

PREMIERS ÉLÉMENTS DE CONCEPTION ET PROGRAMMATION ORIENTÉE À OBJETS

Classes, objets, attributs, méthodes

Modélisation et Programmation par objets 1 – cours 1 et 2

1 Modélisation et programmation orientée objets

Les approches par objets sont fondées sur quelques idées simples qui consistent à décrire un système avec des représentations informatiques proches des entités du problème et de sa solution : si on parle de personnages de jeux, on décrira des objets *Personnages de jeu* dans le langage informatique. Les approches par objets offrent plusieurs avantages :

- facilité du codage initial,
- stabilité du logiciel construit car les structures des objets manipulés sont plus stables que les fonctionnalités attendues,
- réutilisation et maintenance facilitées.

Ce cours présente les concepts essentiels de l'approche objet en s'appuyant sur un langage de modélisation (UML) et un langage de programmation (Java).

1.1 Modélisation orientée objets

La modélisation est la première activité d'un informaticien face à un système à mettre en place.

Pourquoi modéliser ? Modéliser consiste à produire une représentation simplifiée du monde réel pour :

- accumuler et organiser des connaissances,
- décrire un problème,
- trouver et exprimer une solution,
- raisonner, calculer.

Il s'agit en particulier de résoudre le hiatus entre d'un côté le monde réel, complexe, en constante évolution, décrit de manière informelle et souvent ambiguë par les experts d'un domaine, et de l'autre le monde informatique, où les langages sont codifiés de manière stricte et disposent d'une sémantique unique.

Les points difficiles de la modélisation La modélisation est une tâche rendue difficile par différents aspects :

- les spécifications parfois imprécises, incomplètes, ou incohérentes,
- taille et complexité des systèmes importantes et croissantes,
- évolution des besoins des utilisateurs,
- évolution de l'environnement technique (matériel et logiciel),
- des équipes à gérer plus grandes.

Méthodes de modélisation Pour faire face à ces difficultés, les méthodes d'analyse et de conception proposent des guides structurant :

- organisation du travail en différentes phases (analyse, conception, codage, etc.) ou en niveaux d'abstraction (conceptuel, logique, physique),
- concepts fondateurs : par exemple les concepts de fonction, signal, état, objet, classe, etc.,
- représentations semi-formelles, documents, diagrammes, etc.

UML Le langage de modélisation est un formalisme de représentation qui facilite la communication, l'organisation et la vérification.

Nous utiliserons dans ce cours UML (Unified Modeling Language) qui est un langage de modélisation graphique véhiculant en particulier les concepts des approches par objets : classe, instance, classification, etc, en y intégrant beaucoup d'autres aspects : associations, fonctionnalités, événements, états, séquences, etc.

Un même système est décrit par plusieurs modèles, qui peuvent être de différentes natures : ce sont des vues sur le système. On peut par exemple s'intéresser à la structure du système au grain des classes et de leurs interactions, ou bien à un grain plus gros pour s'intéresser au déploiement de l'application, ou encore décrire finement un algorithme, décrire les interactions du système avec ses utilisateurs, etc. Bien sûr, les différentes vues doivent être cohérentes les unes avec les autres.

Chaque modèle est une représentation abstraite d'une réalité, il fournit une image simplifiée du monde réel selon un point de vue.

En UML, les modèles sont représentés graphiquement dans des diagrammes de différents types : diagrammes de classes, d'objets, d'interaction, etc. Nous nous limiterons ici aux diagrammes de classes et d'objets.

1.2 Programmation orientée objets

Classification d'objets La programmation orientée objet organise le code en le faisant reposer sur le concept d'objet. Un objet est une entité qui est décrite par un certain nombre de caractéristiques, et d'opérations permettant de manipuler ces caractéristiques. Les objets sont classifiés, à la manière du raisonnement classificatoire que l'on met en œuvre quand l'on organise des données dans la vie courante : on regroupe dans une même classe les éléments qui ont les mêmes caractéristiques.

Les langages orientés objets ne sont pas tous à classes mais nous nous cantonnerons ici à ceux-ci.

Les classes L'organisation du code est alors basée sur les classes d'objets : tout ce qui concerne une même classe d'objets est au même endroit. On bénéficie également d'une bonne réutilisation (les classes sont faciles à réutiliser dans d'autres contextes) et on peut spécialiser des classes d'objets en leur rajoutant des caractéristiques.

Java On utilisera dans cette UE le langage Java. Java a été créé en 1991.

- Il emprunte une grande partie de sa syntaxe à C++ ;
- il recherchait à l'origine une plus grande simplicité que C++ ;
- il permet de s'abstraire des problèmes de gestion de la mémoire ;
- il n'est pas « tout objet » et n'a pas les capacités de réflexivité des langages à objets les plus avancés, mais en a cependant plus que C++ ;
- il fonctionne à l'aide de deux programmes, un compilateur et un interprète, et ce qui a fait en partie son succès est la possibilité d'avoir cet interprète dans tous les navigateurs internet (ce n'est plus le cas actuellement).

2 Classes et objets

2.1 Classes, objets, et attributs

Les approches orientées objet organisent la structure du système autour des classes : chaque **classe** décrit les caractéristiques que partagent un groupe d'objets.

Attributs Les caractéristiques sont appelées des **attributs** de la classe. Chaque objet porte une valeur pour chaque attribut.

Instances et instanciation Dans la relation qui lie la classe à ses objets, on dit que les objets sont des **instances** de la classe, et le mécanisme de création d'objet à partir d'une classe est donc appelé **instanciation** de classe.

Définir une nouvelle classe permet de définir un nouveau type.

Exemple Prenons comme exemple des personnages dans un jeu. Tous les personnages ont un nom, un nombre de points de vie, et un nombre de pièces. La classe Personnage aura donc trois attributs : nom, ptsVie et nbPieces. Chaque instance de Personnage aura une valeur pour ces trois attributs, par exemple le personnage Popoki aura la valeur "Popoki le chat" pour l'attribut nom, la valeur 100 pour l'attribut ptsVie et la valeur 1000 pour l'attribut nbPieces.

2.1.1 Premier diagramme de classes UML

En UML, chaque classe se représente dans un rectangle, qui peut avoir plusieurs compartiments. Le compartiment du haut contient de nom de la classe, celui juste en dessous contient les attributs.

La classe Personnage de notre exemple est représentée en UML à la figure 1. Chaque attribut est décrit par son nom (par exemple *nbPieces*), puis, après les :, de son type (par exemple *integer*). On remarque au début de chaque déclaration d'attribut un symbole moins (-). Cela correspond à la visibilité de l'attribut, qui est ici privée. Ce moins signifie que seuls les objets de la classe Personnage "voient" cet attribut. Nous reviendrons plus tard sur la visibilité.

