Programme de Projets POO sur 3 jours (2 heures par jour)

L'objectif de ce projet en **programmation orientée objet (POO)** est de permettre aux élèves de développer un projet concret tout en consolidant leurs compétences en POO. Ils auront à choisir entre deux thèmes : **un projet de jeu vidéo**.

Chaque jour est structuré de manière à encourager la réflexion, le développement, l'amélioration, et la présentation du projet.

Jour 1 : Réflexion et Début du Développement

L’objectif de cette première session est de permettre aux élèves de réfléchir à leur projet, de définir les éléments clés et de poser les bases de leur développement en utilisant la programmation orientée objet (POO). Ils ont le choix entre deux thèmes : un jeu vidéo.

Première Heure : Réflexion et Planification

1. Choix du projet : Jeu vidéo

Les élèves doivent choisir un des deux thèmes suivants, en fonction de leur intérêt et de la complexité qu'ils souhaitent aborder. Le choix du thème va influencer les classes et les méthodes à implémenter.

Option 1 : Jeu vidéo

Le projet peut prendre plusieurs formes, avec des éléments variés comme des **personnages**, des **ennemis**, des **objets ramassables** et des **niveaux**. Voici quelques suggestions :

- **Type de jeu** :
    - **Jeu de plateforme** : Le personnage se déplace horizontalement et verticalement, évite des obstacles, et combat des ennemis.
    - **Jeu de labyrinthe** : Le joueur doit trouver la sortie d’un labyrinthe tout en évitant des ennemis ou en ramassant des objets.
    - **Jeu de combat** : Le personnage principal se bat contre différents ennemis dans un système de tours (turn-based combat).

2. Définition des objets et des classes principales

La prochaine étape est de définir les **classes principales** qui seront utilisées dans le projet. Cette étape est cruciale pour comprendre comment structurer les objets et leurs interactions.

Exemples pour le jeu vidéo :

- **Classe Personnage** : Représente le joueur ou les ennemis.
    - **Attributs** :
        - `nom` : Nom du personnage.

        - `points_de_vie` : Points de vie du personnage (pour savoir s'il est vivant ou mort).
        - `position` : La position actuelle du personnage sur une grille ou dans un monde (coordonnées).
    - **Méthodes** :
        - `se_deplacer(direction)` : Permet au personnage de changer de position (ex: haut, bas, gauche, droite).
        - `attaquer(cible)` : Permet au personnage d'infliger des dégâts à une cible (ennemi ou autre personnage).
        - `perdre_vie(degats)` : Réduit les points de vie du personnage lorsqu'il est attaqué.
- **Classe Ennemi** : Hérite de la classe `Personnage`, mais a des comportements spécifiques comme se déplacer aléatoirement ou attaquer automatiquement le joueur.
- **Classe Objet** : Représente les objets ramassables dans le jeu (potions, armes, clés).
    - **Attributs** :
        - `type_objet` : Le type d'objet (potion, clé, etc.).
        - `effet` : L'effet de l'objet (par exemple, récupérer des points de vie).
    - **Méthodes** :
        - `utiliser()` : Applique l'effet de l'objet sur le personnage.
- **Classe Niveau** : Représente un niveau dans le jeu, avec des plateformes ou des obstacles.
    - **Attributs** :
        - `plateformes` : Une liste de plateformes ou d'obstacles sur lesquels le personnage peut marcher ou sauter.
    - **Méthodes** :
        - `charger_niveau()` : Charge les éléments du niveau à l'écran.

3. Identification des interactions entre les classes

Après avoir défini les classes, les élèves doivent réfléchir aux **relations et interactions** entre ces classes.

Exemples pour le jeu vidéo :

- **Personnage et Objet** : Le personnage peut ramasser un objet en utilisant une méthode comme `ramasser(objet)`, puis utiliser l'objet avec la méthode `utiliser()`.
- **Personnage et Ennemi** : Le personnage peut attaquer un ennemi en appelant la méthode `attaquer(cible)`, ce qui va réduire les points de vie de l'ennemi.
- **Personnage et Niveau** : Le personnage peut se déplacer dans un niveau et interagir avec les plateformes ou obstacles.

