OPENCLASSROOMS

Corrigé Activité P1C5

Voici un exemple de corrigé pour cette activité :

Python

```python
# ---------- Classes de Base ----------

class Personne:

    # Classe mère : contient les attributs communs

    def __init__(self, nom: str, number: int):

        self.nom = nom

        # Utilisation de l'Encapsulation (convention _ pour l'attribut interne)

        self._number = number

        print(f"Personne de base créée : {self.nom}")

    # Méthode de base héritée

    def get_info(self):

        return f"Nom: {self.nom}, ID: {self._number}"

# ---------- Mix-ins (Fonctionnalités modulaires) ----------

class JoueurMixin:

    # Ce Mix-in ajoute les fonctionnalités liées au jeu

    def jouer(self):

        # La méthode utilise self.nom (hérité de Personne)

        print(f"{self.nom} est en train de jouer une partie d'échecs.")
```

```
class ArbitreMixin:
    # Ce Mix-in ajoute les fonctionnalités liées à l'arbitrage
    def arbitrer(self):
        print(f"{self.nom} est en train d'arbitrer un match, il vérifie les règles.")

# ---------------- Classe Complexe (Héritage Multiple/Assemblage) ----------------

# La classe ArbitreJoueur hérite de Personne ET des deux Mix-ins.
# L'ordre d'héritage est crucial pour l'initialisation.
class ArbitreJoueur(Personne, JoueurMixin, ArbitreMixin):
    # Surcharge du constructeur pour gérer l'initialisation de la superclasse
    def __init__(self, nom: str, number: int, statut_pro: bool):
        # 1. Appel du constructeur de la classe mère Personne via super().
        # Cela initialise nom et _number.
        super().__init__(nom, number)
        # 2. Initialisation des attributs spécifiques à ArbitreJoueur
        self.statut_pro = statut_pro
        print(f"Rôle complexe Arbitre/Joueur créé pour {self.nom}.")

# ---------------- Démonstration et Instanciation ----------------

# Création de l'instance qui cumule les rôles
arbitre_joueur_bob = ArbitreJoueur("Bob Smith", 205, True)

# Démonstration de l'Héritage (accès aux attributs de Personne)
print(arbitre_joueur_bob.get_info())

# Démonstration de l'Héritage de Mix-ins (accès aux méthodes des deux rôles)
arbitre_joueur_bob.jouer()
```

```
arbitre_joueur_bob.arbitrer()

# ------------------------------------------------------------

# Discussion : Composition vs. Héritage

# La Composition pourrait remplacer cette structure si les rôles (JoueurMixin, ArbitreMixin)

# étaient des objets complets qui gèrent leur propre état. Par exemple :

# class ArbitreJoueur_Comp:

#     def __init__(self, nom, number):

#         self.personne = Personne(nom, number)

#         self.capacite_arbitre = ArbitreModule()

#         self.capacite_joueur = JoueurModule()

#     def jouer(self):

#         self.capacite_joueur.executer_action_de_jeu() # Délégation

# Bien que plus souple, l'Héritage Multiple via Mix-ins est plus direct pour injecter

# de simples comportements sans état complexe.
```

Ce corrigé montre comment la classe ArbitreJoueur obtient la structure de base (via Personne) et agrège ensuite des comportements spécifiques (via JoueurMixin et ArbitreMixin). L'utilisation de super().__init__ garantit que l'état initial hérité est correctement géré. Cette approche est hautement modulaire et illustre comment l'Héritage permet de construire des entités complexes en réutilisant et en assemblant des fonctionnalités existantes.