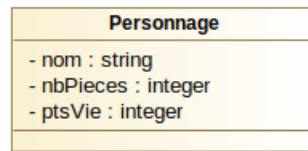


FIGURE 1 – Une classe Personnage avec des attributs

2.1.2 Premier diagramme d'objets UML

Chaque personnage sera un objet, dont le type sera Personnage. On dit aussi qu'un personnage est une instance de la classe Personnage. Pour représenter les objets en UML, on utilise des diagrammes d'objets, où l'on fait apparaître le nom que l'on donne à l'instance, le nom de la classe instanciée, et pour chaque attribut, la valeur portée par l'objet. Si on reprend le personnage Popoki (de nom "Popoki le chat", avec 100 points de vie et 1000 pièces) et qu'on y ajoute le personnage Ronflex (de nom "Ronflex", avec 150 points de vie et 500 pièces), on a la représentation sous forme de diagrammes d'objets donnée à la figure 2.



FIGURE 2 – Deux instances de la classe Personnage (diagramme d'objets)

Notez que le rappel du type des attributs est facultatif, on peut aussi écrire pour un attribut : "nom="Popoki le chat".

2.2 Première classe Java

Regardons ce que donne cette classe personnage en Java.

</>
Programme 1 : Classe Personnage en Java, avec juste des attributs
</>

```

public class Personnage {
    private String nom;
    private int nbPieces;
    private int ptsVie;
}

```

Dans le programme 1 on déclare la classe Personnage, qui est délimitée par des accolades ouvrante et fermante (passons sous silence pour l'instance le mot-clef **public**). Entre ces accolades, on est dans la classe Personnage. Sagement, Java n'utilise pas l'indentation pour délimiter des blocs, mais des accolades. Cela ne dispense pas d'indenter correctement son code, comme c'est fait ici.

On voit ensuite la déclaration des trois attributs. On laisse de côté pour l'instant le mot-clef **private** que l'on commentera plus tard, de même que le mot-clef **public** présent plus loin. Ces attributs sont déclarés avec leur type et leur nom. Le point virgule de fin d'instruction est obligatoire.

2.3 Typer des attributs, multiplicité des attributs

Les attributs ont un type. En UML comme en Java, ce type peut être primitif (entier ou booléen par exemple), peut être une classe, ou encore une énumération. Nous traiterons les énumérations un peu plus tard.

Types primitifs. En UML les types primitifs sont : integer, string, character, float. En Java les types primitifs sont : char, boolean, int, float, long et double ... pas pas String. Il existe une classe String dans la bibliothèque (API) Java qui permet de manipuler des chaînes de caractères.

S'il existe une classe Arme dans notre exemple, on peut envisager un attribut de type Arme, comme illustré ci après en UML (figure 3) et en Java (code 2).

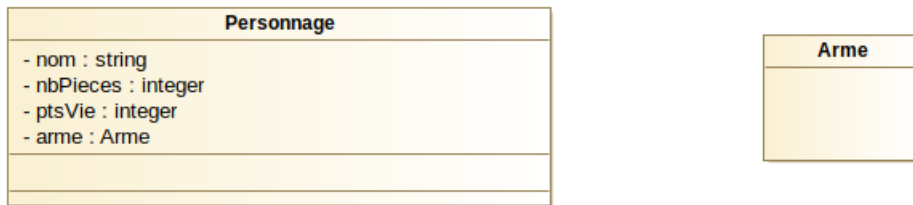


FIGURE 3 – Une classe Personnage avec une arme

</> **Programme 2 : Classe Personnage en Java, avec juste des attributs** </>

```

public class Personnage {
    private String nom;
    private int nbPieces;
    private int ptsVie;
    private Arme arme;
}
  
```

Il est à noter que dans la suite du cours, nous verrons qu'en UML, les attributs typés par des classes sont peu usités, on préfère utiliser des associations.

Multiplicité des attributs Un attribut a une multiplicité, qui indique le nombre minimal et maximal de valeurs portées à un instant donné par l'attribut. Par défaut, cette multiplicité est 1 (qui peut aussi être noté [1..1]). Si on souhaite qu'un attribut puisse porter 0 ou 1 valeur, sa multiplicité est [0..1]. Si un attribut peut porter plusieurs valeurs, sa multiplicité est notée [0..*] ou [1..*]. On peut aussi envisager des multiplicités [2..5] par exemple (de 2 à 5 valeurs). Dans l'exemple précédent, si un personnage peut avoir au même moment plusieurs armes et éventuellement aucune, on aura une multiplicité [0..*] aussi notée [*], comme illustrée à la figure 4.

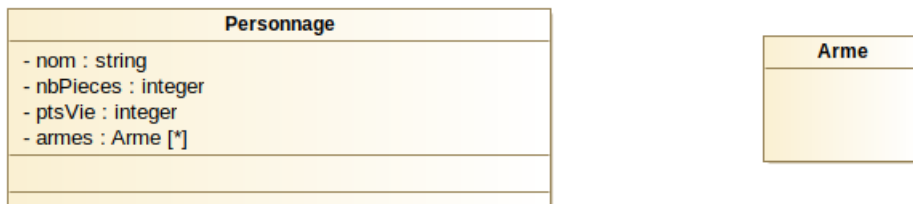


FIGURE 4 – Une classe Personnage avec plusieurs armes

En Java, quand la multiplicité maximale est supérieure strictement à 1, on utilisera des structures de données comme des tableaux. Nous le verrons dans un prochain cours.

2.4 Compléments sur les attributs : attributs dérivés, attributs constants et valeurs initiales

On peut apporter aux attributs des caractéristiques supplémentaires.

2.4.1 Attributs constants

Les valeurs des attributs sont parfois constantes au cours du temps : une fois qu'un objet a pris une valeur pour un attribut constant, il n'en change plus. Par exemple, on peut décider que le nom du personnage de la classe précédente est constant, et qu'une fois qu'un personnage est nommé, le nom ne change plus. Pour ce faire, on rajoute en UML à la fin de la ligne introduisant l'attribut la contrainte : {readonly} ou en ajoutant en tête de déclaration le stéréotype¹ <<readonly>> comme illustré ici. En Java, on utilise le mot-clef `final` comme indiqué au code 3.

1. Nous ne détaillerons pas ici la notion de stéréotype, voyez-les comme une annotation d'élément.

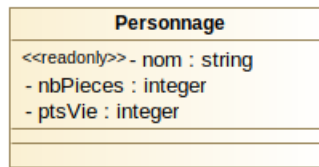


FIGURE 5 – Une classe Personnage avec un attribut constant

</> **Programme 3 : Classe Personnage en Java, avec l'attribut nom constant** </>

```

public class Personnage {
    private final String nom;
    private int nbPieces;
    private int ptsVie;
}

```

2.4.2 Valeur initiale

Certains attributs peuvent avoir une valeur initiale, c'est-à-dire la valeur à laquelle l'attribut sera initialisé pour un objet. Par exemple, dans notre jeu, on peut décider que tout personnage sera initialisé avec 10 points de vie. On le spécifie en UML comme en java en ajoutant cette valeur initiale à la suite d'un signe égal.

