# La programmation orientée objet (POO)


## 1) Notion d'objet :

### 1-1) Avantages de la POO :

Historiquement, la solution aux problèmes logiciels, consistait à diviser une tâche complexe en plusieurs sous-taches ***(procédures ou fonctions)*** et à diviser ces sous-tâches en d'autres sous-tâches encore plus petites ***(d'autres procédures ou fonctions)***.

C'est une **approche procédurale**. Nous avons d'une part, les structures de données et d'autre part les traitements associés à ces données. ***Nous obtenons ainsi un code structuré mais pas très modulaire.***

Cette approche présente certains inconvénients. Si nous désirons faire évoluer le logiciel ou si nous voulons en réutiliser une partie, nous nous apercevons que les fonctions sont liées à des données partagées avec d'autres fonctions. Il est donc difficile d'isoler une partie du programme ou de le faire évoluer. ***Dans ce cas de figure, lorsqu'un programme devient important, il peut s'avérer difficile de s'y retrouver.***

La **POO** ***(programmation Orienté Objet)*** permet de remédier à ce problème en offrant une programmation modulaire extrêmement structurée, très lisible et surtout réutilisable.

### 1-2) Constitution d'un objet : attributs et méthodes

En informatique, un **objet** est une entité (structure, module, ...) qui regroupe :

- ✗ Des données (que l'on appelle **attributs** ou ***données membres***). Ces attributs sont des variables caractérisant **l'état** de l'objet.
- ✗ Des fonctions (que l'on appelle **méthodes** ou ***fonctions membres***). Les méthodes définissent le **comportement** de l'objet. Elles correspondent à l'ensemble des actions que l'objet est à même de réaliser. Ces opérations permettent de faire réagir l'objet aux sollicitations extérieures (ou d'agir sur les autres objets). Leurs actions peuvent également dépendre des valeurs des attributs, ou bien a pour objectif de les modifier (évolution de l'état de l'objet).

L'objet correspond donc à un nouveau type de variable qui ressemble un peu à une variable de type structure. Comme celle-ci, il est composé de variables de différents types (que nous appelons **attributs**), mais il est aussi composé de fonctions (que nous appelons **méthodes**). Ce sont ces méthodes qui vont « ***donner vie et faire évoluer l'objet*** ».

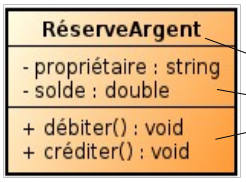
Un objet peut être vu comme une brique autonome avec laquelle nous construisons notre programme.

### 1-3) Description d'un objet : la classe

Comme pour la structure il faut définir quels seront les attributs et les méthodes qui vont composer l'objet. C'est le rôle de la **classe** qui peut être considérée comme le plan ou la notice contenant tous les éléments pour créer un objet.

- ✗ Un objet est donc élaboré à partir d'une classe. On dit qu'il est **l'instance** d'une classe.
- ✗ Tous les objets instanciés à partir d'une même classe sont de la même famille et ont le même type, ce qui veut dire qu'ils ont le même comportement.
- ✗ Par contre, chaque objet instancié est unique et possède sa propre identité.

**Exemple :**
Soit la classe **RéserveArgent** : Un objet de classe **RéserveArgent** doit appartenir à une personne (un propriétaire) et peut contenir une certaine somme d'argent (un solde). Nous pouvons y mettre ou enlever de l'argent (la créditer ou la débiter). Nous en déduisons les **attributs** et les **méthodes** suivants :




En notation UML, une classe est représentée par un rectangle de 3 compartiments :
- 1ier compartiment : nom de la classe.
- 2ième compartiment : liste des attributs.
- 3ième compartiment : liste des méthodes.

À partir de la classe **RéserveArgent**, nous pouvons créer autant de d'objets de la classe **RéserveArgent** que nous voulons. Ils auront tous leur propre identité et évolueront indépendamment. Nous ne définissons que les attributs et les méthodes dont nous avons besoin. Ici la classe **RéserveArgent** est sommaire, nous aurions très bien pu donner d'autres précisions en ajoutant d'autres attributs et méthodes comme le montant maximum qu'elle peut contenir par exemple.

