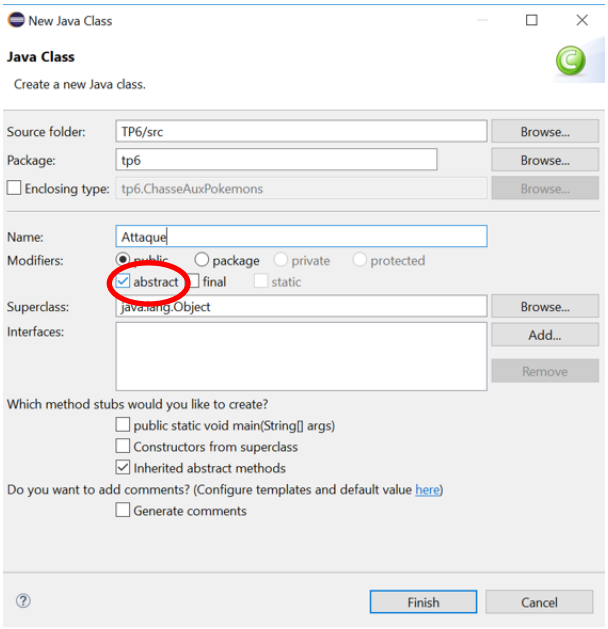
Commencez en créant un nouveau projet Java nommé TP6. Créez dedans un package tp6 où vous pouvez copier tous les fichiers du TP5. Assurez-vous que le code compile et s'exécute encore normalement.

Exercice 1

Dans cet exercice nous allons créer la classe abstraite `Attaque`, qui représente la forme la plus générale d'une attaque. Pour bien comprendre la classe `Attaque` je vous recommande de regarder en même temps la documentation en ligne là-dessus, sur par exemple : https://pokemondb.net/move

1. Pour créer une classe abstraite en Eclipse on peut directement cocher cette option lors de la création de la classe, comme indiqué dans la figure suivante. Ceci n'est pas la seule option de

créer une classe abstraite : bien-sûr qu'on peut simplement ajouter le mot abstract à la déclaration de la classe.



Créez une classe abstraite Attaque dans le package tp6 du projet tp6.

2. Une attaque aura les attributs prochains :
   - Un attribut `nom` de type `String`;
   - Un attribut `compatibilites` de type `String[]` qui stocke les types (“PLANTE“, “EAU“, etc.) des pokemons qui pourront utiliser cette attaque ;
   - Un attribut `puissance` de type `int` qui indique la puissance d'une attaque (Power dans le Pokedex), notamment combien de points HP le pokemon perdra suite à cette attaque ;
   - Un attribut `precision` de type `int`, qui indique la précision (Acc dans le Pokedex) d'une attaque, notamment la fréquence avec laquelle l'attaque marche ;
   - Un attribut `nombreRepetitions` de type `int`, qui indique le nombre de répétitions qu'on peut utiliser cette attaque dans une seule bataille (PP dans le Pokedex) ;
   - Un attribut `repetitionsRestantes` de type `int`, qui stocke le nombre de répétitions encore disponibles pour cette attaque.

   Déclarez ces attributs pour la classe abstraite Attaque.

3. On définit deux constructeurs pour la classe `Attaque`. Le premier constructeur a la signature `Attaque(String nom, String[] compatibilites, int puissance, int precision, int nombreRepetitions)`. Ce constructeur instancie les cinq premiers attributs de la classe Attaque à la valeur correspondante mise en paramètre ; de plus le nombre de répétitions restantes est mis au nombre total de répétitions. Ecrivez ce constructeur.

4. Le deuxième constructeur de la classe `Attaque` aura la signature `Attaque(String nom, int puissance, int precision, int nombreRepetitions)`. Ce constructeur fonctionne comme le premier constructeur, sauf que le tableau de compatibilités sera mis par défaut au

tableau de tous les types de pokemons (indice : dans le code de la correction, un attribut statique qui stocke cette valeur est déjà défini pour la classe `Nourriture`).