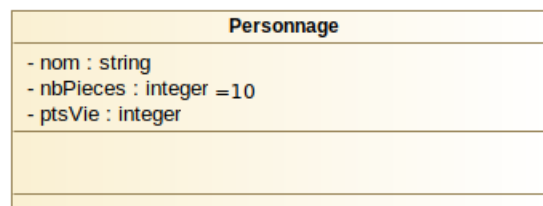


FIGURE 6 – Une classe Personnage avec un attribut avec une valeur initiale

</> **Programme 4 : Classe Personnage en Java, avec une valeur initiale de nombre de pièces** </>

```

public class Personnage {
    private final String nom;
    private int nbPieces=10;
    private int ptsVie;
}

```

Valeur par défaut des attributs en Java Il est à noter qu'en Java, tous les attributs ont une valeur par défaut quand elle n'est pas explicitement précisée : les numériques sont initialisés à zéro, et les booléens à false par exemple. Les attributs typés par une classe ont eux pour valeur par défaut **null**, ce qui représente grosso modo le "rien". Nous y reviendrons.

2.4.3 Attributs dérivés

Il arrive parfois que la valeur d'un attribut puisse être déduite d'autres éléments de la classe. Par exemple, pour notre classe personnage, on peut envisager un attribut estVivant, de type booléen, dont la valeur peut être déduite de celle de l'attribut ptsVie : un personnage est vivant si son nombre de points de vie est strictement positif.

En UML, on appelle un tel attribut un attribut dérivé, puisque sa valeur se dérive d'autres éléments. Cela est indiqué avec un "/" en début de déclaration. On complète en général la description en plaçant dans une note (grosso modo l'équivalent UML de la notion de commentaire) la règle de dérivation.

Il n'y a pas d'équivalent en Java, on ne peut pas directement associer à un attribut Java une règle de calcul. En Java, un attribut dérivé sera implémenté par un attribut Java normal (la valeur de l'attribut sera alors effectivement stockée, et il faudrait prendre garde à la mettre à jour correctement), ou par une méthode (la valeur ne sera alors pas stockée mais calculée à la demande). Nous y reviendrons.

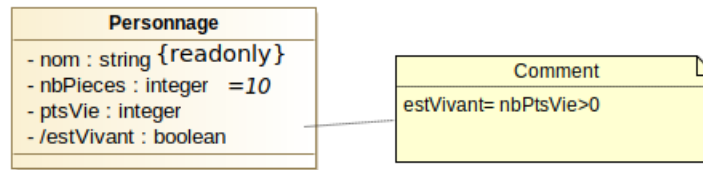


FIGURE 7 – Une classe Personnage avec un attribut dérivé

3 Comportement des objets : opérations et méthodes

Nous avons vu comment définir des classes avec des attributs. Ces classes peuvent engendrer des objets, instances des classes. Pour l’instant, les objets ne peuvent rien faire : ils se contentent de porter des valeurs pour les attributs. Pour animer les objets, on va leur envoyer des messages pour déclencher l’exécution de comportements pré-définis : ces comportements sont définis dans la classe, sous forme d’opération (terminologie UML) ou de méthodes (terminologie Java). Nous commencerons cette section par aborder des méthodes très particulières qui pilotent le cycle de vie des objets. Nous voyons ensuite les méthodes/opérations qui animent les objets.

3.1 Cycle de vie des objets : construction et destruction d’objets

On définit une classe dans l’optique de pouvoir l’instancier (sauf cas très particuliers), c’est-à-dire que l’on souhaite pouvoir ensuite disposer d’objets instances de cette classe. Ces objets ont une durée de vie et seront ensuite éventuellement détruits.

3.1.1 Construction d’objets et constructeurs

On appelle constructeur d’une classe l’opérateur de construction d’objets par instanciation de cette classe.

Lorsque l’on veut construire un objet instance d’une classe, on souhaite souvent donner des valeurs particulières aux attributs de la classe pour l’instance construite. C’est pourquoi il y a souvent un ou plusieurs constructeurs paramétrés dans une classe. Le constructeur sans paramètre est souvent appelé constructeur par défaut, il permet de créer des instances avec des valeurs par défaut pour chacun des attributs.

Constructeur en UML En UML, les constructeurs sont placés dans le compartiment des classes juste en dessous du compartiment des attributs. Ce compartiment contient toutes les opérations de la classe, on distingue les constructeurs par le fait qu’ils portent le stéréotype `<<create>>`.

Dans l’exemple de la figure 8, on a deux constructeurs, l’un paramétré et l’autre pas.

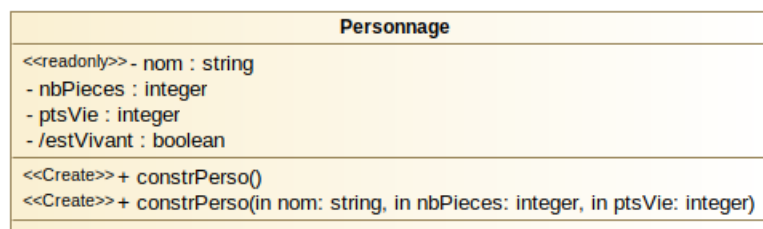


FIGURE 8 – Une classe Personnage avec deux constructeurs

Les paramètres sont typés et séparés par des virgules. Le mot-clef `in` signifie que la valeur est reçue par le constructeur, nous reviendrons sur ce point quand nous verrons le détail des opérations. On peut noter en tête de déclaration un symbole plus (+) qui signifie ici une visibilité publique : le constructeur pourra être invoqué depuis n’importe où. C’est très généralement le cas.

Constructeur en Java Les constructeurs en Java portent le nom de la classe dans laquelle ils se trouvent. Il est à noter qu’en l’absence de constructeur dans une classe, on dispose tout de même d’un constructeur par défaut, non paramétré, qui initialise les valeurs des attributs pour l’objet créé aux valeurs par défaut (la valeur par défaut est par l’attribut lui-même ou par son type).

```

public class Personnage {
    private final String nom;
    private int nbPieces;
    private int ptsVie;
    public Personnage(String nom, int nbPieces, int ptsVie) {
        this.nom = nom;
        this.nbPieces = nbPieces;
        this.ptsVie = ptsVie;
    }
    public Personnage() {
        nom = "anonymous";
        nbPieces = 10;
        ptsVie = 3;
    }
}

```

Nous voyons au programme 5 deux constructeurs de personnages, qui seront utilisés pour instancier la classe Personnage. L'un des constructeurs est paramétré : il réclame les valeurs qui sont nécessaires pour construire le personnage, c'est-à-dire ici les valeurs pour les 3 attributs. Le deuxième constructeur est non paramétré, il donne des valeurs par défaut pour le nouveau personnage (ici le nom "anonymous", 3 points de vie et 10 pièces). Dans le constructeur paramétré, on voit une utilisation du mot-clef `this`, qui désigne l'instance courante, c'est-à-dire ici l'objet que l'on est en train de construire. `this.nom` désigne alors le nom de l'instance courante, et `this.nbPieces` désigne alors le nombre de pièces de l'instance courante. Ici le mot-clef `this` est obligatoire pour lever l'ambiguïté entre les paramètres et les attributs, car ils sont de même type et de même nom. On voit que ce mot-clef n'est par contre pas utilisé dans le constructeur par défaut : il n'y a pas d'ambiguïté à lever dans ce cas.