## 1-4) L'encapsulation :

Un objet est constitué d'attributs qui caractérisent son **état**. Ces attributs ne pourront être modifiés ou lus uniquement en utilisant les méthodes de l'objet. C'est **l'encapsulation**.

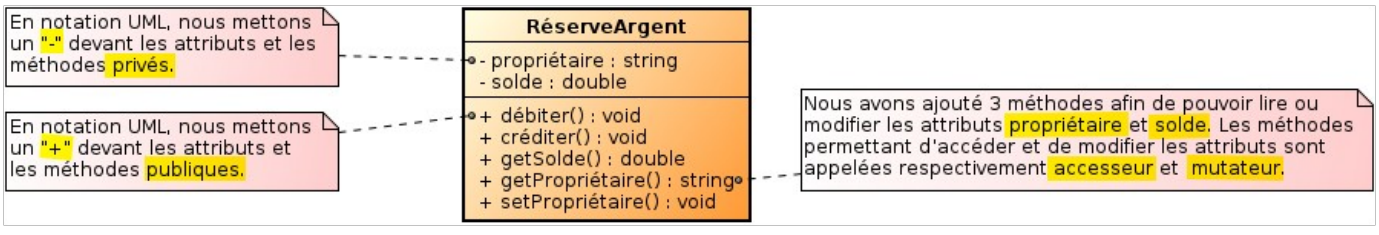
***L'encapsulation consiste donc à masquer les détails d'implémentation d'un objet, en définissant une interface.***

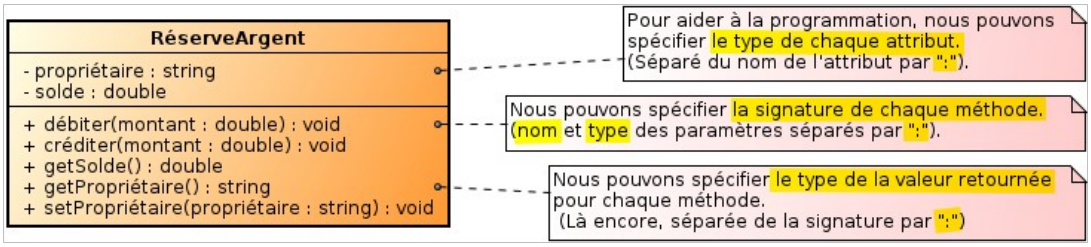
- L'interface est la vue externe d'un objet, elle définit les services accessibles (offerts) aux utilisateurs de l'objet.
- L'encapsulation facilite l'évolution d'une application car elle stabilise l'utilisation des objets : nous pouvons ainsi modifier l'implémentation des attributs d'un objet sans modifier son interface, et donc la façon dont l'objet est utilisé.
- L'encapsulation garantit l'intégrité des données, car elle permet d'interdire, ou de restreindre, l'accès direct aux attributs des objets.

Un objet est donc constitué :
- **D'une partie privée** qui ne peut pas être directement accessible de l'extérieur de l'objet. La partie privée contient généralement les attributs de l'objet.
- **D'une partie publique** qui est accessible de l'extérieur de l'objet. La partie publique contient généralement les méthodes de l'objet (méthodes qui par contre permettent d'accéder aux attributs).

**Exemple :**
La classe **RéserveArgent** vue précédemment contient :
- 2 **attributs** (***propriétaire*** et ***solde***)
- 2 **méthodes** (***débiter()*** et ***créditer()***)

De l'extérieur de l'objet, nous ne pouvons pas accéder aux attributs ***propriétaire*** et ***solde***. Cela peut être volontaire mais si nous désirons quand même qu'un attribut puisse être lu, nous devons implémenter une nouvelle méthode. Dans notre cas, si nous désirons lire les 2 attributs, il faut donc rajouter 2 méthodes que nous pouvons appeler ***getProriétaire()*** et ***getSolde()***.