5. Ecrivez des getters concrètes pour chaque attribut de cette classe.

6. Ecrivez une méthode concrète `void resetNombreRepetitions()` qui fait en sorte que le nombre restant de répétitions est remis au nombre de répétitions maximale.

7. Ecrivez une méthode concrète `void baisserNombreRepetitions()` qui baisse par 1 le nombre de répétitions restantes de l'attaque, toutefois sans le diminuer sous 0.

8. Ecrivez une méthode concrète `String toString()` qui retourne le texte : `<nom d'attaque> : <puissance>, <precision>, <repetitionsRestantes>/ <nombreRepetitions>, <compatibilites dans le format {type1, type2, …}>`

9. La classe `Attaque` aura également deux méthodes abstraites. La première méthode abstraite aura la signature `void utiliserAttaque(Pokemon attaquant. Pokemon victime)`. Comme nous sommes dans la classe Attaque, cette méthode sera utilisée plus tard pour faire en sort que le pokemon `attaquant` attaque le pokemon `victime` avec l'attaque actuelle (`this`).

   Ecrivez cette méthode.

10. La deuxième méthode abstraite que vous allez maintenant écrire dans la classe `Attaque` aura la signature `boolean isCompatible(Pokemon pokemon)`.

Exercice 2

Dans ce deuxième exercice il va falloir adapter la classe `Pokemon` pour que les pokemons puissent stocker et utiliser des attaques.

1. Comme première étape de cet exercice préparez votre classe `ChasseAuxPokemons` : commentez tout sauf la déclaration des divers objets (des pokemons, des joueurs, de la nourriture, etc.).

2. Dans la classe `Pokemon` nous allons introduire les prochains nouveaux attributs qui seront tous de type `int` :
   - Un attribut `attaque` qui définit la capacité d'un pokemon d'attaquer un autre ;
   - Un attribut `defense` qui définit la capacité d'un pokemon de se prémunir contre une attaque ;
   - Un attribut `attaqueSpeciale` qui définit la capacité d'un pokemon d'utiliser (effectivement) une attaque spéciale ;

- Un attribut `defenseSpeciale` qui définit la capacité d'un pokemon de se défendre envers d'une attaque spéciale.
- Un attribut hp qui représente le nombre de points de vie que le pokemon peut perdre en combat avant de s'évanouir.

Ajoutez ces attributs aux attributs de la classe `Pokemon`.

3. Ajoutez un dernier attribut `attaques` de type `Attaque[]` à la classe `Pokemon`. Cet attribut aura la taille 4.

4. Ecrivez des getters pour chaque nouvel attribut de la classe Pokemon. De plus, écrivez un getter pour l'attribut niveau, si un tel constructeur n'existe pas encore.

5. Modifiez les constructeurs de la classe `Pokemon`. Le premier constructeur, qui avait la signature `Pokemon(String nom, String type, int niveau, boolean diurne, String nomDonne, Joueur monJoueur)` aura désormais la signature `Pokemon(String nom, String type, int niveau, boolean diurne, String nomDonne, Joueur monJoueur, int attaque, int defense, int attaqueSpeciale, int defenseSpeciale, Attaque[] attaques)`. En plus de sont fonctionnement passé, ce constructeur initialise les attributs `attaque, defense attaqueSpeciale, defenseSpeciale, attaques` aux attributs donnés en entrée. Il initialise la valeur de l'attribut hp à 30 par défaut.

   Attention : ceci causera des erreurs de compilation dans votre classe `ChasseAuxPokemons`. Ce n'est pas grave, nous allons nous en occuper plus tard.

6. Modifiez le deuxième constructeur de la classe Pokemon tel que sa signature devienne `Pokemon(String nom, String type, int niveau, boolean diurne, Joueur monJoueur, int attaque, int defense, int attaqueSpeciale, int defenseSpeciale, Attaque[] attaques)`. Dans ce constructeur appelez le premier constructeur, tel que les paramètres `nomDonne` et `monJoueur` soient mis à `null`.