En Java, la valeur d'un attribut pour un certain objet dépend finalement de plusieurs facteurs :

- c'est la valeur donnée explicitement dans le constructeur appelé lors de la création de l'objet ;
- quand aucune valeur explicite n'est donnée dans le constructeur, et qu'il y a une valeur initiale spécifiée à la déclaration de l'attribut, c'est cette valeur initiale qui est utilisée ;
- sinon, c'est la valeur par défaut associée au type de l'attribut qui est utilisée (par exemple c'est 0 pour les numériques comme les entiers).

Invoquer un constructeur en Java Pour invoquer un constructeur en Java, on doit faire usage du mot-clef `new`. Même si c'est facultatif, on range en général l'objet obtenu dans une variable ou un attribut, comme illustré dans le programme 6. La première ligne crée un objet correspondant au chat Popoki. Cet objet est référencé par la variable de nom "popoki" et de type Personnage. On parle ici de référence d'objet : popoki ne contient pas l'objet, il référence l'objet (ou pointe vers l'objet si l'on veut faire le parallèle avec la notion voisine de pointeur). L'objet est quelque part en mémoire avec les valeurs associées, et popoki est une référence / un lien vers cet emplacement mémoire. Un même objet peut être référencé plusieurs fois. Par exemple, la dernière ligne de notre exemple introduit une variable nommée encorePopoki, on lui affecte "popoki". L'objet correspond au chat popoki est une seule fois en mémoire, et il y a deux références vers lui : popoki et encorePopoki.

```

Personnage popoki=new Personnage("Popoki le chat", 1000, 100);
Personnage ronflex=new Personnage("Ronflex", 500, 150);
Personnage inconnu=new Personnage();
Personnage encorePopoki =popoki;

```

3.1.2 Destruction d'objets

Les objets, une fois créés, peuvent être détruits. Lors de la destruction d'un objet, il y a parfois des actions à effectuer pour effacer proprement les données de l'objet, c'est le rôle du destructeur. En UML, le destructeur se modélise comme le constructeur, mais avec le stéréotype `<<destroy>>`.

En Java, il n'y a pas vraiment de destructeur, car l'interpréteur Java se charge de gérer l'espace mémoire occupé par les données du programme. Quand il n'existe plus de référence vers un objet, alors l'espace mémoire lui correspondant est "récupéré" par l'interpréteur Java. Un peu plus de détail est donné à la sous-section 9.5.

3.2 Animer les objets : opérations et méthodes

Les méthodes / opérations définissent des comportements des instances de la classe. Elles peuvent manipuler les attributs, ou faire appel à d'autres méthodes de la classe. Elles peuvent être paramétrées et retourner des résultats.

3.2.1 Opérations en UML

Les opérations sont les seuls éléments dynamiques du diagramme de classes. Elles se notent dans le compartiment des classes sous celui des attributs.

Un exemple est donné à la figure 9 :

- la méthode `donnePiecesA` est destinée à permettre à un personnage de donner des pièces au personnage passé en paramètre (le nombre de pièces données est également passé en paramètre) ;
- la méthode `estSousSeuilPauvreté` retourne un booléen pour indiquer si le personnage est sous le seuil de pauvreté (afin qu'il puisse bénéficier d'allocations sociales, ce jeu est merveilleux).

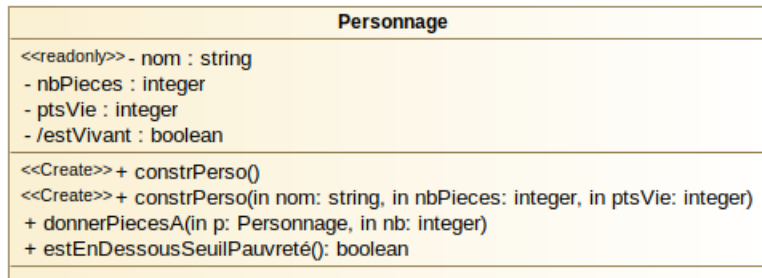


FIGURE 9 – Une classe Personnage avec deux constructeurs et deux méthodes

Une opération a un nom. On essaie en général de lui donner le nom portant le plus de sémantique possible. Une opération, tout comme un attribut, a une visibilité, qui indique qui peut appeler cette opération. Une méthode publique (précédée du symbole plus (+)) pourra être appelée par n'importe quel objet, tandis qu'une méthode privée (précédée du symbole moins (-)) ne pourra être appelée que par des objets instances de la même classe. Nous reviendrons sur la visibilité à la section 7.

Une opération peut avoir des paramètres. On peut spécifier le mode de passage d'un paramètre avec un mot-clef placé juste avant le paramètre :

in le paramètre est une entrée de l'opération, et pas une sortie : il n'est pas modifié par l'opération. C'est le cas le plus courant. C'est aussi le cas par défaut en UML.

out le paramètre est une sortie de l'opération, et pas une entrée. C'est utile quand on souhaite retourner plusieurs résultats : comme il n'y a qu'un type de retour, on donne les autres résultats dans des paramètres out.

inout le paramètre est à la fois entrée et sortie.

Une opération peut avoir un type de retour, qui est placé en fin de ligne, suite à des `:`.

Une opération peut avoir des propriétés précisant le type d'opération, par exemple `{query}` spécifie que l'opération n'a pas d'effet de bord, ce n'est qu'une requête. Les propriétés sont placées entre accolades ou entre chevrons.

On ne spécifie pas dans le diagramme de classes le comportement précis des méthodes. On peut en donner les grandes lignes dans une note de commentaire, ou utiliser d'autres types de diagramme pour modéliser ce comportement, nous ne les verrons pas dans ce cours.

3.2.2 Méthodes en Java

En Java, les opérations UML se traduisent en méthodes.

Comme en UML, on précise la visibilité d'une méthode, de manière similaire à la visibilité d'un attribut. On utilise le mot-clef **public** pour une méthode que l'on peut appeler depuis n'importe où, et **private** pour une méthode que l'on peut appeler depuis une instance de la même classe. Nous reviendrons plus en détail sur la notion de visibilité dans la section 7.

Comme en UML, les méthodes ont des paramètres, qui sont typés. On ne précise pas en Java le "sens" des paramètres. Le mode de passage des paramètres n'est pas décidé par le programmeur, mais régi par des règles Java : les types primitifs sont passés par valeur, les types objets sont passés par référence. Le passage de paramètres en Java est détaillé à la section 9.3.