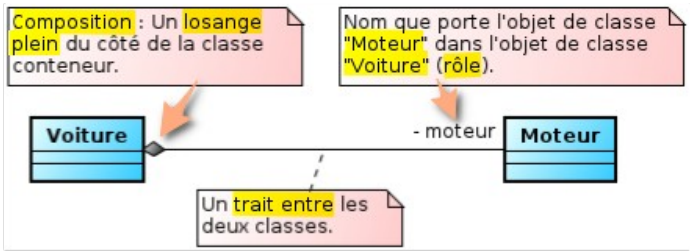



# 2) Les relations entres classes :

## 3-1 ) La composition :

Si **un objet A contient un objet B**, c'est-à-dire que **l'objet B est un attribut d'un objet A**, nous précisons alors que A et B sont reliés par une relation de composition. ***C'est une relation très forte, si l'objet A est détruit, alors l'objet B est aussi détruit***.

**Exemple 1** : Si nous définissons la classe **Voiture** et la classe **Moteur**, la classe **Voiture** contient un objet de classe **Moteur**. En notation UML, la composition se représente comme indiqué ci-contre :



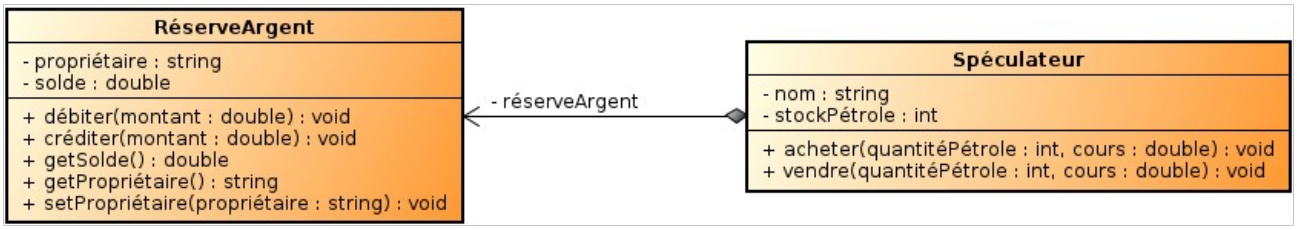

**Exemple 2 :** Soit la classe **Spéculateur**. Cette classe possède :

✗ **3 attributs :**
- **nom** qui est le nom du spéculateur (type **string**)
- **stockPétrole** qui représente une quantité de pétrole en barils (type **int**).
- **réserveArgent** qui est un objet de la classe **RéserveArgent** vue précédemment.

✗ **2 méthodes :** ***acheter()*** et ***vendre()***.

Les deux méthodes ont la même signature, elles prennent deux arguments qui sont :
- La quantité de pétrole vendu ou acheté : **quantitéPétrole** (de type **int**).
- Le prix d'un baril de pétrole en dollars : **cours** (de type **double**).

***Les 2 méthodes mettent à jours les attributs stockPétrole et solde (de réserveArgent).***

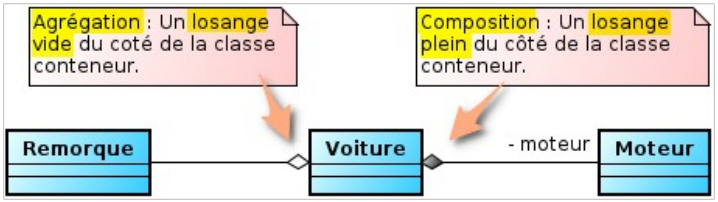



## 3-2) L'agrégation :

L'agrégation, c'est lorsqu'**un objet A possède et utilise un objet B** (qui est **extérieur** à l'objet A). Contrairement à la composition, où l'objet inclus disparaît si l'objet englobant est détruit, dans une agrégation cet objet ne disparaît pas lorsque l'objet englobant est détruit. De plus, dans le cas d'une composition, seul l'objet A utilisait B (dans une agrégation, l'objet B peut très bien être utilisé par un autre objet).

