Programmation orientée objet

Présentation de la programmation orientée objet

Plan du cours

- Avant propos
- Manipulation des objets et des classes
- Construction de nouvelles classes
- Héritage, classes abstraites, interfaces
- Visibilité et encapsulation

Avant propos

- La programmation orienté objets est ardue. Il faut un certain temps pour assimiler tous les concepts associés.
- Pour bien aborder la suite de ce module, il faut **apprendre** et admettre un certain nombre de concepts sans forcement les comprendre (tout du moins dans les premières semaines).
- La POO fait appel à de nombreux mots de vocabulaire **précis**, il est impératif de les connaître pour comprendre (et se comprendre...).
- N'hésitez pas à poser des question au fur et à mesure du cours !
- Les différents éléments seront détaillés en TD/TP ce qui en éclairera la compréhension.
- Pour bien comprendre la POO, il faut appréhender beaucoup de notion de manière simultanées, il n'est pas possible de tout présenter en même temps, relisez donc ce cours plusieurs fois.
- A la fin du cours, vous comprendrez la notation :

```
System.out.println("...");
```

UML en quelques mots

- Dans ce cours, quelques éléments d'UML sont présentés pour illustrer les relations.
- UML est une méthode de modélisation des problèmes informatiques très complète.
- L'approche retenue a été un peu simplifiée par rapport à la norme, ce module n'est pas un cours de génie logiciel.
- Dans un schéma UML, les types sont souvent écrit «en français» car le même schéma UML pourrait être utilisé pour du Java, C++, ... sans grosses modifications.
- Entre Java et UML, certains notation sont inversées (type de retour des méthodes, ...) il est **impératif** de respecter cette contrainte.
- Par contre, vous devez savoir comment passer d'un diagramme UML à un fichier Java sans hésitation ; ce n'est qu'une "mécanique" que les ordinateurs sont capables de réaliser très simplement. Cette opération ne nécessite aucune intelligence, juste l'application de règles simples.

Utilisation d'objet

- Dans le cours précédent nous avons utilisé quelques classes :
  - les chaînes de caractères : `String`
  - le lecteur de flux : `Scanner`
  - le générateur aléatoire : `Random`
  - ...
- De très nombreuses classes sont disponibles dans l'API standard couvrant les principaux besoins (plus d'infos disponibles sur : http://java.sun.com/j2se/1.5.0/docs/api/)
- Il est aussi possible (et c'est le but) de créer de nouvelle classes en associant des classes existantes.

Classes et objets

- `String`, `Scanner`, ... sont des **classes**. Les classes définissent les **propriétés** et les **méthodes** communes à tous les éléments qui seront crées.
- Chaque élément crée est **objet**.
- Les classes définissent :
  - Des **propriétés** (ou **attributs**) qui stockent l'information,
  - Des **méthodes** utilisées pour modifier l'information ou accéder à l'information.

Classes et objets

- Pour pouvoir utiliser un objet, il faut définir une variable qui permettra d’accéder à cet objet.

- La **déclaration** se fait avec le nom d’une classe suivie du nom de la variable.

- L’objet est ensuite crée en utilisant l’opérateur `new` suivi du **constructeur** (une méthode qui porte généralement le même nom que la classe). Cette opération porte le nom de **création** ou **instanciation**.

- Pour accéder aux attributs ou aux méthodes on utilise la variable suivie d’un point et de l’attribut ou de la méthode (qui a **obligatoirement** des parenthèses même s’il n’y a rien dans les parenthèses).