Le type de retour est indiqué en tête de la déclaration, juste avant le nom de la méthode. Quand une méthode ne retourne rien, alors ce type est indiqué **void**.


```

public class Personnage {
    private String nom;
    private int nbPieces;
    private int ptsVie;

    public Personnage(String nom, int nbPieces, int ptsVie) {
        this.nom = nom;
        this.nbPieces = nbPieces;
        this.ptsVie = ptsVie;
    }

    public void donnerPiecesA(Personnage p, int nbP) {
        nbPieces=nbPieces-nbP;
        p.nbPieces=p.nbPieces+nbP;
    }

    public boolean estSousSeuilPauvreté(){
        return nbPieces<20; // seuil arbitrairement fixé à 20
    }
}

```

Dans la classe donnée au code 7, nous trouvons la méthode `donnerPiecesA`. Cette méthode n'est pas une fonction au sens où elle ne retourne pas de valeur. Son type de retour, indiqué en tête de méthode, est donc **void**. Cette méthode prend en paramètre le personnage à qui on va donner les pièces, et le nombre de pièces à donner. Dans le corps de la méthode (délimité par des accolades) on enlève `nbP` au nombre de pièces de l'instance courante, c'est-à-dire l'objet qui recevra l'appel à cette méthode. On rajoute `nbP` au nombre de pièces de `p`. Ici, à la ligne `nbPieces=nbPieces-nbP`; la référence à `this` est implicite. On aurait aussi pu écrire `this.nbPieces=this.nbPieces-nbP`; pour la rendre explicite. Par contre écrire `this.nbP` ne fait pas de sens : `nbP` ne fait pas partie de l'instance courante, c'est un entier pris en paramètre. On trouve également la méthode `estSousSeuilPauvreté`² qui retourne un booléen. La constante 20 présente dans le corps de cette méthode est franchement une mauvaise idée, nous verrons plus tard comment procéder pour l'éviter.

3.3 Invocation de méthodes

Les méthodes placées dans les classes ne s'exécutent que quand elles sont invoquées. Pour invoquer une méthode, il faut tout d'abord disposer d'un objet capable de recevoir l'appel de cette méthode. Par exemple, pour invoquer la méthode `estSousSeuilDePauvreté` de la classe `Personnage`, il faut disposer d'une instance de la classe `Personnage`, et c'est à cette instance précise qu'on va pouvoir demander si elle est sous le seuil de pauvreté.

```

Personnage popoki=new Personnage("Popoki le chat", 1000, 100);
boolean estPauvre=popoki.estSousSeuilPauvreté();

```

On voit une telle invocation dans le code 8. On utilise une notation dite pointée³ : d'abord le nom de l'instance sur laquelle on va appeler la méthode, un point, puis le nom de la méthode appelée et ses éventuels paramètres effectifs. On parle aussi d'envoi de message : l'instance sur laquelle est appelée la méthode est le receveur d'un message d'appel de méthode. Ici, `popoki` est le receveur.

4 Exécuter un programme orienté objets

Nous avons écrit notre première classe dans sa version minimale. Comment l'exécuter ? Si l'on y réfléchit un peu, exécuter une classe n'a pas de sens. Que faudrait-il exécuter ? Avec quelles valeurs ? De fait, pour "exécuter" un programme à objets, il faut créer des instances, et appeler des méthodes sur ces instances. Mais où écrire un tel code ? En java, on l'écrit dans une méthode particulière appelée `main`, et qui a une signature étrange (qui s'explique très bien mais nous passerons cela sous silence ici) et qui joue le rôle de point d'entrée du programme.

2. Il est fortement déconseillé de placer des accents dans les noms des méthodes et de tout autre éléments Java mais la maniaquerie des enseignants avec les accents est ce qu'elle est ...

3. La même notation est utilisée pour l'accès aux attributs, d'un objet ; cependant celle-ci est peu utilisée du fait que les attributs sont en général privés.

Il est à noter que si l'on souhaite exécuter le programme pour le tester, il est préférable de placer le code de test dans des méthodes de test, éventuellement dans des classes de test. Nous n'aborderons pas finement le test dans cette UE.

```
</> Programme 9 : Utilisation de la Classe Personnage en Java, première version </>

public static void main(String[] args) {
    Personnage poki=new Personnage("Popoki le chat", 100, 1000);
    Personnage flex=new Personnage("Ronflex", 150, 1500);
    flex.donnerPiecesA(poki, 25);
}
```

Le code du listing 9 illustre donc une méthode main. La première ligne de cette méthode crée poki, un objet de type Personnage. La ligne suivante crée flex, un autre objet de type Personnage. À la ligne suivante, on voit un appel de méthode. La méthode donnerPiecesA est appelée sur l'objet flex. En d'autres termes, l'objet flex reçoit le message d'appel de la méthode donnerPiecesA. Des paramètres effectifs sont passés lors de cet appel de méthode, ici : poki et 25. À l'exécution de cette instruction, le flot de contrôle passe à la méthode donnerPiecesA vue au listing 1. Le contexte d'exécution est alors placé de telle sorte que `this` correspond à flex, et p à poki. On note au passage que les objets sont passés par référence : quand l'objet poki est passé en paramètre à la méthode donnerPiecesA, ce que reçoit la méthode n'est pas une copie de l'objet poki (et de ses valeurs d'attributs) mais une référence vers cet objet. Ainsi, quand la méthode effectue des modifications sur le personnage p, comme à l'exécution de notre exemple p est une référence à poki, c'est poki qui est modifié. De retour de l'appel de méthode (à la fin du main), poki (et flex) sont modifiés par rapport à leur création.

5 Caractéristiques de classe, caractéristiques d'instances

Certaines caractéristiques (attributs ou méthodes) ont trait aux instances, d'autres sont indépendantes des instances, elles n'ont trait qu'aux classes.

Pour clarifier, on se place uniquement dans le cadre des attributs pour commencer, en se penchant sur l'exemple de seuil de pauvreté de nos personnages. Nous avons vu que placer le seuil de pauvreté "en dur" dans la méthode n'était pas une bonne chose. En effet, si plusieurs méthodes doivent se servir de ce seuil, c'est une mauvaise idée de dupliquer cette valeur, en cas d'évolution ultérieure de la valeur. On pourrait alors avoir comme idée de placer le seuil de pauvreté comme attribut de la classe Personnage, initialisée à 20 (par exemple). L'avantage est qu'alors on se sert de cet attribut dans les méthodes, partout où on utilise le seuil de pauvreté. Mais alors chacune des instances de Personnage portera une valeur, qui sera la même (par exemple : 20). Si l'on a 1000 objets de type Personnage, il y aura donc 1000 fois écrit "20" en mémoire. Ce n'est pas une très bonne idée. Si l'on y réfléchit, l'attribut n'a pas une valeur propre à chaque instance, mais qui est globale à toutes les instances. On parle dans ce cas d'attribut de classe (par opposition à attributs d'instance, qui sont ceux que nous avons vu jusqu'ici). En UML, on le note en soulignant l'attribut.