***La manière habituelle d'implémenter l'agrégation en C++ se fait au travers de pointeurs ou de références.***

**Exemple 1** : Si nous définissons la classe **Voiture** et la classe **Remorque**, la classe **Voiture** utilise un objet de classe **Remorque**. En notation UML, l'agrégation se représente comme indiqué ci-contre :



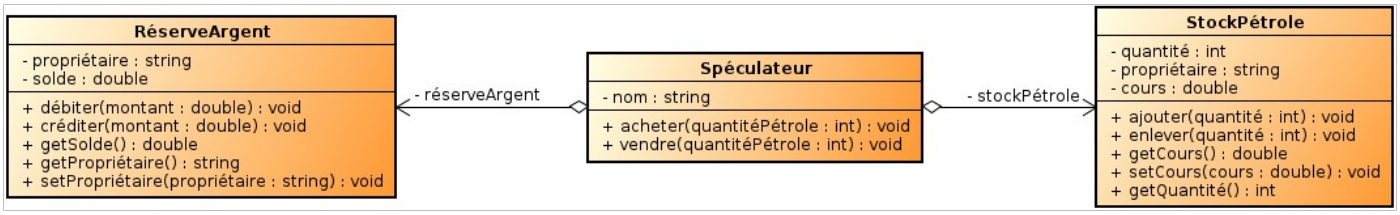

**Exemple 2 :** Reprenons l'exemple de la classe **Spéculateur**.
Si nous regardons bien, il est plus judicieux de considérer que la réserve d'argent n'est pas une composition de spéculateur. En effet si le spéculateur disparaît, la réserve d'argent ne disparaît pas avec lui. **La réserve d'argent est finalement un objet extérieur qui appartient au spéculateur, c'est donc une agrégation**.

De plus, soit le nouvel objet **stockPétrole** définit par les attributs et méthodes ci-dessous :

✗ **attributs :**
- **propriétaire** : nom du propriétaire du stock.
- **quantité** : la quantité de pétrole en stock.
- **cours** : le cours du baril de pétrole.

✗ **méthodes :**
- ***ajouter()*** qui permet d'ajouter des barils de pétrole au stock.
- ***enlever()*** qui permet d'enlever des barils de pétrole au stock.
- ***setCours()*** **mutateur** qui permet de mettre à jour le cours du baril de pétrole.
- ***getCours()*** **accesseur** qui permet de savoir quel est le cours du baril de pétrole.
- ***getQuantite()*** **accesseur** qui permet de savoir quel est la quantité de pétrole en stock.

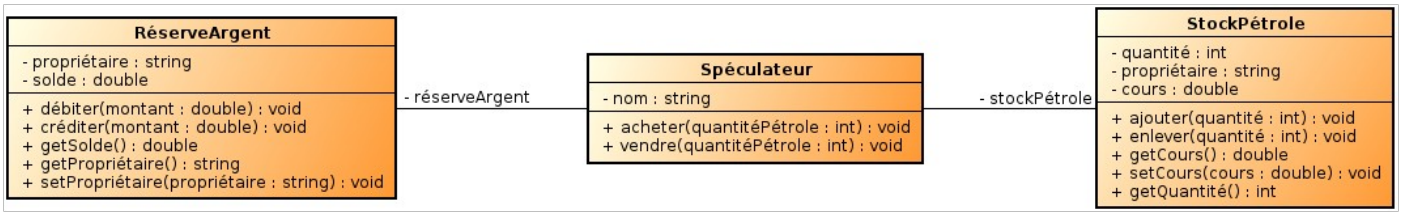
**stockPétrole** est aussi un objet qui appartient à la classe **Spéculateur**.
Nous ajoutons l'attribut **nom** de type **string** à la classe **Spéculateur**.




## 3-3) Associations simples :

Une association est une relation qui indique une connexion structurelle entre différentes classes. La composition et l'agrégation sont des cas particuliers d'associations, mais pour une association simple, nous n'avons plus la notion hiérarchique de constitution ou de possession d'une classe par rapport à l'autre. Les classes sont associées d'égale à égale. Une association simple se représente par un trait entre les deux classes.

**Exemple :**
Dans l'exemple précédent, si le spéculateur travaille sur des réserves d'argent et des stocks de pétrole qui ne lui appartiennent pas, s'il effectue des transactions pour d'autres personnes, alors nous sommes en présence d'une association simple.



