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 : **la gestion d'une bibliothèque**.

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 : la gestion d’une bibliothèque.

Première Heure : Réflexion et Planification

1. Choix du projet : Gestion d’une bibliothèque

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 2 : Gestion d’une bibliothèque

L’autre option consiste à développer un système pour gérer une bibliothèque. Les principales entités incluront des **livres**, des **abonnés** et des **bibliothécaires**. Voici les éléments clés :

- **Caractéristiques du projet** :
    - Les **livres** peuvent être empruntés et rendus, et chaque abonné a une liste de livres qu'il a empruntés.
    - Les **bibliothécaires** sont responsables de la gestion des prêts et des retours.
    - Le système doit suivre les dates limites pour les retours et gérer les retards.

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 la gestion d’une bibliothèque :

- **Classe Livre** : Représente les livres disponibles dans la bibliothèque.
    - **Attributs** :
        - `titre` : Titre du livre.
        - `auteur` : Auteur du livre.
        - `statut_emprunt` : Indique si le livre est disponible ou emprunté.

    - **Méthodes** :
        - `emprunter()` : Change le statut du livre à "emprunté".
        - `retourner()` : Remet le statut du livre à "disponible".
- **Classe Abonné** : Représente les personnes qui empruntent les livres.
    - **Attributs** :
        - `nom` : Nom de l'abonné.
        - `livres_empruntes` : Une liste des livres actuellement empruntés par l'abonné.
    - **Méthodes** :
        - `emprunter_livre(livre)` : Ajoute un livre à la liste des livres empruntés.
        - `rendre_livre(livre)` : Retire un livre de la liste des livres empruntés.
- **Classe Bibliothécaire** : Gère les transactions de la bibliothèque.
    - **Attributs** :
        - `nom` : Nom du bibliothécaire.
    - **Méthodes** :
        - `gérer_emprunt(abonné, livre)` : Vérifie si un livre est disponible, et si oui, permet à l'abonné de l'emprunter.

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 la gestion d'une bibliothèque :

- **Abonné et Livre** : L'abonné peut emprunter un livre, ce qui va ajouter ce livre à la liste de `livres_empruntes` de l'abonné et modifier l'attribut `statut_emprunt` du livre à "emprunté".
- **Bibliothécaire et Abonné** : Le bibliothécaire peut vérifier la liste de livres empruntés par un abonné et déterminer s'il a le droit d'emprunter d'autres livres.
- **Bibliothécaire et Livre** : Le bibliothécaire peut gérer l'emprunt ou le retour d'un livre, modifiant ainsi son statut.

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 la gestion d'une bibliothèque :

1. **Créer la classe Livre** :
    - Définir les attributs (`titre`, `auteur`, `statut_emprunt`) et les méthodes `emprunter()` et `retourner()`.
2. **Créer la classe Abonné** :

- Implémenter les méthodes pour emprunter et rendre des livres.

3. **Créer la classe Bibliothécaire** :
    - Ajouter les méthodes pour gérer les prêts et retours.
4. **Simuler l'emprunt de livres** :
    - Tester l'interaction entre abonnés, bibliothécaires et livres.

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 (Livre, etc.).
2. **Développement des méthodes de base** :
    - Implémenter des méthodes simples comme `emprunter()` pour un livre.
3. **Tests préliminaires** :
    - Effectuer des tests basiques pour s'assurer que les interactions fonctionnent comme prévu (par exemple, emprunter un livre).

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 la gestion d’une bibliothèque :

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

```
class Livre:
    def __init__(self, titre, auteur):
        self.titre = titre  # Titre du livre
        self.auteur = auteur  # Auteur du livre
        self.est_emprunte = False  # Statut de l'emprunt du livre

    def emprunter(self):
        if not self.est_emprunte:
            self.est_emprunte = True  # Le livre est maintenant emprunté
            print(f"{self.titre} a été emprunté.")
        else:
            print("Ce livre est déjà emprunté.")
```

Ici, la classe `Livre` définit les attributs `titre`, `auteur`, et `est_emprunte`. La méthode `emprunter()` change le statut du livre en "emprunté" et imprime un message correspondant.