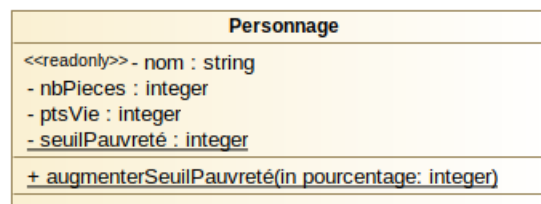


FIGURE 10 – Une classe Personnage avec un attribut et une méthode de classe

En Java, on utilise souvent le terme d'attribut statique, du fait du mot-clef `static` qui dénote ces attributs dans le code Java. Pour le seuil de pauvreté, il serait déclaré ainsi : `private static int seuilPauvreté=20;`

Un attribut de classe peut avoir une valeur qui bouge au cours du temps : ici le seuil de pauvreté pourrait par exemple passer à 25. Si au contraire l'attribut est de classe et constant, alors il s'agit d'une constante du système. Par exemple, c'est avec un attribut de classe et constant qu'on définirait la constante mathématique π .

Il existe le pendant des attributs de classe du côté des méthodes : les méthodes de classe. Ce sont des méthodes que l'on peut appeler sans disposer d'instance de la classe. Ces méthodes ne peuvent pas manipuler d'attributs d'instances, ni appeler des méthodes d'instances, car cela ne ferait pas de sens. En UML, les méthodes de classe sont soulignées. En java, on utilise une fois encore le mot-clef `static`.

```
</> Programme 10 : Personnage en Java avec attribut et méthode de classe </>

public class Personnage{
    private static int seuilPauvreté=20;
```

```

    public static void augmenterSeuilPauvreté(int pourcentage){
        seuilPauvreté=seuilPauvreté*(1+pourcentage*0.01);
    }
}

```

Pour appeler une méthode de classe, on peut soit le faire comme une méthode d'instance en utilisant le nom d'une instance (c'est déconseillé car propice aux confusions), soit en utilisant le nom de la classe à la place au nom de l'instance, comme illustré au code 11, dans lequel on augmente de 2% le seuil de pauvreté. On procède de même pour accéder aux valeurs des attributs

```

</>                               Programme 11 : Appel de méthode de classe                               </>
Personnage.augmenterSeuilPauvreté(2);

```

6 Type de données énuméré : les énumérations

Une énumération est un type de données dont on peut énumérer toutes les valeurs possibles.

Par exemple :

- la civilité d'une personne qui a pour valeurs possibles : Mme, M
- les stations de ski d'un grand domaine qui ont pour valeurs possibles : Valmorel, Combelouvière, Saint-François-Longchamp
- les niveaux à l'école primaire qui ont pour valeurs : CP, CE1, CE2, CM1, CM2

Les valeurs possibles sont appelées les littéraux d'énumération.

Les valeurs possibles doivent être connues statiquement en Java, elles ne peuvent pas être modifiées (notamment enrichies) en cours d'exécution.

On peut par exemple dans notre exemple envisager que chaque personnage puisse avoir un don à choisir parmi : guérison, téléportation, invisibilité.

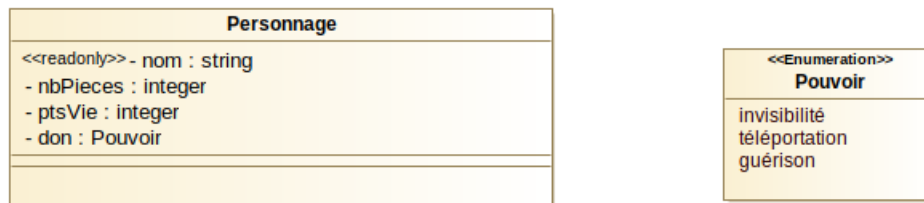


FIGURE 11 – Une classe Personnage et une énumération

```

</>                               Programme 12 : Une énumération en Java                               </>

```

```

public enum Pouvoir{
    guérison,
    téléportation,
    invisibilité;
}

```

Pour utiliser une énumération Java, on utilise une notation pointée : nom de l'énumération, point, nom du littéral.

```

</>                               Programme 13 : Classe Personnage utilisant l'énumération Pouvoir en Java                               </>

```

```

public class Personnage{
    private Pouvoir don;

    public void devientGuérisseur(){
        don=Pouvoir.guérisseur;
    }
}

```

7 Encapsulation et espaces de noms

7.1 Organisation en paquetages

Les systèmes contiennent en général un grand nombre de classes, il est donc nécessaire de disposer d'un mécanisme pour les organiser, les classer. Au niveau des modèles UML, on peut souligner qu'il n'y a pas que les classes à organiser, mais d'autres éléments dont nous parlerons peu dans cette UE. Dans un paquetage, on peut imbriquer d'autres paquetages, on obtient donc des hiérarchies de paquetages.

On peut faire le parallèle avec les hiérarchies de répertoires des systèmes de fichiers. Pour organiser ses fichiers dans un système de fichiers, on utilise des répertoires, qui sont possiblement imbriqués les uns dans les autres.

De même qu'il n'y a pas une unique façon d'organiser ses répertoires, il n'y a pas de codification de la façon dont on doit organiser son système en packages.

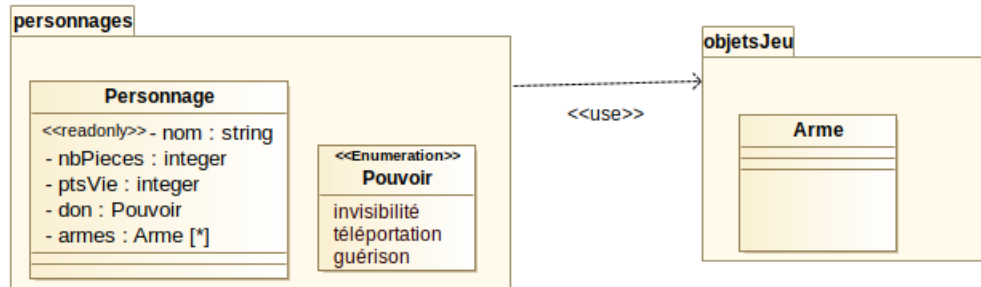


FIGURE 12 – Illustration des packages en UML

En UML, les packages se notent par de grands rectangles (avec un petit onglet portant leur nom) englobant les éléments contenus. On peut préciser qu'un paquetage en utilise un autre. Dans notre exemple, nous avons placé la classe **Personnage** et l'énumération **Pouvoir** dans le paquetage **personnages**, et la classe **Arme** dans le paquetage **objetsJeu**. Puisqu'un personnage a ici des armes, le paquetage **personnages** utilise le paquetage **objetsJeu**.

En java, pour placer une classe (ou une énumération) dans un package, on ajoute avant la déclaration de la classe une ligne de type : `package personnages;`, comme illustré au code 14.