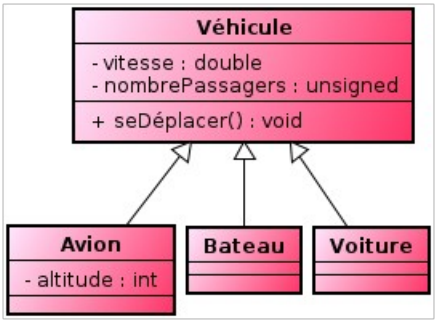

**Remarques :**
- ✔ L'implémentation est la même que pour une agrégation.
- ✔ Si le spéculateur doit gérer les réserves d'argent et les stocks de pétrole de plusieurs personnes, nous pouvons lui ajouter les méthodes ci-dessous qui permettent de définir sur quelle réserve d'argent et sur quel stock de pétrole se feront les transactions :
  - ✗ void ***setReserve(ReserveArgent *reserveArgent)***
  - ✗ void ***setStock(StockPetrole *stockPetrole)***

## 3-4) L'héritage :

**L'héritage** permet de créer une nouvelle classe à partir d'une classe existante. Le nom d'héritage provient du fait que la classe dérivée ***(classe fille)*** contient les attributs et les méthodes de la classe dont elle est issue ***(classe mère)***. L'intérêt majeur de l'héritage est de ne pas avoir à repartir de zéro lorsque nous voulons spécialiser une classe existante. Les attributs et méthodes de la classe mère vont s'ajouter à ceux définis dans la classe fille. Cela nous permet d'éviter à devoir réécrire un même code source plusieurs fois. L'héritage évite ainsi la duplication et favorise la réutilisation.

**Exemple 1 :** Soit la classe **Véhicule** qui contient la méthode ***seDéplacer()*** et les attributs **vitesse** et **nombrePassagers**. Nous pouvons créer ensuite les classes **Voiture**, **Avion**, et **Bateau** qui sont des spécialisations de la classe **Véhicule**. Chacune des classes spécialisées possède ses propres attributs et méthodes auxquelles viennent s'ajouter ceux hérités. Ainsi par exemple, pour la classe **Avion,** nous pouvons définir l'attribut **altitude** qui lui sera propre mais elle bénéficie également de la méthode ***seDéplacer()*** ainsi que les attributs **vitesse** et **nombrePassagers** sans avoir à les redéfinir.

***En UML, l'héritage est représenté par une flèche (triangle vide).***



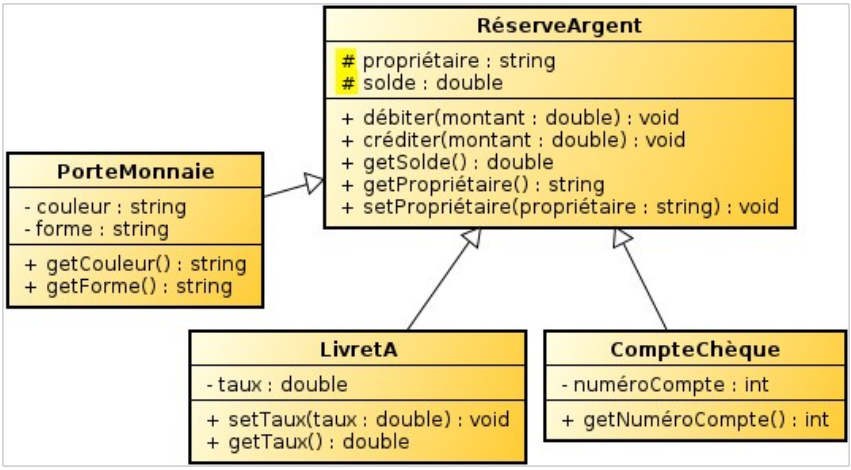

## Exemple 2 : RéserveArgent

Nous avons vu que les attributs et les méthodes peuvent avoir une portée :

- **public** (accessible de l'extérieur de la classe et symbolisés par un **'+'**).
- **private** (non accessible à l'extérieur de la classe est symbolisés par un **'-'**).