4. Rédiger un plan de développement

Le **plan de développement** permet de définir les étapes à suivre pour la construction du projet. Voici un exemple de structure logique pour les deux thèmes.

Exemple pour un jeu vidéo :

1. **Créer la classe Personnage** :
    - Définir les attributs (`nom`, `points_de_vie`, `position`).
    - Implémenter les méthodes de base comme `se_deplacer(direction)` et `attaquer(cible)`.
2. **Créer la classe Ennemi** (héritant de Personnage) :
    - Ajouter des comportements spécifiques comme `se_deplacer_aleatoirement()`.
3. **Créer la classe Objet** :
    - Définir les objets ramassables dans le jeu, avec leurs effets sur le personnage.
4. **Créer la classe Niveau** :
    - Modéliser un niveau avec des plateformes et obstacles.
5. **Tester l'interaction** :
    - Simuler un jeu où le personnage se déplace, ramasse des objets, et combat des ennemis.

A suivre : Commencer le Développement

Une fois la planification terminée, les élèves peuvent **commencer à coder** leur projet. Voici quelques étapes clés pour commencer :

1. **Initialisation des classes** :
    - Écrire les **constructeurs (`__init__`)** pour les classes principales (Personnage, etc.).
2. **Développement des méthodes de base** :
    - Implémenter des méthodes simples comme `se_deplacer()` pour un personnage.
3. **Tests préliminaires** :
    - Effectuer des tests basiques pour s'assurer que les interactions fonctionnent comme prévu (par exemple, déplacer un personnage).

Cette première session permet aux élèves de bien structurer leur projet et de poser des bases solides avant de passer aux **améliorations et corrections** lors du jour suivant.

Deuxième Heure : Commencer le Développement

L’objectif de cette session est de permettre aux élèves de **débuter la programmation** de leur projet en utilisant la **programmation orientée objet (POO)**. Cela inclut l’initialisation des classes principales, l’écriture des constructeurs et des méthodes, ainsi que des tests préliminaires pour s’assurer que tout fonctionne comme prévu.

1. Initialisation du projet

Les élèves doivent maintenant créer les **classes principales** qu'ils ont définies lors de la première heure. Voici les étapes détaillées :

Écriture des constructeurs (`__init__`)

Les constructeurs sont des méthodes spéciales utilisées pour initialiser les objets d’une classe. Les élèves devront définir les attributs de chaque classe pour qu'ils puissent être utilisés tout au long du programme.

Exemple pour un jeu vidéo :

Voici un exemple d’implémentation pour la classe `Personnage` :

```
class Personnage:
    def __init__(self, nom, points_vie, position):
        self.nom = nom  # Nom du personnage
        self.points_vie = points_vie  # Points de vie du personnage
        self.position = position  # Position sur une grille (ex: [x, y])

    def se_deplacer(self, direction):
        # Logique de déplacement basée sur la direction
        if direction == "droite":
            self.position[0] += 1  # Déplacement à droite
        elif direction == "gauche":
            self.position[0] -= 1  # Déplacement à gauche
        elif direction == "haut":
            self.position[1] -= 1  # Déplacement vers le haut
        elif direction == "bas":
            self.position[1] += 1  # Déplacement vers le bas
        print(f"{self.nom} s'est déplacé vers {direction} et est maintenant à
{self.position}.")
```

Dans cet exemple, le personnage a trois attributs : `nom`, `points_vie`, et `position`. La méthode `se_deplacer()` met à jour la position en fonction de la direction spécifiée.

2. Tester les premières classes et interactions

Après avoir défini les classes et leurs méthodes, les élèves doivent effectuer des **tests initiaux** pour vérifier que leurs objets peuvent interagir correctement. Ces tests permettront de détecter d'éventuelles erreurs dans la logique ou les méthodes.