7. Mettez à jour la méthode toString() pour également inclure les nouveaux attributs ajoutés. Les attaques nulls ne sont pas retournées.

8. Ajoutez une méthode privée `int trouverAttaque(Attaque attaque)` qui cherche l'attaque mis en paramètre dans le tableau attaques. Si on trouve l'attaque, la méthode retourne l'index de l'attaque dans le tableau; sinon on retourne par défaut la valeur -1.

9. Ajoutez une méthode `void ajouterAttaque(Attaque attaque)` à cette classe avec le fonctionnement suivant : si l'attaque est compatible avec le pokemon et si le pokemon a encore de la place pour cette attaque on ajoute l'attaque sur une position disponible. Utilisez votre méthode `trouverAttaque` pour programmer la méthode `ajouterAttaque`.

10. Actuellement, notre premier constructeur instancie les attaques d'un pokemon au tableau mis en entrée, qu'ils soient compatibles avec le pokemon ou non. Nous allons désormais utiliser la méthode `ajouterAttaque` pour ajouter un par un les éléments du paramètre attaques à l'attribut attaques, car cette méthode vérifie la compatibilité aussi. Qu'est-ce qui se passe si la taille du paramètre est supérieure à la taille de l'attribut ?

11. Nous voulons éventuellement pouvoir également remplacer une attaque par une autre. Nous ferons ceci en ajoutant une deuxième méthode ajouterAttaque, cette fois-ci avec la signature `void ajouterAttaque(Attaque attaque, int i)` . Si i est un index valide dans le tableau attaques, la valeur qui existe déjà sur cette position-là est remplacée par l'attaque en entrée (si celui-ci est compatible avec le pokemon).

12. Après chaque bataille, les attaques d'un pokemon se rechargeront. Notamment le nombre de répétitions de chaque attaque sera remis au nombre maximal de répétitions possibles. Dans la classe `Pokemon` nous allons écrire une méthode `void rechargerAttaques()`. Quelle méthode de la classe Attaque peut-on utiliser ?

    Attention : pour s'assurer de ne pas avoir des erreurs de type `NullPointerException` il faut être sûrs de ne jamais appeler une méthode pour un objet `null` !

13. Si une attaque est efficace contre le pokemon, alors il sera blessé -- notamment il perdra des points de vie (HP). Lorsqu'un pokemon perd tous ses HP, alors il s'évanouit. Ecrivez premièrement une méthode `void blessure(int dommage)` qui fait baisser le nombre de HP du pokemon par la valeur de `dommage` (toutefois sans baisser sous 0).

14. Ecrivez une méthode `boolean sEstEvanoui()` qui retourne `true` si les points de vie (HP) du pokemon sont à 0.

15. Ecrivez une méthode `void utiliserAttaque(int index, Pokemon victime)`. Cette méthode permettra à un pokemon d'attaquer un autre pokemon (la victime) en utilisant l'attaque dont l'index est indiqué en paramètre. Dans la méthode on vérifie premièrement que le pokemon ne s'est pas évanoui et que l'index donné en entrée a une valeur valide (entre 0 et la taille du tableau attaques). Finalement, si l'attaque indiquée est non-`null`, alors nous appelons la méthode `utiliserAttaque` de la classe `Attaque` pour l'attaque indiqué, avec le pokemon actuel (`this`) en tant qu'attaquant et victime en tant que victime.

    On pourrait se demander pourquoi on n'indique pas l'attaque qu'on veut utiliser par un objet de type `Attaque`, sinon seulement par un index. Ceci est un choix de design : notre but sera ultérieurement de demander à l'utilisateur de saisir l'index de l'attaque qu'il veut utiliser. C'est alors qu'on va appeler cette méthode.

16. Finalement nous allons permettre une vue synthétique des attaques d'un pokemon, qui plus tard servira à faire vraiment la bataille des pokemons à partir de la classe `ChasseAuxPokemons`.