Pour utiliser dans une classe une autre classe qui n'est pas dans le même package, il faut soit l'importer (voir illustration au code 15), soit l'utiliser en la préfixant par le nom du package (voir illustration au code 16).

</> **Programme 14 : Classe Arme dans un package en Java** </>

```
package objetsJeu;

public class Arme{
    ...
}
```

</> **Programme 15 : Packages et imports en Java : une version de la classe Personnage** </>

```
package personnages;

import objetsJeu.Arme;

public class Personnage {
    private Arme arme;
    ...
}
```

</> **Programme 16 : Packages et imports en Java : une autre version de la classe Personnage** </>

```
package personnages;

public class Personnage {
    private objetsJeu.Arme arme;
```

```
...
}
```

Les packages définissent des espaces de noms en Java. On peut ainsi avoir deux classes homonymes dans 2 packages différents, Java sera à même de les différencier.

L'API Java est elle-même organisée dans de nombreux packages.

7.2 Visibilité

Nous avons déjà brièvement abordé la visibilité des attributs et des méthodes, en UML et en Java. Nous y revenons ici plus en détails.

La visibilité d'un élément détermine quels objets peuvent "voir" cet élément. La visibilité d'un attribut détermine plus précisément quels objets peuvent accéder à la valeur de cet attribut (accès en lecture ou en écriture). La visibilité d'une méthode détermine quels objets peuvent invoquer la méthode.

7.2.1 Visibilité en UML

En UML, il y a seulement 4 sortes de visibilité possible, donnant accès à un éléments à 4 grandes classes d'objets :

- visibilité publique, notée +, pour donner accès à tous les objets ;
- visibilité privée, notée -, pour donner accès uniquement aux objets de la même classe ;
- visibilité au paquetage, notée ~, pour donner accès à tout objet dont la classe de base est dans le même package ;
- visibilité protégée, notée #, pour donner accès à tous les objets de même classe, ou d'une de ses sous-classes.

Nous laissons temporairement de côté la dernière visibilité que nous reverrons lors d'un prochain cours.

Si l'on reprend l'exemple de la figure 8, les 4 attributs ont été laissés privés. Cela signifie que seules des instances de la classe `Personnage` y ont accès. Au contraire, les méthodes ont été mises publiques : cela signifie que n'importe quel aura le droit de l'invoquer (à condition bien sûr de disposer d'une référence d'objet instance de la classe `Personnage`).

7.2.2 Visibilité en Java

En java, la visibilité est très proche du modèle proposé par UML :

- visibilité publique, notée avec le mot-clef **public**, pour donner accès à tous les objets ;
- visibilité privée, notée avec le mot-clef **private**, pour donner accès uniquement aux objets de la même classe ;
- visibilité au paquetage, indiquée par une absence de mot-clef, pour donner accès à tout objet dont la classe de base est dans le même package ;
- visibilité protégée, notée avec le mot-clef **protected**, pour donner accès à tous les objets de même classe, ou d'une de ses sous-classes, ainsi qu'à tous les objets dont la classe est dans le même package.

La visibilité est une sorte de sécurité sur l'accès aux données portées par un objet : on contrôle qui peut voir les valeurs des attributs et qui peut appeler les méthodes. L'idée là derrière est de ne pas laisser n'importe quel objet venir faire n'importe quoi avec un autre objet.

En général (notamment à l'exception des constantes), les attributs sont laissés privés. Ainsi, ils ne peuvent être manipulés que par des objets de la même classe, et en qui on a donc toute confiance pour s'en servir à bon escient. Très pragmatiquement, l'accès à un attribut privé n'aura lieu que dans le code de la classe dans laquelle il est défini, par exemple dans une de ses méthodes.

À de rares exceptions, les constructeurs sont laissés publics, ce qui est assez naturel puisqu'ils servent à créer des objets.

Les méthodes privées sont des méthodes qui sont appelées uniquement par d'autres méthodes de la même classe, mais qui n'ont pas vocation à être utilisées ailleurs.

Notons enfin qu'en Java, d'autres éléments ont une visibilité comme les classes (d'où le mot-clef **public** présent en tête de chaque classe jusqu'ici) mais nous n'aborderons pas cela dans cette UE.

7.2.3 Accesseurs d'attributs

Comme nous l'avons vu, les attributs sont en règle générale privés, pour éviter que les valeurs des attributs soient mal manipulées. Regardons par exemple l'attribut `ptsVie` de la classe `Personnage`. Cet attribut est un entier, mais en fait, il ne doit pas être négatif. On ne peut pas laisser n'importe quel objet y accéder directement, car alors un objet mal informé pourrait y placer une valeur négative. De manière similaire, quand la valeur de cet attribut est modifiée et devient nulle, l'objet doit réagir, puisque le personnage meurt. Un objet externe pourrait placer le nombre de points de vie à 0 et oublier de prévenir le personnage qu'il est mort par exemple. Donc il faut laisser l'attribut privé, et contrôler, au sein de la classe `Personnage`, comment il doit être modifié (accès en écriture). L'accès en lecture pose ici moins de problème. Bien que l'attribut soit privé, on veut tout de même que d'autres objets puissent un peu accéder aux valeurs de l'attribut, mais pas n'importe comment, d'une manière qui aura été prévue par la classe : c'est le rôle des accesseurs. Les accesseurs sont des méthodes qui contrôlent l'accès aux attributs. Pour un même attribut, on peut prévoir un accesseur en lecture et/ou un accesseur en écriture. L'accesseur en lecture peut permettre de

faire des statistiques sur les accès à l'attribut. L'accesseur en écriture peut permettre d'effectuer des vérifications sur les valeurs, ou bien encore de prendre en charge des attributs dérivés.

accesseurs et attributs dérivés Un attribut dérivé peut être implémenté par une méthode, ou un attribut mis à jour quand cela est nécessaire. Ainsi l'accesseur en écriture d'un attribut peut permettre d'aller mettre à jour un attribut dérivé qui en dépend.

Convention de nommage Par convention, on nomme `getAtt` et `setAtt` les accesseurs en lecture et en écriture sur un attribut de nom `att`.

Exemples On donne ci-après plusieurs exemples. Dans le code 17, l'attribut dérivé `estSousSeuilPauvreté` est implémenté avec un attribut normal, qui est mis à jour dans l'accesseur en écriture `setNbPieces`. On note également qu'on ne met pas d'accesseur en écriture pour le nom, puisqu'il est constant.