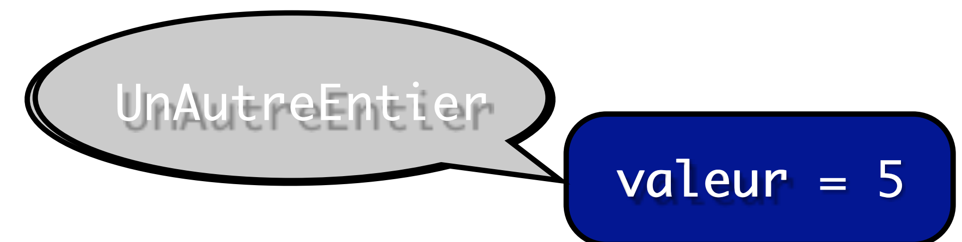
- Les objets pouvant en contenir d’autres, plusieurs identifiants (méthode ou attribut) peuvent être mis l’un au bout de l’autre, simplement séparé par des points.

Objets et références

- Les objets sont désignés par les variables, mais contrairement aux types, les variables **ne** contiennent **pas** les objets.

```
class MonEntier {
    int valeur;
}
```

```
class TestEntier {
    public static void main(String[] args) {
        MonEntier UnEntier;
        MonEntier UnAutreEntier;

        UnEntier = new MonEntier();
        UnAutreEntier = new MonEntier();

        UnEntier.valeur = 5;
        UnAutreEntier.valeur = 5;

        if (UnEntier==UnAutreEntier){
            System.out.println("Les 2 variables désignent
le même objet");
        }else{
            System.out.println("Les 2 variables sont
différentes");
        }
    }
}
```



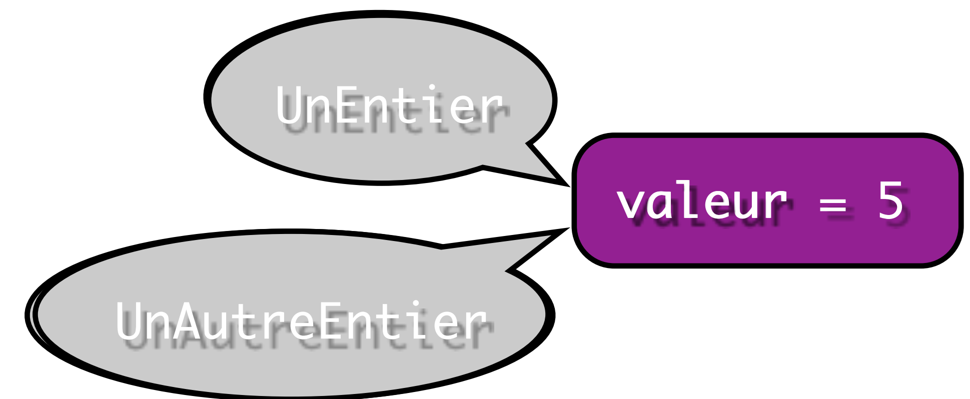

```
Les 2 variables sont différentes
```

Objets et références

- Deux variables peuvent référencer le même objet.

```
class MonEntier {
    int valeur;
}
```

```
class TestEntier {
    public static void main(String[] args) {
        MonEntier UnEntier;
        MonEntier UnAutreEntier;

        UnEntier = new MonEntier();
        UnAutreEntier = UnEntier;

        UnEntier.valeur = 5;

        if (UnEntier==UnAutreEntier){
            System.out.println("Les 2 variables désignent
le même objet");
        }else{
            System.out.println("Les 2 variables sont
différentes");
        }
    }
}
```

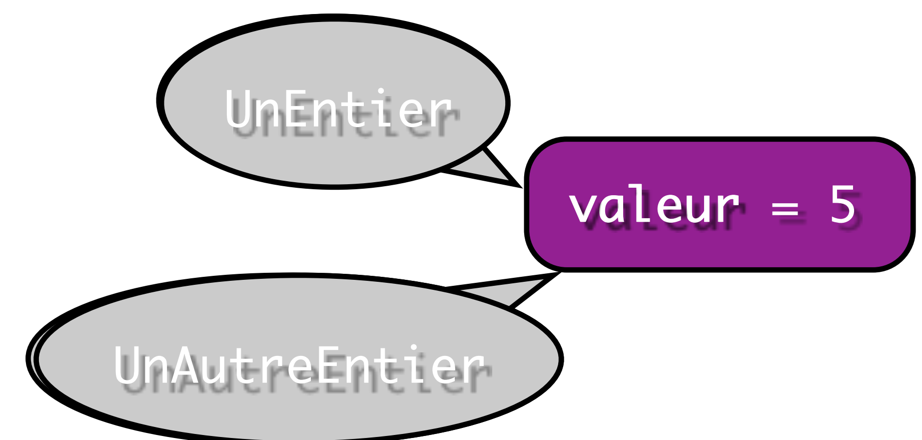



```
Les 2 variables désignent le même objet
```