Ecrivez donc dans la classe `Pokemon` une méthode `void afficherEtatAttaques()`, qui affiche les attaques non-`null`s d'un pokemon, une attaque par ligne, dans le format :

`<index> : <nom attaque>, <répétitions restantes>/<répétitions totales>`

Exercice 3

Dans cet exercice nous allons créer deux classes concrètes, `AttaquePhysique` et `AttaqueSpeciale`, qui héritent toutes les deux de la classe `Attaque`. Les deux types d'attaques diffèrent principalement en termes de la compatibilité de l'attaque : les attaques physiques seront compatibles avec tous les types de pokemons, tandis que les attaques spéciales ne sont utilisables que par quelques types de pokemon.

Dans chacune de ces deux sousclasses, nous allons implémenter les méthodes suivantes : un constructeur et les deux méthodes abstraites de la classe Attaque, notamment isCompatible et utiliserAttaque.

1. Créez une classe `AttaquePhysique` qui hérite de la classe `Attaque`. Normalement Java vous indique déjà à ce point les méthodes manquantes (le constructeur) et les méthodes obligatoires à implémenter (les deux méthodes abstraites) dans la classe `AttaquePhysique`.
2. Le constructeur de la classe `AttaquePhysique` aura la signature `AttaquePhysique(String nom, int puissance, int precision, int nombreRepetitions)`. Il instancie les attributs de la classe aux valeurs en entrée, sauf l'attribut compatibilités, qui est instancié uniformément à l'ensemble de valeurs `String` représentant les types possibles de pokemons.

   Rappel : dans ma correction -- et peut-être dans votre code également, vous aurez une constante dans la classe `Nourriture` qui stocke tous les types de pokemon.

3. Pour la méthode `isCompatible(Pokemon pokemon)` nous allons premièrement vérifier si le pokemon en entrée est `null` -- si c'est le cas on retourne false. Dans toute autre cas, on retourne `true`.

4. Pour la méthode void utiliserAttaque(Pokemon attaquant, Pokemon victime), premièrement on vérifie si les deux paramètres ne sont pas nulls. Si au moins un des paramètres est null, rien ne se passera. Si, par contre les deux pokemons sont valides, les prochaines étapes de l'attaque seront observées :
   - Assez de répétitions de l'attaque ? Premièrement on vérifie s'il y a encore au moins une répétition de cette attaque -- sinon, rien ne se passe.
   - Succès/non-succès : un pokemon a un attribut attaque et un attribut defense. Le succès d'une attaque dépend d'un côté de ces scores, et dépend aussi de la chance. Notamment on génère un nombre aléatoire aleatoireAttaquant entre 0 et le niveau du pokemon attaquant. Puis, on génère un nombre aléatoire aleatoireVictime. On considère que l'attaque aura un succès si :

- la somme entre la valeur de l'attaque du pokemon attaquant et la valeur aleatoireAttaquant est strictement supérieure à la somme entre la valeur de la defense du pokemon victime et la valeur aleatoireDefense.
- Un entier choisi aléatoirement entre 0 et 100 sera inférieur à la précision de l'attaque.

Pour générer une valeur aléatoire entière dans l'interval 0-x on va utiliser des objets de type Random. Pour cela il faut premièrement importer le bon package : java.util.Random. Nous allons définir une variable random de type Random dans la méthode utiliserAttaque, en utilisant l'instruction :

```
Random random = new Random() ;
```

Puis, à chaque fois qu'on veut générer un nombre aléatoire, nous allons utiliser la méthode nextInt(int valeurMax) de la classe Random. Par exemple si on voulait générer un entier aléatoirement entre 0 et 5, on écrivait random.nextInt(6).

- Dommage : si l'attaque du pokemon a un succès, alors la victime sera blessée. Les dommages seront un entier aléatoire entre 0 et la puissance de l'attaque.

- Nombre de répétitions restantes : le nombre de répétitions restantes baissera par 1.