</> Programme 17 : Accesseurs en Java </>

```
public class Personnage {
    private final String nom;
    private int nbPieces;
    private int ptsVie;
    private boolean estSousSeuilPauvreté;
    private static int seuilPauvreté=10;

    public String getNom() {
        return nom;
    }

    public int getNbPieces() {
        return nbPieces;
    }

    public void setNbPieces(int nbPieces) {
        if (nbPieces>=0) {
            this.nbPieces = nbPieces;
            estSousSeuilPauvreté=nbPieces<Personnage.seuilPauvreté;
        }
    }

    public int getPtsVie() {
        return ptsVie;
    }

    public void setPtsVie(int ptsVie) {
        if (ptsVie>=0) {
            this.ptsVie = ptsVie;
        }
        if (ptsVie==0) {
            System.out.println("argh ! Le personnage "+nom+" meurt ...");
        }
    }

    public boolean getEstSousSeuilPauvreté() {
        return estSousSeuilPauvreté;
    }
}
```

Dans la version du code 18, l'attribut dérivé n'est plus implémenté par un attribut mais juste par son accesseur en lecture qui calcule la valeur à retourner à partir du nombre de pièces.

```
public class Personnage {
    private final String nom;
    private int nbPieces;
    private int ptsVie;

    private static int seuilPauvreté=10;

    public String getNom() {
        return nom;
    }

    public int getNbPieces() {
        return nbPieces;
    }

    public void setNbPieces(int nbPieces) {
        if (nbPieces>=0) {
            this.nbPieces = nbPieces;
        }
    }

    public int getPtsVie() {
        return ptsVie;
    }

    public void setPtsVie(int ptsVie) {
        if (ptsVie>=0) {
            this.ptsVie = ptsVie;
        }
        if (ptsVie==0) {
            System.out.println("argh ! Le personnage "+nom+" meurt ...");
        }
    }

    public boolean getEstSousSeuilPauvreté() {
        return nbPieces<Personnage.seuilPauvreté;
    }
}
```

7.3 Encapsulation et dialogues entre objets

En programmation, l'encapsulation est le principe qui consiste à regrouper des données avec procédures/fonctions permettant de les manipuler. L'encapsulation est souvent accompagnée Ce principe est souvent accompagné du masquage des données pour assurer que les données ne sont manipulées que via les procédures/fonctions qui ont été prévues à cet effet. La programmation par objet telle que nous l'avons vu jusqu'ici permet en ce sens l'encapsulation des données. Les données sont les valeurs portées par les attributs, elles sont manipulées par les méthodes/opérations qui ont été prévues à cet effet. Chaque objet veille donc sur ses données, au travers des opérations prévues dans la classe. Cela est illustré à la figure 13 : les données sont au cœur de l'objet, on n'y accède qu'au travers des méthodes placées en périphérie (à l'interface). Notez que cette représentation n'est pas du tout fidèle à la représentation mémoire.

Quand un objet a besoin d'une donnée qu'il n'a pas, il la demande à un autre objet, en lui envoyant un message d'appel de méthode. Ainsi, les objets conversent, c'est cette image qui a donné le nom du langage à objets SmallTalk.

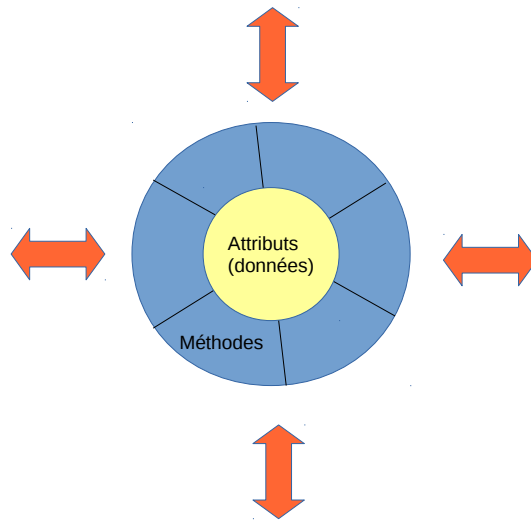


FIGURE 13 – Encapsulation des données dans un objet

8 Modélisation de classes : quelques recommandations

9 Quelques caractéristiques du langage Java

9.1 Compilation et exécution

9.2 Types primitifs et types objets

9.3 Références d'objets et comparaison d'objets

9.4 Structures de contrôle usuelles

9.5 Gestion de la mémoire

9.6 Conventions de nommage

9.7 La méthode toString

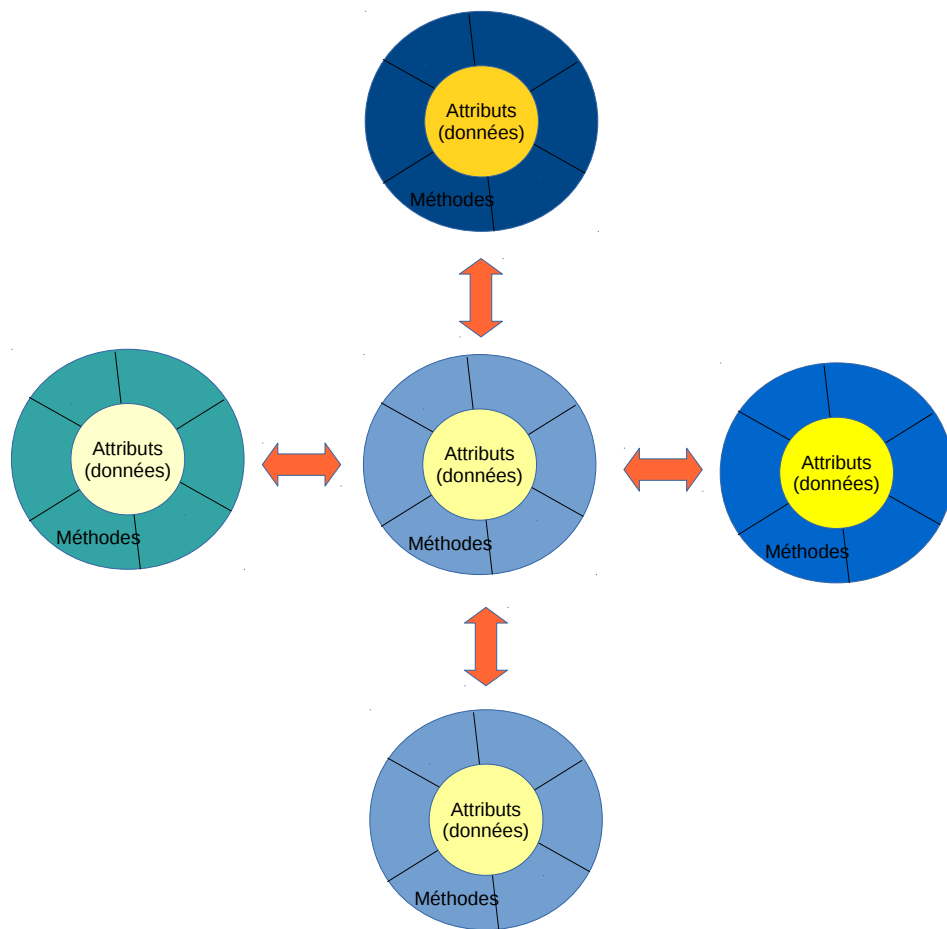


FIGURE 14 – Encapsulation et communication entre objets