Tests simples pour le jeu vidéo

Les élèves peuvent créer un personnage et tester sa capacité à se déplacer :

```
# Création d'un personnage
```

```
personnage1 = Personnage("Héros", 100, [0, 0])

# Tester le déplacement
personnage1.se_deplacer("droite")  # Attendu : [1, 0]
personnage1.se_deplacer("haut")    # Attendu : [1, -1]
```

Ce qui doit être vérifié :

- La position du personnage change correctement après chaque appel de la méthode `se_deplacer()`.
- Les messages de confirmation s'affichent correctement.

Conclusion de la Deuxième Heure

À la fin de cette heure, les élèves auront mis en place les bases de leur projet, avec des classes fonctionnelles et des méthodes testées. Ces premières étapes sont cruciales pour le développement futur de l'application. L'étape suivante consistera à améliorer et réviser leurs projets en ajoutant de nouvelles fonctionnalités et en corrigeant les bugs éventuels.

Jour 2 : Amélioration et Correction

L'objectif de cette deuxième journée est d'améliorer les projets développés lors du **jour 1**, en y ajoutant des fonctionnalités et en corrigeant les éventuels problèmes. Cette journée met l'accent sur le **raffinement du code** et la **finalisation des fonctionnalités clés**.

Première Heure : Améliorations du Projet

1. Analyse des fonctionnalités manquantes

Les élèves doivent d'abord examiner leur projet et réfléchir aux **fonctionnalités supplémentaires** ou **améliorations** qu'ils souhaitent ajouter. Cette étape permet d'élargir le projet tout en mettant en œuvre des concepts plus avancés.

Pour le jeu vidéo :

Les élèves peuvent réfléchir à des éléments comme :

- **Ajouter des ennemis supplémentaires** : Chaque ennemi peut avoir ses propres caractéristiques, comme des points de vie et un comportement d'attaque.
- **Système de points de vie** : Les personnages et les ennemis peuvent perdre des points de vie pendant les combats.
- **Système de niveaux** : Quand un joueur atteint un certain point ou tue tous les ennemis, il peut passer à un niveau supérieur.

2. Ajouter de nouvelles fonctionnalités

Après avoir identifié les fonctionnalités manquantes, les élèves commencent à coder et à intégrer ces nouvelles fonctionnalités.

Exemples d'améliorations pour le jeu vidéo :

1. **Système de points de vie pour les personnages et ennemis :** Chaque personnage (ou ennemi) peut perdre des points de vie lorsqu'il est attaqué. Si ses points de vie tombent à 0, il est éliminé.

```
class Personnage:
    def __init__(self, nom, points_vie, position):
        self.nom = nom
        self.points_vie = points_vie
        self.position = position

    def attaquer(self, cible):
        cible.points_vie -= 10  # Réduire les points de vie de la cible
        if cible.points_vie <= 0:
            print(f"{cible.nom} est vaincu !")
```

2. **Ajout d'un système de niveaux :** Les élèves peuvent créer plusieurs niveaux. Par exemple, lorsqu'un personnage atteint une certaine position ou tue tous les ennemis, il passe au **niveau suivant**.

```
class Niveau:
    def __init__(self, numero, obstacles, ennemis):
        self.numero = numero
        self.obstacles = obstacles
        self.ennemis = ennemis

    def est_complet(self):
        return len(self.ennemis) == 0  # Si tous les ennemis sont vaincus

# Exemple de passage à un niveau supérieur :
if niveau.est_complet():
    print(f"Niveau {niveau.numero} terminé ! Passage au niveau suivant.")
```

3. Implémentation et Tests

Après avoir codé ces nouvelles fonctionnalités, les élèves doivent s'assurer que leur programme fonctionne correctement. Ils doivent tester leurs modifications en exécutant plusieurs scénarios pour voir comment les différents objets interagissent.