Destruction d'objet

- Un objet est détruit par un système automatique (le *garbage collector*) quand plus aucune référence ne le désigne.
- Pour supprimer une référence il suffit de l'associer à `null`.
- Quand un objet est détruit toutes les références qu'il contenait sont détruites (mais pas forcement les objets associés, d'autre références peuvent exister ailleurs).

```
class TestEntier {
    public static void main(String[] args) {
        MonEntier UnEntier;
        MonEntier UnAutreEntier;

        UnEntier = new MonEntier();
        UnAutreEntier = UnEntier;

        UnEntier.valeur = 5;

        UnEntier = null;
        UnAutreEntier = null;
    }
}
```



Utilisation des méthodes

- Les méthodes peuvent retourner des données (un type, un objet,...) , elles sont alors utilisées dans une affectation ou un test.
- Les objets reçus sont crées par la méthode, on doit juste déclarer des variables qui les référenceront.
- Pour les méthodes qui ne renvoient pas de donnée, on les utilise simplement sur l'objet.

```
class TestEntier {
    public static void main(String[] args) {
        File fichier;
        FileWriter fw;
        String nomComplet;

        fichier = new File("Toto.txt");

        nomComplet = fichier.getAbsolutePath();
        System.out.println("Le nom complet du fichier
est : " + nomComplet);

        if (fichier.canWrite()){
            fw = new FileWriter(fichier);
            fw.write('a');
            fw.close();
        }
    }
}
```

Déclaration des variables référençant les objets.

Création des objets et affectation références

Appel d'une méthode et affectation de sa valeur de retour

Appel d'une méthode et test de sa valeur de retour

Utilisation de méthodes sans valeur de retour

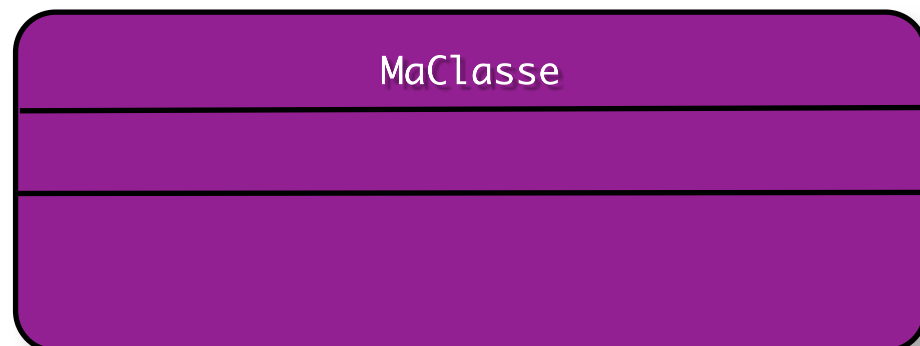
Classes

- Les classes sont des éléments de programme qui définissent les attributs et les méthodes qui seront utilisés par un (ou plusieurs) objet(s).

- En Java une classe doit être dans un fichier et un fichier ne peut contenir qu'une classe (en seule classe déclarée `public` ce qui est la majorité des cas).