***Ils peuvent aussi avoir une portée protected c'est-à-dire qu'ils seront accessibles à partie des classes filles (héritées).***

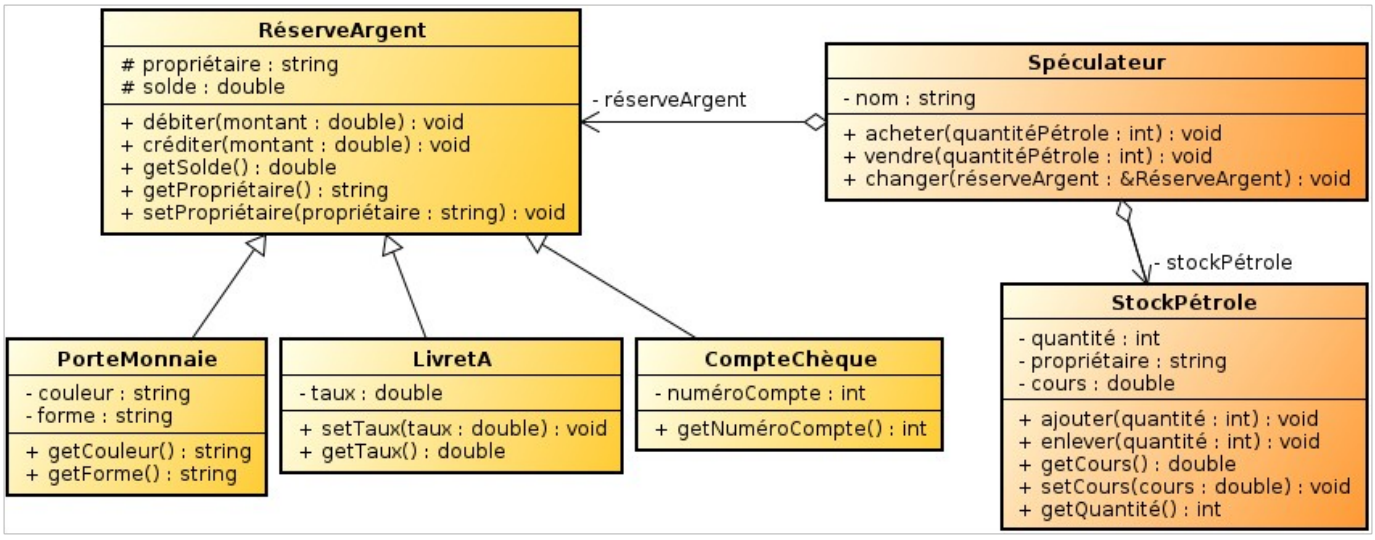
***La portée protected est symbolisée par un '#' devant l'attribut ou la méthode.***




Les classes **CompteChèque**, **LivretA** et **PorteMonnaie** sont bien des spécialisations de **RéserveArgent**. Elles peuvent donc hériter des attributs et des méthodes de la classe **RéserveArgent** (attributs et méthodes qui n'ont pas besoin d'être redéfinis). Ainsi, la classe **LivretA** dispose de :

- **3 attributs** (**taux** qui lui est propre puis **propriétaire** et **solde** qu'elle hérite de **RéserveArgent**).
- **6 méthodes** (***setTaux()*** et ***getTaux()*** qui lui sont propre, puis ***débiter(), créditer(), getSolde()*** et ***getProprietaire()*** qu'elle hérite de **RéserveArgent**.

Nous allons aussi modifier la classe **Spéculateur** en ajoutant la méthode ***changer()*** qui permet de changer la réserve d'argent sur laquelle nous effectuons des opérations.

