Développement orienté objet

TD7 : Abstraction & Polymorphisme

1er exercice : Moyens de paiement

On veut modéliser plusieurs moyens de paiement dans un magasin :

- Carte bancaire
- Espèces
- Chèque

Tous doivent fournir une méthode `payer(montant: number): void`.

1. Créer une **classe abstraite** `MoyenPaiement` avec :
    - une méthode abstraite `payer(montant: number): void`
2. Implémenter les trois classes filles :
    - `CarteBancaire` → affiche : *"Paiement de XX€ par carte accepté."*
    - `Cash` → affiche : *"Paiement de XX€ en espèces effectué."*
    - `Cheque` → affiche : *"Paiement de XX€ par chèque enregistré."*
3. Écrire une fonction : `function effectuerPaiement(p: MoyenPaiement, montant: number): void`, qui appellera la méthode abstraite de `MoyenPaiement`.
4. Tester des paiements avec des chèques, du cash, ou une carte bancaire.

2e exercice : Moyens de transports

On veut modéliser plusieurs moyens de transport, chacun avec une méthode `déplacer()`. Ex:

- `Voiture` : roule sur la route
- `Vélo` : avance à la force des jambes
- `Train` : circule sur les rails

Utilisez les concepts d'abstraction et polymorphisme pour pouvoir appeler `deplacer()` sans connaitre le type exact de transport.

3e exercice : Combat de Créatures 🐉⚔🐺

Un combat oppose plusieurs **créatures**, chacune ayant ses propres **caractéristiques** et **capacités** spéciales.

Règles du jeu

- Chaque joueur contrôle une **créature**.

- Le combat se déroule en **tour par tour**.
- À chaque tour, un joueur doit choisir une **action** :
    1. **Attaquer** un adversaire.
    2. **Se défendre** pour réduire les dégâts du prochain tour.
    3. **Utiliser sa capacité spéciale**.
- Si une créature tombe à **0 PV**, elle est éliminée.
- Le dernier survivant remporte la partie.

Classes à créér

Chaque `Creature` possède :

- Un **nom**
- Des **points de vie** (PV)
- Une **attaque de base**
- Une **défense**
- Une **capacité spéciale** (optionnelle ou contextuelle)

Chaque classe dérivée de `Creature` peut redéfinir un ou plusieurs des comportements suivants :

- `attaquer(cible: Creature)`
- `seDefendre()`
- `utiliserCapacite(cibles: Creature[], joueur?: Joueur)`

Types de créatures

Type	PV	Attaque	Défense	Capacité spéciale (action 3)
Dragon	100	20	10	**Souffle de feu** : attaque toutes les créatures (25% de réussite sur chaque créature)
Loup	80	15	5	**Rage** : effectue automatiquement une **double attaque** (50% de chance pour la deuxième) quand il attaque
Gobelin	60	10	3	**Esquive** : active un état d'esquive pour le prochain tour (50 % de chance d'éviter une attaque)
Vampire	90	18	8	**Drain de vie** : attaque un ennemi de `14` et récupère la moitié des dégâts infligés en PV
Golem	120	10	15	**Peau de pierre** : se met en mode pierre pour réduire tous les dégâts subis pendant ce tour

Résumé :

Action choisie	Dragon	Loup	Gobelin	Vampire	Golem
Attaquer	attaque simple	attaque avec chance de double coup	attaque simple	attaque simple	attaque simple

Action choisie	Dragon	Loup	Gobelin	Vampire	Golem
Défendre	réduit dégâts	réduit dégâts	réduit dégâts	réduit dégâts	réduit dégâts
Capacité	attaque tous (25%)	**n/a** *(capacité passive)*	active esquive (30%)	attaque + vol de PV	réduit dégâts (×0.5) pendant le tour

Travail à faire

Vous devez utiliser l'**abstraction** et le **polymorphisme** vu en cours, ainsi que les concepts déjà vu dans les TDs précédents (**héritage**, **encapsulation**).

1. Classe abstraite `Creature`

Implémentez, en respectant les principes d'**encapsulation**, une classe `Creature` avec les attributs et les méthodes suivantes :

- `attaquer(cible: Creature)`
- `seDefendre()`
- `utiliserCapacite(cibles: Creature[], joueur?: Joueur)`
- `recevoirDegats(amount: number)`

Reflechissez où est l'intérêt de l'abstraction.

2. Sous-classes de `Creature`

Créez les cinq classes filles : `Dragon`, `Loup`, `Gobelin`, `Vampire`, `Golem` et redéfinissez les méthodes si nécessaire.

Attention : Certaines capacités sont passives (ex : le loup), d'autres doivent être activées en tant qu'action spécifique (ex : le dragon).

3. Classe `Joueur`

Chaque joueur a :

- un nom
- une créature
- la méthode `jouerTour()` pour choisir l'action à effectuer

4. Classe `Combat`

Créez une classe `Combat` qui gère :

- la liste des joueurs
- l'ordre des tours
- la suppression des joueurs éliminés
- la boucle principale du combat

Ajoutez des messages pour afficher les actions, les dégâts, les PV restants, etc.

5. UML

Effectuez le **diagramme de classes** du jeu.