- En Java une classe est déclarée en utilisant le mot clé `class` (normalement précédé de `public`) suivie du nom de la classe avec toute l'implémentation entre des accolades

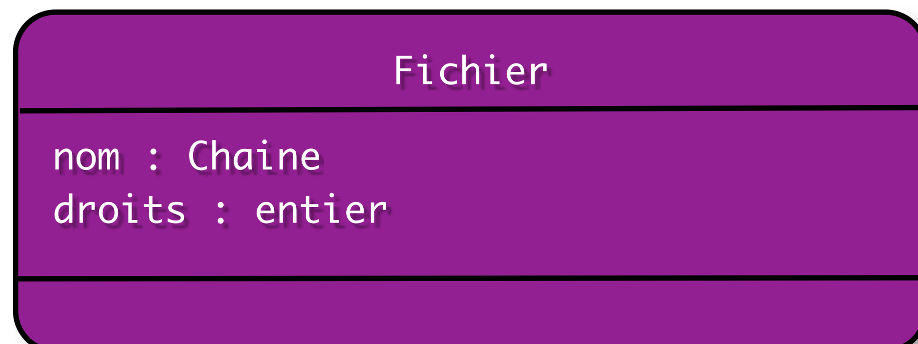
- Dans les schéma UML, les classes sont représentées par une "boite" comprenant 3 cadre avec le nom de la classe dans le premier cadre.




```
public class MaClasse {


}
```

Attributs

- Les attributs sont des données stockés dans les objets.
- Chaque instance de la classe (chaque objet) à sont propre jeu d'attributs.
- Les attributs sont déclarés comme les variables :
  - Un type, une classe ou une interface
  - Un nom permettant d'accéder à cet élément.
- Dans les schéma UML, les attributs sont représentées dans le 2ème cadre.




```java
public class Fichier {
  String nom;
  int droits;
}
```

Lecture et écriture des attributs

- Pour accéder à un attribut présent dans un objet, on utilise la variable qui référence cet objet suivie d'un point et du nom de l'attribut.

- Les attributs (visibles...) peuvent être lus et écrits comme n'importe quelle autre variable.

```
public class Entiers {
    int valeur;
}
```

```
public class Programme {
    public static void main(String[] args) {
        Entiers monEntier
        monEntier = new Entiers();
        monEntier.valeur = 10;
        System.out.println("Dans monEntier la valeur est : " + monEntier.valeur);
        if (monEntier.valeur == 10) {
            System.out.println("La valeur stockée est bien 10");
        }
    }
}
```

Attributs

- Les attributs sont propres à chaque objet, ils ne sont pas partagés entre les différents objets d'une même classe.

```
public class Entiers {
    int valeur;
}
```

```
public class Programme {
    public static void main(String[] args) {
        Entiers monEntier, monAutreEntier;
        monEntier = new Entiers();
        monAutreEntier = new Entiers();
        monEntier.valeur = 10;
        monAutreEntier.valeur = 20;
        System.out.println("Dans monEntier la valeur est : " + monEntier.valeur);
        System.out.println("Dans monAutreEntier la valeur est : " +
monAutreEntier.valeur);
    }
}
```

```
Dans monEntier la valeur est : 10
Dans monAutreEntier la valeur est : 20
```

Attributs statiques

- Un attribut statique est un attribut qui peut être utilisé sans créer d'objet. Pour y accéder, on fait suivre le nom de la classe d'un point puis du nom de l'attribut.
- Tous les objets issus de cette classe se partagent cet attribut.
- Le modificateur `static` permet de définir un attribut statique (ce modificateur est placé juste avant le type). Dans un schéma UML, un attribut statique est souligné.

```
public class MaClasse {
    static int nbElements;
    MaClasse() {
        nbElements++;
    }
}
```

```
Nb elements : 0
Nb elements : 2
Nb elements : 2
```