Tests pour le jeu vidéo :

- **Tester le combat entre le personnage et un ennemi** :

```
heros = Personnage("Héros", 100, [0, 0])
ennemi = Personnage("Gobelin", 30, [1, 0])

# Combat :
heros.attaquer(ennemi)  # Gobelin perd 10 points de vie
heros.attaquer(ennemi)  # Gobelin perd encore 10 points de vie
heros.attaquer(ennemi)  # Gobelin est vaincu
```

- **Tester la transition entre les niveaux** :

```
niveau1 = Niveau(1, [], [ennemi])
if niveau1.est_complet():
    print("Niveau terminé !")
else:
    print("Il reste des ennemis à vaincre.")
```

Deuxième Heure : Correction de Bugs et Optimisations

Après avoir ajouté de nouvelles fonctionnalités et fait des tests, les élèves passent à la phase de **correction de bugs** et à l'**optimisation** de leur code.

1. Identifier les bugs potentiels

Les élèves doivent tester le programme dans différents scénarios pour repérer d'éventuels **bugs**. Cela pourrait inclure :

- Des erreurs logiques (par exemple, un personnage se déplaçant en dehors des limites du jeu).

2. Correction des bugs

Lorsque des bugs sont identifiés, les élèves doivent les corriger en ajustant leur code. Par exemple :

- **Pour le jeu vidéo ,** vérifier que les positions restent dans les limites du niveau.

3. Optimisation du code

Les élèves peuvent également chercher à rendre leur code plus efficace ou plus clair :

- **Simplifier les méthodes** : Éviter la répétition de code en utilisant des boucles ou des fonctions auxiliaires.
- **Améliorer la lisibilité** : Ajouter des commentaires ou nommer les variables de manière plus claire.

Conclusion :

À la fin de cette deuxième journée, les élèves auront amélioré leur projet en ajoutant des fonctionnalités importantes et en s'assurant que tout fonctionne correctement. Cela inclut des tests complets pour identifier les bugs potentiels et les corriger. Ils se préparent également à finaliser et présenter leur projet lors du **jour 3**.

Deuxième Heure : Corrections de bugs et optimisation

Cette heure est dédiée à l'**affinage du projet**, en corrigeant les erreurs rencontrées lors des tests, en optimisant le code pour améliorer ses performances, et en garantissant qu'il est lisible et maintenable. Cette étape est cruciale car elle permet aux élèves de perfectionner leur travail et d'apprendre l'importance du **débogage** et de l'**optimisation** dans le développement logiciel.

1. Débogage

Revue des erreurs et problèmes rencontrés lors des tests

Pendant la première heure du jour 2, les élèves ont testé leurs nouvelles fonctionnalités. À ce stade, ils doivent s'assurer que tous les **bugs** ou comportements inattendus sont corrigés. Voici quelques types de problèmes qu'ils pourraient rencontrer :

- **Erreurs de syntaxe** : Problèmes de parenthèses, mauvais noms de variables, fautes d'indentation, etc. Python est particulièrement sensible à l'indentation, donc des erreurs peuvent facilement survenir si le code n'est pas bien formaté.
- **Boucles infinies** : Les boucles qui ne s'arrêtent jamais parce que la condition d'arrêt n'est pas définie ou mal placée.

  Exemple :

  ```
  while personnage.points_vie > 0:
      personnage.attaquer(ennemi)
  ```

  Si l'ennemi n'est jamais vaincu ou si `personnage.points_vie` ne diminue pas correctement, cela pourrait créer une boucle infinie. Les élèves devront vérifier que les conditions d'arrêt des boucles sont bien respectées.
- **Conflits de méthode ou variables** : Lorsque plusieurs méthodes ou variables portent le même nom dans différentes classes, cela peut provoquer des comportements inattendus.

  Exemple : Si une classe de `Personnage` a une méthode `attaquer()` et qu'une autre classe héritant de cette classe en redéfinit la méthode sans la référencer correctement, cela pourrait entraîner des conflits ou la mauvaise méthode pourrait être appelée.