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 la gestion d’une bibliothèque

Les élèves peuvent également créer un livre et tester la méthode d’emprunt :

```
# Création d'un livre
livre1 = Livre("Harry Potter", "J.K. Rowling")

# Tester l'emprunt du livre
livre1.emprunter()  # Attendu : "Harry Potter a été emprunté."
livre1.emprunter()  # Attendu : "Ce livre est déjà emprunté."
```

Ce qui doit être vérifié :

- L’état de `est_emprunte` du livre change correctement lorsqu’il est emprunté.
- Les messages corrects s'affichent lors de l'emprunt et de la tentative d’emprunt d’un livre déjà emprunté.

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 la gestion d'une bibliothèque :

Les élèves peuvent identifier des fonctionnalités supplémentaires comme :

- **Dates de retour des livres** : Chaque livre doit être rendu avant une date limite. Si la date est dépassée, des frais de retard peuvent être appliqués.
- **Gestion des abonnés** : Les abonnés peuvent emprunter plusieurs livres à la fois, mais ils ne peuvent pas emprunter plus d'un certain nombre de livres.
- **Nouvelles méthodes** pour gérer les livres (retour, prolongation d'emprunt, etc.).

2. Ajouter de nouvelles fonctionnalités

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

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**.

Exemples d'améliorations pour la gestion d'une bibliothèque :

1. **Gestion des dates de retour et des frais de retard :** Chaque emprunt doit avoir une date limite, et si le livre est rendu en retard, des frais doivent être ajoutés.

```
from datetime import datetime, timedelta

class Livre:
    def __init__(self, titre, auteur):
        self.titre = titre
        self.auteur = auteur
        self.est_emprunte = False
        self.date_retour = None

    def emprunter(self):
        if not self.est_emprunte:
            self.est_emprunte = True
            self.date_retour = datetime.now() + timedelta(days=14)  # Date de
retour dans 14 jours
            print(f"{self.titre} emprunté. Date de retour : {self.date_retour}")
        else:
```

```
            print("Ce livre est déjà emprunté.")

    def retourner(self):
        if self.est_emprunte:
            if datetime.now() > self.date_retour:
                jours_retard = (datetime.now() - self.date_retour).days
                frais_retard = jours_retard * 1  # 1 euro par jour de retard
                print(f"Retard de {jours_retard} jours. Frais : {frais_retard}
euros.")
            self.est_emprunte = False
            print(f"{self.titre} a été retourné.")
```

2. **Méthode pour retourner un livre :** Les élèves peuvent ajouter une méthode `retourner_livre()` qui marque le livre comme retourné et gère les frais de retard.

```
def rendre_livre(self, livre):
    livre.retourner()
    print(f"{livre.titre} a été rendu.")
```

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 la gestion de la bibliothèque :

- **Tester l'emprunt et le retour d'un livre avec des frais de retard** :

```
livre1 = Livre("1984", "George Orwell")
livre1.emprunter()  # Emprunt du livre

# Simuler un retour en retard :
livre1.date_retour -= timedelta(days=16)  # Simuler un retard de 2 jours
livre1.retourner()  # Retour du livre avec frais de retard
```

Ces tests doivent couvrir tous les scénarios possibles (comme un livre déjà emprunté ou un ennemi avec plus de points de vie) afin de vérifier que tout fonctionne comme prévu.

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 liées à la gestion des dates dans le système de bibliothèque.

2. Correction des bugs

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

- **Pour la bibliothèque**, vérifier que les dates sont bien manipulées (par exemple, un livre ne peut pas être rendu avant son emprunt).

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.
- **Conflits de méthode ou variables** : Lorsque plusieurs méthodes ou variables portent le même nom dans différentes classes, cela peut provoquer des comportements inattendus.

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.
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.
- **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.

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

Optimiser la performance

Les élèves peuvent également chercher des moyens d'améliorer la **performance** de leur programme. Quelques optimisations simples incluent :

- **Utiliser des structures de données efficaces** : Par exemple, pour des recherches fréquentes dans une bibliothèque, utiliser un **dictionnaire** pour stocker les livres (au lieu d'une liste) permet de rechercher un livre par titre ou auteur plus rapidement.

  Exemple :

```
# Optimiser la recherche de livres avec un dictionnaire
bibliotheque = {
    "1984": Livre("1984", "George Orwell"),
    "Brave New World": Livre("Brave New World", "Aldous Huxley")
}

def chercher_livre(titre):
    return bibliotheque.get(titre, None)
```

- **Réduire les répétitions coûteuses** : Si une méthode est appelée plusieurs fois avec les mêmes paramètres, il peut être judicieux de **mémoriser** les résultats pour éviter des recalculs.