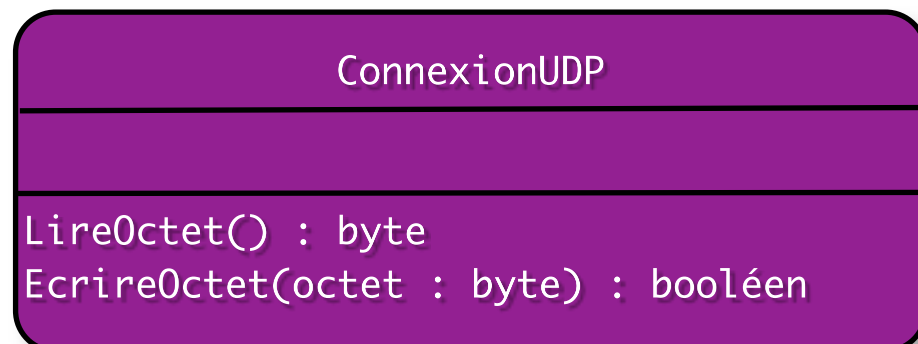
```
public class Essai {
    public static void main(String[] args) {
        MaClasse monObjet, monAutreObjet;
        System.out.println("Nb elements : " + MaClasse.nbElements);
        monObjet = new MaClasse();
        monAutreObjet = new MaClasse();
        System.out.println("Nb elements : " + monObjet.nbElements);
        System.out.println("Nb elements : " + monAutreObjet.nbElements);
    }
}
```

Méthodes

- Les méthodes permettent de modifier les attributs et/ou d'interagir avec l'extérieur.

- Une méthode est composée de 2 parties :
  - la **déclaration** (l'entête, signature, ...) qui représente la liste des arguments d'entrée.
  - **l'implémentation** qui contient les instructions. Dans un programme Java, il ne peut pas y avoir d'instructions (`if`, `for`, ...) ailleurs que dans des implémentations de méthodes.

- Dans les schéma UML, les méthodes sont représentées dans le 3ème cadre.




```java
public class ConnexionUDP {
  byte LireOctet(){return ...}
  boolean EcrireOctet(byte octet){return ...}
}
```

Méthodes

- Une méthode est déclarée par son type de retour suivi du nom de la méthode et des paramètres entre parenthèses.
  - le type de retour peut être un type ou une classe ou `void`
  - les arguments sont déclarés comme des variables (donc type suivi du nom), séparés par des virgules.
- Une méthode dont le type de retour n'est pas `void` doit utiliser l'instruction `return` pour renvoyer un élément du type spécifié.

```
public class Entiers {
    int valeur;

    int Produit(int a){
        int temp;
        temp = valeur * a;
        return temp;
    }

    void Somme(int a){
        valeur = valeur + a;
    }
}
```

Surcharge de méthodes (ou polymorphisme paramétrique)

- Dans une classe, une même méthode peut avoir plusieurs signatures (nombre et type des données d'entrées) différentes. Le nom de la méthode est identique mais le nombre et/ou le type des arguments change.
- Le compilateur choisit quelle méthode appeler en fonction des arguments d'appel.
- Pour un même ensemble d'arguments (combinaison de type et de nombre), il peut y avoir qu'une seule déclaration.

```
boolean Connecter(String url){
    ...
}
boolean Connecter(String url, int port){
...
}
```

```
MonObjet.Connecter("www.google.fr");
...
MonObjet.Connecter("www.serveur.com",8080);
```

```
int Additionner(int a,int b){
    ...
}
int Additionner(int c,int d){
    // Une méthode Additionner(int , int) existe déjà !
}
```

Constructeur

- Le **constructeur** est une méthode particulière qui est appelée lors de l'utilisation de l'opérateur `new`.

- Le constructeur a le même nom que la classe et n'a **aucun** type de retour (même pas `void`).

- Par défaut, un constructeur vide sans argument existe (même s'il n'a pas été écrit dans le code).

```
public class Entiers {
    int valeur;
    public Entiers() {
        valeur = 0;
    }
}
```

```
public class Programme {
    public static void main(String[] args) {
        Entiers monEntier;
        monEntier = new Entiers();
    }
}
```

Constructeur

- Il est très souvent surchargé pour pouvoir initialiser les différents attributs.