Méthodologie de débogage

Les élèves doivent aborder le débogage de manière systématique. Voici quelques étapes à suivre pour identifier et résoudre les erreurs :

1. **Reproduire le bug** : Les élèves doivent d'abord reproduire systématiquement l'erreur qu'ils ont identifiée. Cela leur permettra de comprendre dans quelles conditions le bug se manifeste.
2. **Lire les messages d'erreur** : Python fournit des **messages d'erreur** très détaillés qui peuvent aider à localiser le problème. Les élèves doivent lire attentivement ces messages pour comprendre l'origine du bug.

3. **Utiliser des impressions (debugging par `print`)** : Une méthode simple pour comprendre ce qui se passe dans le code consiste à ajouter des `print()` à des endroits stratégiques pour suivre l'exécution du programme.

   Exemple :

   ```
   def se_deplacer(self, direction):
       print(f"Position actuelle : {self.position}")
       if direction == "droite":
           self.position[0] += 1
       print(f"Nouvelle position : {self.position}")
   ```

4. **Vérification des entrées et sorties des fonctions** : Ils doivent s'assurer que les **valeurs passées aux fonctions** sont correctes et que les **retours des fonctions** sont conformes à ce qu'ils attendent.
5. **Test pas à pas** : S'il y a une erreur logique complexe, ils peuvent tester le code **ligne par ligne** pour voir comment les variables évoluent à chaque étape.
6. **Revérification des boucles et conditions** : Les élèves doivent vérifier que toutes les boucles ont des **conditions de sortie** correctes et que les blocs conditionnels (`if`, `else`, etc.) sont bien alignés avec la logique attendue.

2. Optimisation du code

Une fois les erreurs corrigées, les élèves peuvent améliorer l'**efficacité** et la **lisibilité** de leur code. Cela les aide non seulement à produire un projet plus élégant, mais aussi à mieux comprendre comment rendre leur code plus performant.

Réorganiser les méthodes

L'optimisation commence par un examen de la manière dont les méthodes et les classes sont structurées. Voici quelques points d'optimisation possibles :

- **Éviter la redondance** : Si plusieurs morceaux de code se répètent, il est conseillé de les regrouper dans une méthode ou fonction réutilisable.

  Exemple : Si plusieurs personnages ont des comportements similaires dans le jeu, les élèves peuvent créer une méthode partagée dans la classe `Personnage` plutôt que de dupliquer le code.

  ```
  class Personnage:
      def attaquer(self, cible):
          degats = self.calculer_degats()
          cible.perdre_points_vie(degats)

      def calculer_degats(self):
          return 10  # Peut être modifié pour chaque personnage
  ```

- **Modularité** : Encourager les élèves à diviser leur code en **petites méthodes réutilisables** plutôt que d’avoir de longues méthodes complexes. Cela rend le code plus facile à lire et à maintenir.

  Exemple :

- Diviser une méthode `jouer_niveau()` en plusieurs petites méthodes comme `gerer_mouvement()`, `gerer_combat()`, etc.

Ajouter des commentaires clairs

Les élèves doivent également apprendre à **documenter leur code** à l'aide de commentaires. Cela les aide à clarifier les parties complexes du programme, et facilite la relecture pour eux-mêmes ou pour d'autres.

- **Commentaires descriptifs** : Chaque méthode ou bloc de code complexe doit être accompagné d'une **explication concise** de ce qu'il fait.

  Exemple :

  ```
  class Personnage:
      def __init__(self, nom, points_vie):
          # Initialise un personnage avec un nom et des points de vie
          self.nom = nom
          self.points_vie = points_vie
  ```

- **Utilisation des docstrings** : Pour documenter les classes et les méthodes de manière plus formelle, ils peuvent utiliser des **docstrings**.

  Exemple :

  ```
  def attaquer(self, cible):
      """
      Attaque un ennemi et réduit ses points de vie.

      Arguments:
      cible -- Le personnage ciblé par l'attaque
      """
      cible.points_vie -= 10
  ```

Conclusion de la deuxième heure

À la fin de cette session de correction et d'optimisation, les élèves doivent avoir :

- Corrigé les **bugs** identifiés lors des tests.
- **Optimisé** leur code pour le rendre plus lisible et performant.
- Ajouté des **commentaires** pour expliquer les parties complexes de leur projet.
- Validé que toutes les fonctionnalités implémentées fonctionnent de manière fluide.

