Prendre en main la programmation orientée objet
Compétences évaluées

- Prendre en main la programmation orientée objet

Description

Le projet Epicrafters' Journey bat son plein. Tandis que l'implémentation de la gestion de blocs de construction est en cours de réalisation, nous allons ici nous concentrer sur la gestion des personnages. Durant ce quiz, vous serez ainsi amenés à réaliser le code nécessaire pour que des personnages existent dans le jeu.
Tous les personnages ont un nom mais il en existe plusieurs catégories. Dans cette première version il y aura :

- Joueur : c'est le personnage classique et le plus commun.
- Héro : c'est un personnage particulier qui intervient pour résoudre les pires situations
- Méchant : c'est un personnage particulier qui est la cause des pires situations

Tous les personnages ont un nom. Les joueurs ont également des points de vie. Les héros ont des points d'attaques, cela leur permet d'éliminer un méchant. Les méchants ont des points de défenses, ainsi pour être éliminé un héro doit réduire ses points de défenses à 0 grâce à ses points d'attaques.

Question 1 : Vous recevez la tâche suivante :
ID de la tâche : 1
Nom de la tâche : Créer la classe Joueur
Description de la tâche :
La définition métier d'un "Joueur" est un personnage avec un nom et des points de vie.
Créer une classe Java respectant cette définition, permettant d'instancier un Personnage en spécifiant les valeurs des caractéristiques.
Quel code Java écrivez-vous ?

```
public class Joueur {
    private String nom;
    private int pointsDeVie;

    public Joueur(final String nom, final String pointsDeVie) {
        this.nom = nom;
        this.pointsDeVie = pointsDeVie;
    }
}
```

```
public interface Joueur {
    private String nom;
    private int pointsDeVie;

    public Joueur(final String nom, final String pointsDeVie) {
        this.nom = nom;
        this.pointsDeVie = pointsDeVie;
    }
}
```

```
public class Joueur {
    public String nom;
    public int pointsDeVie;

    public Joueur construct(final String nom, final String pointsDeVie) {
```

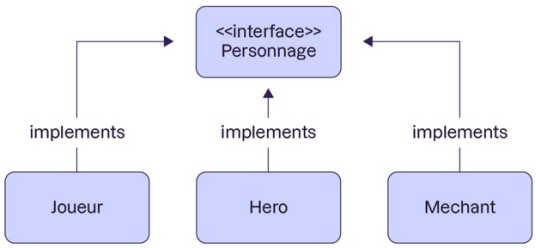
```
        this.nom = nom;
        this.pointsDeVie = pointsDeVie;
    }
}
```

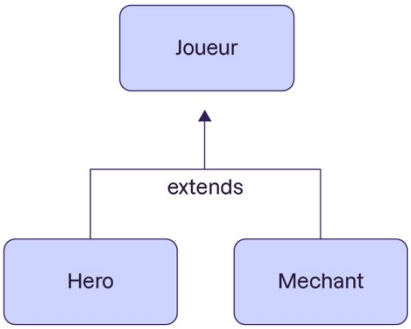
Question 2 : Les tâches 2 et 3 concernent respectivement la création des classes Hero et Mechant. Jerry, un autre développeur de l'équipe, a réalisé la tâche 2 et voici le code qu'il a écrit :

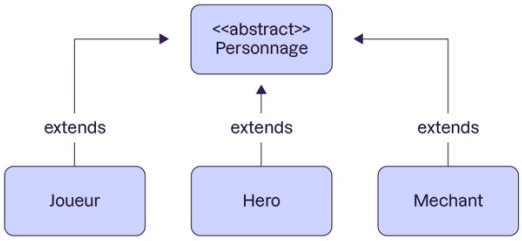
```
public class Hero {
    private String nom;
    private int pointsDAttaques;

    public Hero(final String nom, final String pointsDAttaques) {
        this.nom = nom;
        this.pointsDAttaques = pointsDAttaques;
    }
}
```

Vous analysez le code de classe Joueur et de la classe Hero. Quelle problématique identifiez-vous ?
La classe Hero ne possède pas un attribut de type Joueur.
Les deux classes n'ont pas d'accesseurs pour leurs attributs.
L'attribut nom est dupliqué dans ces deux classes.

Question 3 : Pour organiser les classes Joueur, Hero et Mechant, quelle solution soumettez-vous à Marc, le tech lead de l'équipe ?







Question 4 : Vous recevez la tâche suivante :
ID de la tâche : 4
Nom de la tâche : Implémenter la notion de catégorie pour les personnages
Description de la tâche :
Il a été défini que tous les personnages seront associés à une catégorie. Les catégories sont assimilées à des métiers et sont au nombre de 4 : Scientifique, Médecin, Ouvrier et Soldat.
Ces catégories sont fixes et ne comportent aucune autre information si ce n'est leur nom.
Qu'allez-vous créer comme élément de programmation orienté objet pour coder les catégories ?
un **record** pour chaque catégorie avec le nom de la catégorie en attribut.
un **enum** qui contiendra les quatre catégories.
une **interface** pour chaque catégorie.