5. Prochainement nous allons créer la classe AttaqueSpeciale. Elle hérite également de la classe Attaque. Son constructeur aura la signature `AttaqueSpeciale(String nom, String[] compatibilites, int puissance, int precision, int nombreRepetitions)`. Il instancie les attributs de la classe aux valeurs en entrée.

6. La méthode `isCompatible(Pokemon pokemon)` rend `true` seulement si le type du pokemon est dans la liste de compatibilités. Attention : pour vérifier cela il faut premièrement vérifier que le pokemon en entrée n'est pas `null` (si le pokemon est `null`, on retourne false par défaut).

7. La méthode `utiliserAttaque` de la classe `AttaqueSpeciale` fonctionnera précisément de la même façon que celle de la classe `AttaquePhysique`, sauf qu'on utilise à chaque fois l'attribut `attaqueSpeciale` plutôt qu'attaque et `defenseSpeciale` plutôt que defense.

Exercice 4

Dans cet exercice le but sera de créer quelques attaques de chaque type (`AttaqueSpeciale` et `AttaquePhysique`). Pour chaque nouveau type d'attaque, nous allons seulement programmer le constructeur. Le nom, ainsi que toutes les autres informations concernant les compatibilités des attaques spéciales sont à trouver dans le Pokedex : https://pokemondb.net/move/all.

1. Créez les prochaines sousclasses de la classe `AttaquePhysique` : `AttaqueMorsure` (Bite dans le Pokedex), `AttaqueCroquer` (Crunch), `AttaqueFeinte` (Feint) et `AttaqueCoupDeTete`

(Headbutt). Pour chacune de ces classe écrivez seulement leur constructeur, qui aura la signature `<nom de la classe>()`,

2. Faites le même pour créer les classes suivantes d'`AttaqueSpeciale` : `AttaqueBulle` (Bubble), `AttaqueEnfer` (Inferno), `AttaquePistoleEau` (Water Gun) et `AttaqueTournadeFeuilles` (Leaf Tornado).

Exercice 5

Finalement dans cet exercice nous allons utiliser la structure de classes qu'on vient de bâtir. Nous allons nous concentrer sur la classe `ChasseAuxPokemons`.

1. Commentez tout le code de cette méthode sauf la partie sauf les diverses déclarations de variable.
2. Vous avez normalement des erreurs au niveau de la déclaration de chacun de vos pokemons. Pourquoi ?
3. Mettez à jour la déclaration de chacun de vos pokemons, en utilisant les informations sur l'attaque, défense, attaque spéciale, défense spéciale, HP, etc. du Pokedex (par exemple https://pokemondb.net/pokedex/piplup pour Piplup). Pour les attaques mettez un sous-ensemble des attaques créées.
4. Qu'est-ce qui se passe si vous donnez en entrée d'un de vos pokemons une attaque spéciale avec laquelle il n'est pas compatible ? Essayez de le faire et regardez le résultat en utilisant des `System.out.println` bien choisis.
5. [Difficile] Dans la méthode main, écrivez une bataille entre deux pokemons de votre choix. Au contraire de ce que va se passer dans une vraie bataille, vous allez contrôler tous les deux pokemons. Vous allez utiliser un objet de type Scanner pour vous donner la possibilité de choisir les attaques que vous voulez utiliser pour chaque pokemon. Vous pouvez choisir quel pokemon commence. La bataille continuera jusqu'à ce qu'un des pokemons s'évanouisse. Le pokemon de votre choix commencera à chaque ronde et l'autre pokemon répondra. A chaque ronde vous allez afficher les attaques du pokemon qui va attaquer et vous demanderez l'index de l'attaque qu'on veut utiliser. Vous allez lire la réponse des pokemons en utilisant la méthode `nextInt()` de la classe `Scanner`. Attention : il faut vérifier que la valeur de l'Index est bonne. Puis, vous faites la même chose pour l'autre pokemon. Une fois qu'un pokemon s'évanouit, la bataille est finie et on reset les attaques de chaque pokemon.