  Exemple : Si un personnage a besoin de calculer ses dégâts plusieurs fois, les élèves peuvent stocker ce calcul plutôt que de le recalculer à chaque attaque.

- **Simplifier les boucles** : Si une boucle effectue des calculs complexes ou inutiles, ils peuvent les simplifier. Par exemple, parcourir une liste de livres une seule fois pour rechercher plusieurs critères peut être plus efficace que plusieurs boucles imbriquées.

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.
2. **Ajout de détails finaux :**
    - **Messages pour l'utilisateur** : L'expérience utilisateur (UX) est importante même pour un petit projet. Ajouter des messages clairs permet aux utilisateurs de comprendre ce qui se passe dans le programme.

      Exemple :

      ```
      def emprunter_livre(self, livre):
          if livre.est_emprunte:
              print(f"Le livre '{livre.titre}' est déjà emprunté.")
          else:
              livre.est_emprunte = True
              print(f"Vous avez emprunté '{livre.titre}'.")
      ```

    - **Interface utilisateur basique** : Si possible, les élèves peuvent ajouter une **interface utilisateur simple** comme des menus en console pour naviguer dans les options.
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 une gestion de bibliothèque, ils doivent tester les fonctionnalités comme l’emprunt et le retour de livres, la gestion des dates de retour, etc.
2. **Vérifier les cas extrêmes et les erreurs possibles :**
    - Tester ce qui se passe si un joueur essaie d'attaquer un ennemi déjà mort, ou si un abonné tente d'emprunter un livre déjà emprunté.
    - Tester les **entrées utilisateurs incorrectes** pour s’assurer que le programme ne plante pas.

    Exemple :

    ```
    def rendre_livre(self, livre):
        if not livre.est_emprunte:
            print(f"Le livre '{livre.titre}' n'a pas été emprunté.")
        else:
            livre.est_emprunte = False
            print(f"Vous avez retourné '{livre.titre}'.")
    ```

3. **Validation des performances :**
    - Les élèves doivent vérifier que leur programme fonctionne bien même lorsque le nombre de livres ou 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 (gestion de bibliothèque).
    - Expliquer l’objectif du projet et ce qu’il permet de faire.
2. **Concepts de POO utilisés :**
    - Présenter les **classes principales** créées (par exemple, `Livre`, `Bibliothécaire`, etc.).
    - Expliquer l’utilisation de l'**héritage** (si applicable), des **méthodes**, des **attributs**, et la manière dont les objets interagissent entre eux.

    Exemple :

    - Dans une bibliothèque, les livres, abonnés et bibliothécaires interagissent via des méthodes spécifiques.
3. **Fonctionnalités principales :**
    - Décrire les fonctionnalités principales implémentées, comme la gestion des emprunts dans la bibliothèque.
    - Montrer des extraits de code qui illustrent des parties clés du projet.
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 système de **gestion de bibliothèque**.

  Exemple pour la gestion d'une bibliothèque :

  - "Nous avons choisi de développer une application de gestion de bibliothèque. Elle permet de gérer les emprunts de livres par les abonnés, avec des fonctionnalités comme l'ajout de nouveaux livres, la gestion des retours, et le calcul des pénalités en cas de retard."

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 une gestion de bibliothèque, ils peuvent simuler un utilisateur qui emprunte un livre ou rend un livre.
2. **Montrer les interactions clés** :
   - **Pour une gestion de bibliothèque** : Montrer comment un abonné peut emprunter un livre, comment les pénalités sont calculées, ou comment un bibliothécaire gère les prêts.
3. **Explorer plusieurs cas d'utilisation** :
   - Si possible, le groupe devrait montrer plusieurs fonctionnalités, par exemple : ce qui se passe comment les retards de livres sont gérés dans la bibliothèque.

   Exemple pour une bibliothèque :

   - "Nous allons maintenant simuler l'emprunt d'un livre déjà emprunté pour montrer que notre programme gère les erreurs."

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 la gestion de la bibliothèque** : Gestion des dates d'emprunt et des pénalités, ou éviter les emprunts en double.
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 :
    - "Que se passe-t-il si un abonné ne retourne jamais son livre ?"
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 :

- "Pour le système de bibliothèque, vous pourriez ajouter un moyen de notifier les abonnés lorsque la date limite de retour approche."

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 `Livre` 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 fonctionnel (gestion de bibliothèque).