Question 5 : Marc s'est occupé d'une autre tâche qui consistait à créer une interface nommée IPersonnage et qui déclare la méthode suivante :
public void construireUnBloc(IBloc bloc); .
Le code est donc :

```
public interface IPersonnage {
    public void construireUnBloc(IBloc bloc);
}
```
Les classes Joueur, Hero et Mechant implémentent désormais cette interface. Quel sera l'effet sur ces classes ?
Elles devront toutes écrire le code de la méthode construireUnBloc;
Elles ne seront pas impactées par cette modification.
Elles devront toutes hériter d'une autre classe qui implémente construireUnBloc.
Question 6 : Au vu du code suivant :
```
public class Main {
    public static void main(String[] args) {
        List<IPersonnage> personnages = new ArrayList<IPersonnage>();
        personnages.add(new Joueur("Bob", 100);
        personnages.add(new Hero("Batman", 50);
        personnages.add(new Mechant("Joker", 150);
        for(IPersonnage personnage : personnages) {
            personnage.construireUnBloc(new Bloc());
        }
    }
}
```
Quelle appréciation de code allez-vous fournir lors d'une session de revue de code ?
Compilation et exécution valide : Une instance peut être typée par l'interface qui est implémentée. Par polymorphisme l'implémentation de la méthode déclarée dans l'interface sera automatiquement exécutée via l'instructionpersonnage.construireUnBloc(new Bloc());.
Compilation valide mais erreur à l'exécution : Ce code échoue à l'exécution bien que la compilation ne pose pas de problèmes. Le typage via l'interface IPersonnage dans la liste ne permettra pas au code de savoir quelle implémentation de constuireUnBloc il faut exécuter.
Échec à la compilation et aucune exécution possible : Bien que les classes Joueur, Hero et Mechant implémentent l'interface IPersonnage, le compilateur Java ne permet pas d'instancier ces objets et de les ajouter dans la liste dans une liste qui manipule des objets typés par l'interface.
Question 7 : Une nouvelle direction est donnée et finalement.
Voici les nouvelles règles à appliquer aux classes Joueur, Hero et Mechant :

- Tous les personnages ont un nom et des points de vie.
- Les héros ont des points d'attaques pour leur permettre d'éliminer un méchant.
- Les méchants ont des points de défenses, ainsi pour être éliminé il faut réduire ses points de défenses à 0 puis réduire ses points de vie à 0.

La classe Mechant a le code suivant :
```
public class Mechant extends Personnage {
    private int pointsDeDefenses;
    public Mechant(final int pointsDeDefenses) {
        this.pointsDeDefenses = pointsDeDefenses;
    }
    public boolean subiUneAttaque(final int pointDAttaques) {
        pointsDeDefenses = pointDeDefenses - pointDAttaques;
        if(pointsDeDefenses < 0) {
            return true;
        }
        return false;
    }
}
```
La méthode subiUneAttaque renvoie faux si le méchant a encore des points de défenses et vrai s'il n'en a plus.
La tâche suivante vous est assignée :
ID de la tâche : 5
Nom de la tâche : Implémenter une exception si un méchant déjà éliminé se fait attaquer

Description de la tâche :
Lorsqu'un méchant n'a plus de points de défense, il est éliminé. Si un héros attaque un méchant éliminé, il faut remonter une exception MechantElimineException. La classe MechantElimineException a déjà été créée.
Quelle modification proposez-vous pour la méthode subiUneAttaque ?

```
public boolean subiUneAttaque(final int pointDAttaques) {
        if(pointsDeDefenses < 0) {
            return new MechantElimineException();
        }
        pointsDeDefenses = pointDeDefenses - pointDAttaques;
        if(pointsDeDefenses < 0) {
            return true;
        }
        return false;
    }
```

```
public boolean subiUneAttaque(final int pointDAttaques) throws MechantElimineException {
if(pointsDeDefenses < 0) {
        throw new MechantElimineException();
    }
    pointsDeDefenses = pointDeDefenses - pointDAttaques;
    if(pointsDeDefenses < 0) {
        return true;
    }
    return false;
}
```

```
public boolean subiUneAttaque(final int pointDAttaques) {
    try {
        pointsDeDefenses = pointDeDefenses - pointDAttaques;
        if(pointsDeDefenses < 0) {
            return true;
        }
        return false;
    } catch(MechantElimineException exception) {
        System.out.println("Le mechant est déjà éliminé");
    }
}
```

Question 8 : Vous devez comptabiliser le nombre de méchants qu'un héros a vaincu mais vous ne pouvez pas modifier le contenu de la classe Hero.
Marc, le tech lead, vous suggère d'utiliser une collection pour associer une instance de héros et le nombre de méchants vaincus.
Quelle solution technique choisirez-vous ?

```
public class Main {
    public static void main(String[] args) {
        Hero hero = new Hero("Superman", 100);
        // … le code qui permet de vaincre des méchants n'est pas écrit par simplification mais il pourrait être exécuté ici.
        Map<Hero, Integer> classement = new HashMap<Hero, Integer>();
        classement.put(hero, 2); // 2 victoires
}
}
```

```
public class Main {
```

```
    public static void main(String[] args) {
        Hero hero = new Hero(“Superman”, 100);
        // … le code qui permet de vaincre des méchants n’est pas écrit par simplification mais il pourrait être exécuté ici.
        Set<Object> classement = new LinkedHashSet<Object>();
        classement.add(hero);
classement.add(2); // 2 victoires
}
}
```

```
public class Main {
    public static void main(String[] args) {
        Hero hero = new Hero(“Superman”, 100);
        // … le code qui permet de vaincre des méchants n’est pas écrit par simplification mais il pourrait être exécuté ici.
        List<Hero> classement = new ArrayList<Hero>();
        classement.add(hero); // 1ère victoire
classement.add(hero); // 2ème victoire
}
}
```