- Dans ce cas, la version du constructeur à appeler est utilisée après l'opérateur new.

```
public class Entiers {
    int valeur;
    Entiers() {
        valeur = 0;
    }
    Entiers(int uneValeur){
        valeur = uneValeur;
    }
}
```

```
public class Programme {
    public static void main(String[] args) {
        Entiers monEntier;
        monEntier = new Entiers(10);
    }
}
```

Méthodes statiques

- Une méthode statique est une méthode qui peut être appelée sans créer d'objet. La classe `Math` en fait grand usage.
- Le modificateur `static` permet de définir une méthode statique (ce modificateur est placé juste avant le type de retour).
- Si à l'intérieur de cette méthode, d'autres méthodes (ou d'autres attributs) sont appelés il doivent aussi être déclarés statiques.
- Dans un schéma UML, une méthode statique est soulignée.

```
double val;
double angle = 56.8745;
val = Math.cos(angle);
```

Méthode `main`

- Un programme Java est composé de plusieurs objets interagissant ensemble.
- La machine virtuelle (JVM) exécutera le programme à partir de la méthode `main`.
- Cette méthode est une méthode statique afin qu'elle puisse être appelée sans créer un objet.
- Les arguments de la méthode `main` sont les paramètres de la ligne de commande.

```
public class MaClasse {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            System.out.println("Argument " + i + " valeur : " + args[i]);
        }
    }
}
```

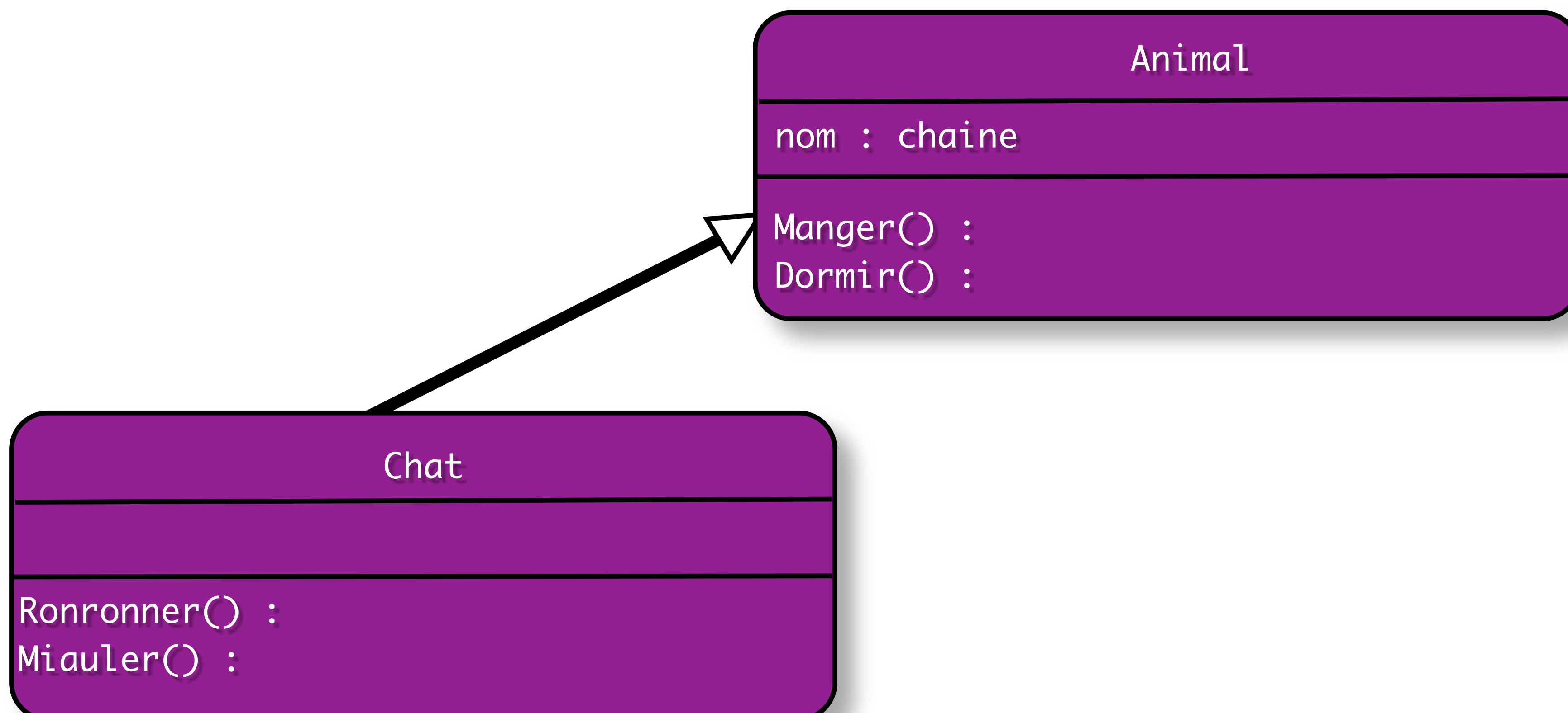
```
new-host:bin jb$ java MaClasse argument1 truc
Argument 0 valeur : argument1
Argument 1 valeur : truc
```

