# Informatique 2: Travaux Pratiques

*Programmation Orientée Objet 2/3 – Opérateurs de comparaison et arithmétiques ; Méthodes et attributs de classes*

Le but de cette séance est de consolider les connaissances sur les classes et les objets en général et de se familiariser avec les méthodes et les attributs de classe (notamment cerner les différences avec méthodes et attributs d'instance).
**Procéder par étape et tester votre code (dans un bloc main) pour chaque étape.** Un dossier contenant des fichiers de test vous est fourni sur Moodle pour que vous puissiez tester votre code.
*Temps mentionné (🕓) à titre strictement indicatif.*

## Exercices de base

### Fraction

Cet exercice se base sur la classe **Fraction** commencée lors de la séance précédente.

**Question 1:** (🕓 *30 minutes*) L'objectif de cet exercice est d'utiliser la surcharge des opérateurs pour l'égalité et la comparaison d'objets **Fraction**.

1. Ecrire la méthode d'instance `__eq__` qui prend en argument, en plus de `self`, un objet **Fraction** (nommé *autre*) et qui renvoie un booléen indiquant si l'instance considérée est égale à l'argument. Tester en utilisant l'opérateur ==.

   **>_ Exemple**

   **Fraction(1,2) == Fraction(2,4)** devrait renvoyer `True`.

2. Vérifier que le paramètre *autre* est bien une fraction (utiliser `isinstance`) ; lever une exception dans le cas contraire.

   **>_ Exemple**

   **Fraction(4,2) == 2** devrait lever une exception.

3. Modifier la méthode précédente pour prendre en compte les entiers. Faire de même pour le cas où *autre* est de type `float`, mais pouvant être converti en entier

   **>_ Exemple**

   **Fraction(4,2) == 2** devrait maintenant renvoyer `True`.

4. Définir les autres opérateurs de comparaison. Pour ce faire, réutiliser les opérateurs déjà implémentés (par exemple, pour `__ne__` on réutilisera `__eq__`). Tester avec les opérateurs de comparaison ($<$, $>$, *etc.*).
5. Définir les opérateurs arithmétiques correspondant à + (`__add__` et `__radd__`), -, * et **.
6. Créer une liste de fractions et d'entiers et appeler les fonctions `min`, `max`, `sum` et `sorted` pour tester.

   **>_ Exemple**

   ```
   l = [Fraction(1,2), Fraction(−2,3), 1]
   print(sorted(l), min(l), max(l), sum(l))
   ```

**Question 2:** (🕓 *(10 minutes)*) L'objectif de cet exercice est d'ajouter des méthodes de classe pour gérer la liste des instances créées.

1. Définir un attribut de classe privé **__list_instances** pour mémoriser la liste des fractions créées. Modifier le constructeur de la classe pour maintenir cette liste à jour.

> **Attention**
>
> Penser à initialiser l'attribut **__list_instances**

2. Définir une méthode de classe **get_list_instances** pour accéder à cette liste. Renvoyer une copie pour éviter les modifications par l'utilisateur !
3. Définir une méthode de classe **get_nb_instances** pour accéder à la taille de cette liste, c'est-à-dire le nombre total d'instances de **Fraction** créées.

### Supermarché

**Question 3:** (*(25 minutes)*) On veut gérer les produits d'un supermarché ; en particulier on veut appliquer des réductions à certains produits et appliquer des réductions générales à tous les produits.

1. Définir une classe **Produit**. Un **Produit** est caractérisé par un nom (*str*), un prix (*float*, positif), et une réduction (*int*, en % et donc compris entre 0 et 100, initialisé à 0). On veut stocker ces valeurs dans des attributs privés. Créer le constructeur **__init__** en levant une exception appropriée pour les problèmes de type et de valeur.
2. On va effectuer des vérifications de valeur sur la réduction à plusieurs endroits dans le code de la classe. On veut donc définir une méthode pour lever une exception si la valeur est incorrecte. Quel type de méthode est le plus approprié ? Une méthode d'instance, de classe ou statique ?

```
def verification_reduction(reduction):
    if reduction > 100 or reduction < 0:
        raise ValueError("Les réductions doivent être comprises entre 0 et 100 (\%)")}

```

3. Définir un setter **set_reduction** pour fixer la valeur de la réduction. Utiliser la vérification précédente dans le constructeur et dans le setter.
4. Définir une méthode d'instance **get_prix_initial** qui renvoie le prix initial (i.e. celui stocké dans l'attribut privé *prix*) du produit.
5. On veut pouvoir appliquer une réduction générale à tous les produits. On veut donc définir un attribut **__reduction_generale** , initialisé à 0, contenant cette valeur et une méthode **set_reduction_generale**, permettant de la modifier. Quel type d'attribut et de méthode sont les plus appropriés ? D'instance ou de classe ? Penser à utiliser la méthode **verification_reduction** pour vérifier la validité de la valeur passé en argument.
6. Définir une méthode d'instance **get_prix_courant** qui renvoie le prix réel (i.e. après application du discount général et du discount individuel).

> **>_Exemple**
>
> Pour un produit ayant un prix initial de **10 CHF**, en appliquant une réduction générale de 20% sur l'ensemble de produits du magasin et une réduction de 50% sur le produit, on obtiendrait un prix final de **4 CHF**

7. Définir la méthode **__str__** pour représenter un produit sous la forme textuelle suivante

```
[Product] Saumon (10.0 CHF)
[Product] Pain (1.5 CHF, valait 3.0 CHF)
```

Utiliser les méthodes précédemment définies pour obtenir les prix originaux et réduits.

### LocationSample projet

**Question 4:** (🕒 *(10 minutes)*) Définir la classe **LocationSample**. Vérifier les valeurs passées en paramètre et lever une exception en cas de problème. Un **LocationSample** contient une *date* (utiliser un entier pour créer un nouveau timestamp, par exemple un nombre de secondes écoulées depuis une certaine date de référence ; à terme, utiliser le module **datetime.datetime**) ainsi qu'une **Location** (cf la classe **Location** du TP précédent).

**Question 5:** (🕒 *(5 minutes)*) Définir la méthode **__str__** de sorte à afficher un objet **LocationSample** sous la forme suivante :
**"LocationSample [timestamp: 1254, location: Location [latitude: 48.85479, longitude: 2.34756]]"**

**Question 6:** (🕒 *(5 minutes)*) Implémenter les *getters*.

**Question 7:** (🕒 *(10 minutes)*) Redéfinir les méthodes de comparaisons suivant l'ordre chronologique entre deux objets **LocationSample**.

**>_ Exemple**

**ls1** $<$ **ls2** si et seulement si **ls1.date** $<$ **ls2.date**

**Question 8:** (🕒 *(5 minutes)*) Implémenter la méthode **distance_spatiale** qui prend en argument un objet **Location** et renvoie la distance géographique (le nombre de mètres qui les sépare) entre l'objet **LocationSample** et l'objet **Location** passé en paramètre.