Cela les prépare pour la dernière journée, où ils finaliseront et présenteront leur projet devant leurs camarades.

Jour 3 (2 heures) : Finalisation et Présentation

Cette journée marque la conclusion du projet, avec une première heure consacrée à la finalisation et une seconde heure dédiée à la présentation devant la classe. C'est une étape essentielle où les élèves devront peaufiner leur projet, effectuer les derniers tests, et organiser leur présentation pour mettre en valeur leur travail.

Première Heure : Finalisation du Projet

1. Nettoyage du code et ajout des détails finaux

Avant de considérer leur projet comme terminé, les élèves doivent procéder à une dernière révision de leur code. Voici quelques points clés à aborder :

1. **Nettoyage du code :**
    - **Organisation et lisibilité :** Les élèves doivent veiller à ce que leur code soit bien structuré, lisible et facile à comprendre. Cela implique de :
        - Supprimer le code inutile ou les parties commentées qui ne sont plus nécessaires.
        - Réorganiser les méthodes et les classes pour améliorer la lisibilité.
        - Ajouter des **commentaires** explicatifs là où le code pourrait sembler complexe ou difficile à comprendre.

    Exemple :

```
# Classe représentant un personnage dans le jeu
class Personnage:
    def __init__(self, nom, points_vie):
        self.nom = nom
        self.points_vie = points_vie  # Nombre de points de vie du
personnage

    def attaquer(self, cible):
        """
        Réduit les points de vie de la cible lorsque le personnage
attaque.
        """
        degats = 10
        cible.points_vie -= degats
        print(f"{self.nom} attaque {cible.nom} et lui inflige {degats}
points de dégâts.")
        if cible.points_vie <= 0:
            print(f"{cible.nom} est vaincu!")
```

2. **Ajout de détails finaux :**
    - **Interface utilisateur basique** : Si possible, les élèves peuvent ajouter une **interface utilisateur simple** comme des menus en console pour naviguer dans les options.

      Exemple pour un jeu :

```
while jeu_en_cours:
    action = input("Que voulez-vous faire ? (attaquer / se déplacer)
: ")
    if action == "attaquer":
```

```
            joueur.attaquer(ennemi)
        elif action == "se déplacer":
            direction = input("Dans quelle direction ? (gauche / droite)
: ")
            joueur.se_deplacer(direction)
```

3. **Structuration générale du programme :**

    - Les élèves doivent s'assurer que leur code est **séparé en sections logiques** (par exemple, définition des classes, implémentation des fonctions principales, partie exécution du programme).
    - Ils peuvent également s'assurer qu'il y a une **méthode principale** (`main()`) qui lance le programme, en suivant une convention de codage propre.

2. Test Final

Une fois que le code est propre et bien organisé, les élèves doivent effectuer un **test final complet** de leur projet. Ce test permet de s'assurer que tout fonctionne comme prévu et qu'il n'y a pas de bugs ou de fonctionnalités manquantes.

Étapes à suivre pour le test final :

1. **Tester toutes les fonctionnalités principales :**

    - Si le projet est un jeu vidéo, ils doivent s'assurer que le personnage peut se déplacer, attaquer, et interagir avec les objets ou ennemis.

    Exemple :

    ```
    # Tester si le personnage peut attaquer un ennemi
    joueur.attaquer(ennemi)
    assert ennemi.points_vie < 100, "Le personnage n'inflige pas de dégâts
    correctement."
    ```

2. **Validation des performances :**

    - Les élèves doivent vérifier que leur programme fonctionne bien même lorsque le nombre d'ennemis augmente.