L’héritage

- L’héritage permet de construire une (ou plusieurs) classe(s) à partir d’une classe existante.

- Le code peut être “factorisé” et réutilisé ultérieurement, les mises à jour sont facilitées.

- L’héritage est représenté par une flèche vide dans les schéma UML



L’héritage

- En Java, on déclare un héritage en utilisant le mot clé `extends` lors de la déclaration de la classe.
- A la suite du nom de la classe on ajoute `extends` suivie du nom de la classe mère.
- Il ne peut y avoir qu’une seule classe mère de désignée.
- Tous les éléments présents dans la classe mère se retrouvent dans la classe fille (comme si on réalisait un “copier/coller”).

```
public Chat extends Animal{
    ...
}
```

Redéfinition de méthodes

- Une méthode de la classe mère peut être réécrite dans une classe fille. On parle de **surcharge** (ou polymorphisme d'héritage).

- La méthode doit avoir **exactement** la même définition (nom de la méthode, type et ordre des arguments). La méthode de la classe fille remplace alors celle de la classe mère lors de l'appel.

```
public class Rectangle {
    int largeur;
    int longueur;
    int CalculerAire(){
        return largeur*longueur;
    }
}
```

```
public class Carre extends Rectangle{
    int CalculerAire(){
        return largeur*largeur;
    }
}
```

Redéfinition de méthodes

- Pour réutiliser le code de la classe mère lors de la surcharge, on utilise le mot clé `super` suivit d'un point et de la méthode à appeler.

- La même syntaxe peut être utilisée pour accéder à une propriété de la classe mère.

- Dans le constructeur, `super()` correspond à l'appel du constructeur de la classe mère. Ce doit être la première instruction du constructeur si on veut l'utiliser.

- Pour appeler une version surchargée (du constructeur), les argument doivent être placés dans les parenthèses.

```
public class ClasseMere {
    void AfficheText(String unTexte){
        System.out.println("Mère : " + unTexte);
    }
}
```

```
public class ClasseFille extends ClasseMere {
    void AfficheText(String unTexte){
        super.AfficheText(unTexte);
        System.out.println("Fille : " + unTexte);
    }
}
```

```
public class TestHeritage {
    public static void main(String[] args) {
        ClasseFille unObjet;
        unObjet = new ClasseFille();
        unObjet.AfficheText("mot");
    }
}
```

```
Mère : mot
Fille : mot
```

La classe `Object`

- Toutes les classes héritent de `Object` si elle n'héritent pas d'une autre classe. Donc, toutes les classes héritent (de manière direct ou indirect) de la classe `Object`.

