Notion de classe

- La **classe** définit le **type** de l'objet (son moule)

- Tout objet n'est que **l'instance** d'une classe
- Tous les objets d'une classe sont sur le même modèle *(même type)*
- La classe décrit les « **attributs** » et les « **méthodes** » de chacun de ses objets

- Ces différents composants sont encapsulés et leur accès est protégé.

Découvrir les principes de la programmation objets

- 4 grands principes :

1. Encapsulation

   (avec : 1bis. Généricité)

2. Héritage
3. Polymorphisme
4. Liaison dynamique

(1): le principe d'encapsulation

- Protection des attributs et des méthodes

- L'accès aux données (internes) ne peut être fait qu'au travers de méthodes *(mécanisme de protection)*
  - Les données sont privées (cachées)
  - Les méthodes publiques définissent l'interface de l'objet

- Intérêt :
  - Séparer le quoi du comment
  - Facilité d'évolution
  - Pas de modification anarchique

(1) Le principe d'encapsulation (suite)

- Les objets interagissent et communiquent entre eux par l'envoi de **messages**

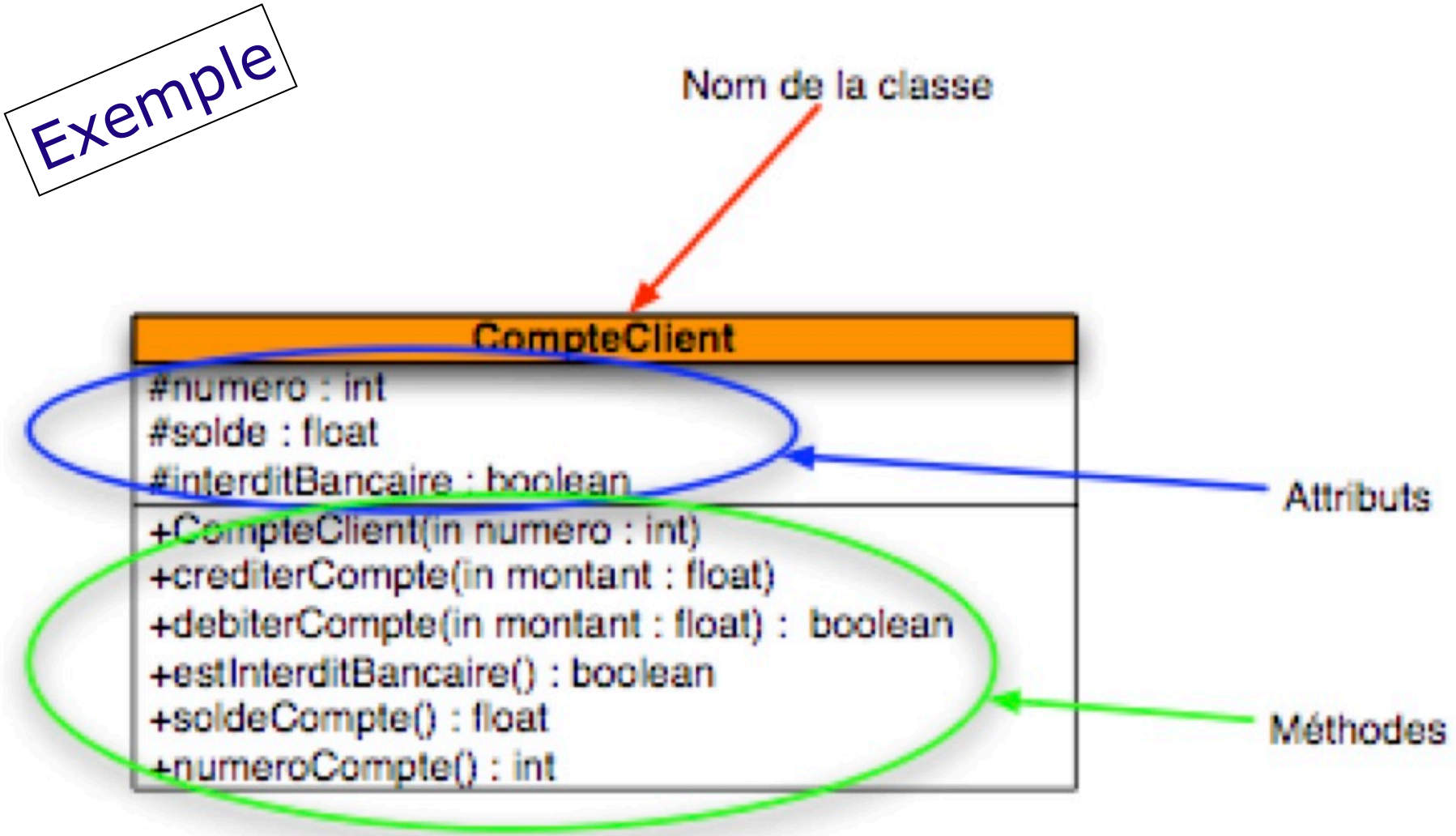
- Méthodes publiques = messages acceptables

  il existe aussi des méthodes privées

- Messages d'un objet vers un autre, caractérisés par :
  - Objet cible (récepteur du message, et qui effectuera la méthode)
  - Nom de la méthode
  - Paramètres typés de la méthode
  - Résultat typé (ou void)