3. Préparation de la présentation

Les élèves doivent prendre un moment pour préparer une **présentation structurée** de leur projet. L'objectif est de démontrer leur compréhension des concepts de **programmation orientée objet (POO)**, ainsi que les différentes étapes du développement de leur projet.

Voici quelques points clés à inclure dans leur présentation :

1. **Introduction au projet :**
    - Présenter le thème choisi (jeu vidéo).
    - Expliquer l'objectif du projet et ce qu'il permet de faire.
2. **Concepts de POO utilisés :**

- Présenter les **classes principales** créées (par exemple, `Personnage`, etc.).
- Expliquer l'utilisation de l'**héritage** (si applicable), des **méthodes**, des **attributs**, et la manière dont les objets interagissent entre eux.

Exemple :

- Dans un jeu, un personnage hérite de certaines caractéristiques d'une classe de base et ajoute ses propres comportements.

3. **Fonctionnalités principales :**

   - Décrire les fonctionnalités principales implémentées, comme le combat dans le jeu.
   - Montrer des extraits de code qui illustrent des parties clés du projet.

   Exemple : Montrer comment l'attaque est gérée dans un jeu vidéo :

```
def attaquer(self, cible):
    cible.points_vie -= 10
    if cible.points_vie <= 0:
        print(f"{cible.nom} est vaincu!")
```

4. **Défis rencontrés :**

   - Discuter des principaux **problèmes** ou **défis techniques** rencontrés pendant le développement (par exemple, difficultés avec les boucles, gestion des erreurs utilisateurs, etc.).
   - Expliquer comment ces défis ont été résolus.

5. **Conclusion et Démonstration :**

   - Terminer en montrant une **démonstration pratique** du projet. L'élève peut exécuter le programme, montrer les interactions, et répondre aux questions des camarades ou de l'enseignant.

Deuxième Heure : Présentation

Cette phase est cruciale pour évaluer le travail réalisé par chaque groupe, tout en favorisant un échange constructif entre les élèves. Chaque groupe a l'occasion de partager son projet, de montrer ses fonctionnalités et d'expliquer les défis rencontrés pendant le développement. Ensuite, le feedback et les discussions permettent d'identifier les points forts et les axes d'amélioration des projets.

1. Présentation par chaque groupe (30 à 40 minutes)

Chaque groupe dispose d'environ **7 à 10 minutes** pour présenter son projet. Voici les étapes clés de la présentation à suivre pour chaque groupe :

a. Description du projet

- **Présentation du thème choisi** : Les élèves commencent par présenter leur projet, en expliquant s'ils ont choisi de développer un **jeu vidéo**.

  Exemple pour un jeu vidéo :

  - "Nous avons choisi de créer un jeu vidéo où le joueur contrôle un personnage qui doit explorer des niveaux en évitant des obstacles et en combattant des ennemis. Le but est de terminer chaque niveau sans perdre tous ses points de vie."

b. Démonstration du programme en fonctionnement

Une fois la description terminée, le groupe effectue une **démonstration en temps réel** de son programme. Cela permet à toute la classe de voir concrètement comment le projet fonctionne.

Étapes de la démonstration :

1. **Lancer le programme** : Le groupe montre comment démarrer le programme. Si c'est un jeu vidéo, ils peuvent montrer les premiers niveaux ou combats.
2. **Montrer les interactions clés** :
   - **Pour un jeu vidéo** : Démontrer comment le personnage se déplace, attaque les ennemis, ou interagit avec des objets dans le jeu.
3. **Explorer plusieurs cas d'utilisation** :
   - Si possible, le groupe devrait montrer plusieurs fonctionnalités, par exemple : ce qui se passe quand le joueur gagne ou perd dans le jeu.

   Exemple pour un jeu vidéo :

   - "Voici ce qui se passe quand le personnage attaque un ennemi. Si l'ennemi perd tous ses points de vie, il est vaincu."

c. Explication des difficultés rencontrées et des solutions apportées

Après la démonstration, le groupe doit aborder les **défis techniques** qu'ils ont rencontrés pendant le développement du projet et expliquer comment ils ont résolu ces problèmes.