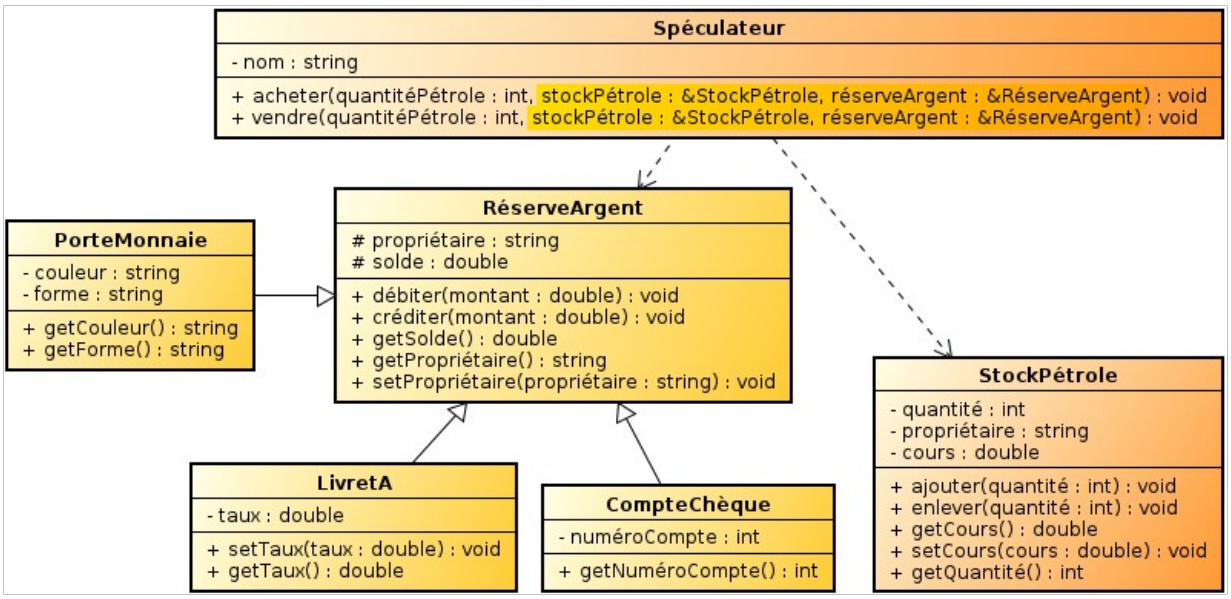



## 3-5) Les relations de dépendances :

Une association ***(association simple, agrégation ou composition)*** entre un objet A et un objet B est une relation forte. Cela influence directement la structure des classes A et B en se traduisant par la présence dans les attributs de A d'une variable supplémentaire correspondant à B (et réciproquement si nécessaire).

La relation de dépendance est beaucoup moins forte. C'est une relation unidirectionnelle qui indique qu'une classe A utilise une classe B mais sans incidence sur la structure de la classe A ***(pas d'attributs correspondant à B parmi les attributs de A)***. Par contre, nous pouvons retrouver ***des objets de classe B dans les signatures des méthodes de la classe A***.

**Exemple :**
Nous pouvons réaliser le programme de spéculation précédent en remplaçant les agrégations par des relations de dépendances. Comme dans **Spéculateur**, il n'y a plus d'attributs correspondant à une réserve d'argent et à un stock de pétrole. Il est alors nécessaire de modifier les méthodes ***vendre()*** et ***acheter()*** en leur passant en paramètres des références de la réserve d'argent et du stock de pétrole concernés par la transaction.




## 5) Le polymorphisme :

Le mot polymorphie vient du grec et signifie « qui peut prendre plusieurs formes ». Le polymorphisme est le fait qu'une méthode puisse accepter des paramètres de types différents (de différentes formes) et que le comportement de la méthode soit différent en fonction du type de paramètre reçu. Nous distinguons deux types de polymorphisme, la ***surcharge*** et la ***redéfinition***.

- **Polymorphisme de surcharge :** Des méthodes qui possèdent des noms identiques mais une signature différente. Exemple : Lorsque nous possédons plusieurs constructeurs dans une classe, ils portent tous le même nom mais avec des signatures différentes.
- **Polymorphisme de redéfinition :** Si une méthode définie dans une classe mère n'est pas adaptée à une classe fille, nous pouvons la redéfinir dans la classe fille, et c'est la méthode redéfinie qui sera alors appelée. Ici, non seulement, les méthodes possèdent le même nom mais aussi la même signature.

**Exemple :**
Imaginons un jeu d'échec composé des objets représentés par les classe suivantes : **Tour**, **Cavalier**, **Fou**, **Roi, Dame** et **Pion** chacune héritant de la classe **PièceÉchec**.

Dans chaque classe fille, nous redéfinissons la méthode ***seDéplacer()*** correspondant au mode de déplacement propre de la classe fille.

La méthode ***seDéplacer()*** pourra donc s'appliquer a des objets de types différents et aura un comportement différent en fonction du type d'objet auquel elle s'applique.