Notion de protection/visibilité
- Public : accessible depuis tous les autres objets
- Privé/Protégé : accessible depuis l'objet, ou depuis un nombre restreint d'objets

Autre exemple : l'interface d'une classe CompteClient
dans une application (très simplifiée) de gestion de comptes bancaires



Travail pratique n°2:
un autre petit programme en java

```
public class Point {
    protected int x, y;
    public Point(int abs, int ord) {
            ❖ … // à compléter pour créer un objet point
    }
    public Point(Point p) {
            ❖ … // à compléter pour créer en copiant un autre objet point
    }
    public int getX() {
            ❖ … // à compléter }
    public int getY() {
            ❖ … // à compléter }
    public void afficher() {System.out.println("abs = "+x+", ord = "+y)}
    … // à compléter
}
```

Travail demandé :

- créer le fichier **`Point.java`** pour représenter la classe des : `Point`
- créer le fichier **`Test.java`** (contenant un programme qui teste la classe des `Point`)

- compiler et exécuter votre programme de Test (`javac Test.java` et `java Test`)

- *ajouter une méthode pour **translater** un point*
- *modifier `Test.java` pour **tester la translation.***

Travail pratique n°3:
combinaison d'objets en Java

```
public class Point {
    protected int x, y;
    public Point(int abs, int ord) {
        . . . // à compléter pour créer un objet point
    }
    public Point(Point p) {
        . . . // à compléter pour créer en copiant un autre objet point
    }
    public int getX() {
        . . . // à compléter }
    public int getY() {
        . . . // à compléter }
    public void afficher() {System.out.println("abs = "+x+", ord = "+y)}
        . . . // à compléter
}
```

Travail demandé :

- en s'inspirant de la classe **`Point`**, créer le fichier **`Segment.java`** (décrivant la classe des segments définis à partir de leur deux extrémités)

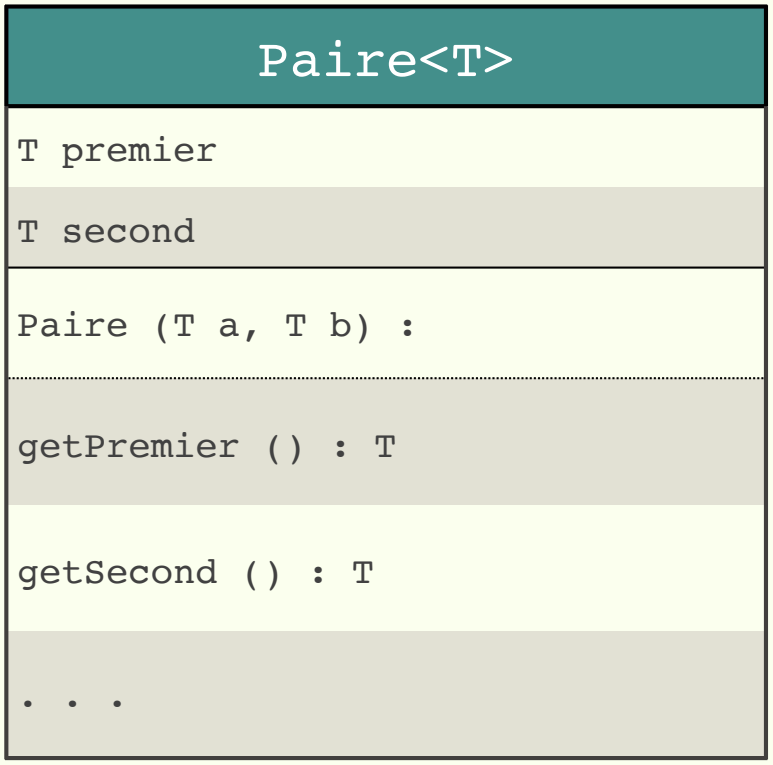
- *définir un **constructeur** de segment à partir de ses deux extrémités passées en paramètres de la méthod*
- *définir un constructeur de segment qui est la copie d'un autre segment passé en paramètre*
- *ajouter une méthode pour **translater** le segment*
- *ajouter une méthode pour **afficher** les coordonnées des extrémités du segment*

(1bis): Généricité

- Le type des **attributs**, des **méthodes** (de leur résultat ou de leurs paramètres) peut ne pas être connu lorsqu'on crée la classe
- Il existe des « paramètres de type »
-

Exemple

Une paire d'objets d'un type quelconque mais instancié à l'exécution



- *Ce n'est pas une spécificité des L.O. mais c'est particulièrement adapté.*
- De même, on peut avoir de la généricité contrainte.