1. **Exemples de difficultés courantes :**
    - **Dans le jeu vidéo** : Problèmes liés aux déplacements du personnage, interactions entre les personnages et les ennemis, ou gestion des niveaux.
2. **Expliquer les solutions trouvées :**
    - Ils doivent détailler les solutions apportées. Par exemple, comment ils ont optimisé certaines parties du code ou ajouté des conditions pour éviter les erreurs d'utilisation.

   Exemple :
    - "Nous avons eu du mal à gérer les attaques entre le personnage et l'ennemi au début. Pour résoudre cela, nous avons créé une méthode distincte qui vérifie les points de vie avant d'autoriser une attaque."

Résumé :

La présentation de chaque groupe doit se concentrer sur :

- **La description du projet**.
- **La démonstration** des fonctionnalités principales.
- **L'explication** des problèmes rencontrés et des solutions trouvées.

2. Feedback et discussions (20 à 30 minutes)

Après chaque présentation, un temps est alloué aux **questions**, aux **suggestions** et aux **discussions collectives**. Ce moment permet à toute la classe de participer activement à l'évaluation des projets et d'enrichir la réflexion collective.

a. Questions des élèves et du professeur

1. **Encourager la classe à poser des questions** :
    - Chaque élève est encouragé à poser des questions sur le projet présenté.
    - Les questions peuvent porter sur la logique du code, les choix de conception, ou les difficultés techniques.

   Exemples de questions :
    - "Comment avez-vous géré le passage entre les niveaux dans le jeu vidéo ?"
2. **Questions du professeur** :
    - Le professeur peut poser des questions pour approfondir les concepts POO utilisés, comme l'héritage ou la gestion des méthodes.

b. Suggestions d'améliorations

Après les questions, des **suggestions** sont offertes pour améliorer le projet. Les élèves et le professeur peuvent proposer des idées pour ajouter des fonctionnalités ou optimiser certaines parties du code.

Exemples de suggestions :

- "Ce serait intéressant d'ajouter une barre de progression pour indiquer combien de niveaux restent dans le jeu."

c. Discussions collectives sur les principes POO observés

Chaque présentation est l'occasion d'identifier les **concepts de POO** (programmation orientée objet) appliqués dans les projets.

1. **Expliquer les concepts POO observés** :
    - Les élèves peuvent discuter de l'importance de la création de classes, des méthodes utilisées, et comment l'héritage a été mis en place.
2. **Discussions sur les bonnes pratiques** :
    - Le professeur peut souligner les **bonnes pratiques de codage** observées, comme la lisibilité du code, la réutilisabilité des classes, ou la clarté des commentaires.

    Exemple :

    - "Dans ce projet, la classe `Personnage` est bien structurée et pourrait être facilement réutilisée dans d'autres applications de gestion de documents."
3. **Identifier les améliorations possibles sur la base des concepts POO** :
    - Le professeur et les élèves peuvent aussi discuter des points à améliorer du point de vue de la conception POO. Par exemple, encourager une meilleure abstraction des classes ou l'ajout de méthodes pour simplifier certaines parties du programme.

Conclusion

Cette deuxième heure permet à chaque groupe de partager son projet avec la classe et d'obtenir un feedback constructif. Les élèves non seulement présentent leur travail, mais apprennent aussi à analyser les projets des autres, tout en renforçant leur compréhension des concepts POO. Les discussions, suggestions et analyses aident à améliorer les compétences en codage et à identifier les bonnes pratiques pour de futurs projets.

Résumé des 3 jours

- **Jour 1** : Réflexion et début du développement avec création des classes de base.
- **Jour 2** : Amélioration, ajout de nouvelles fonctionnalités, et correction de bugs.
- **Jour 3** : Finalisation, tests finaux, et présentation du projet.

Ce programme permet aux élèves d'approfondir leurs compétences en **POO** tout en travaillant sur un projet concret, qu'il soit ludique (jeu vidéo).