- Cette classe comprend quelques méthodes (que l'on surcharge) comme :

    - `toString()` qui est renvoie une chaîne représentant l'objet (très utile pour le "debugage")

    - `clone()` est utilisée pour renvoyer une copie de l'objet.

    - `hashCode()` renvoie un entier qui "représente" l'objet. Deux objets ayant un hashcode identique sont considéré comme identique. Java utilise cette valeur dans les différentes collections (tableaux, listes, ...)

    - `finalize()` est appelée lorsque l'objet est détruit.

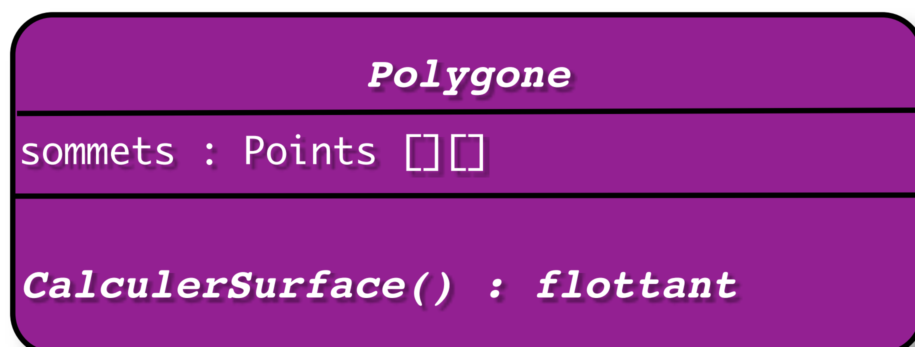
Méthodes abstraites

- Une méthode abstraite (“résumée”) est une méthode qui n’est pas implémentée. Seule l’entête de la méthode est spécifiée puis terminée par un point virgule (donc pas d’accolade).

- La méthode est définie et elle sera implémentée lors d’un héritage.

- Dans un schéma UML, les éléments abstraits sont écrits en italique.

- En Java, le mot clé `abstract` est placé devant le type de retour d’une méthode pour spécifier une méthode abstraite.



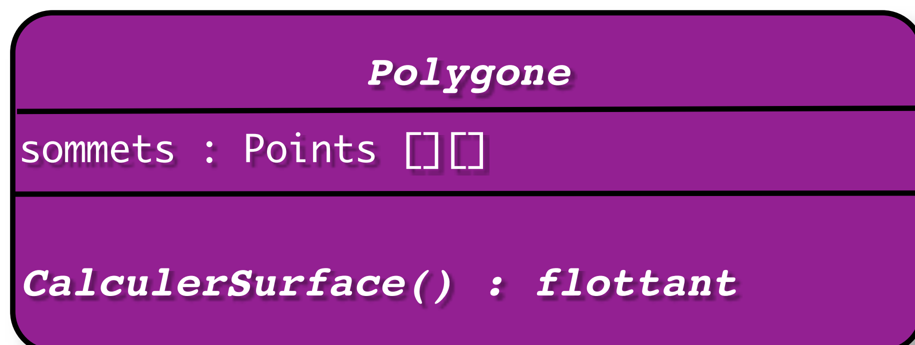

```
... Polygone {
    Points[][] sommets;

    abstract float CalculerSurface();
}
```

Classes abstraites

- Une classe abstraite est une classe qui ne peut pas être instanciée (on ne peut pas créer d'objet de cette classe) ; l'héritage est la seule utilisation de cette classe.

- En Java, toute classe qui contient une ou plusieurs méthode(s) abstraite(s) est une classe abstraite.

- En Java, le mot clé `abstract` est placé devant le mot `class` lors de la déclaration pour définir une classe abstraite.

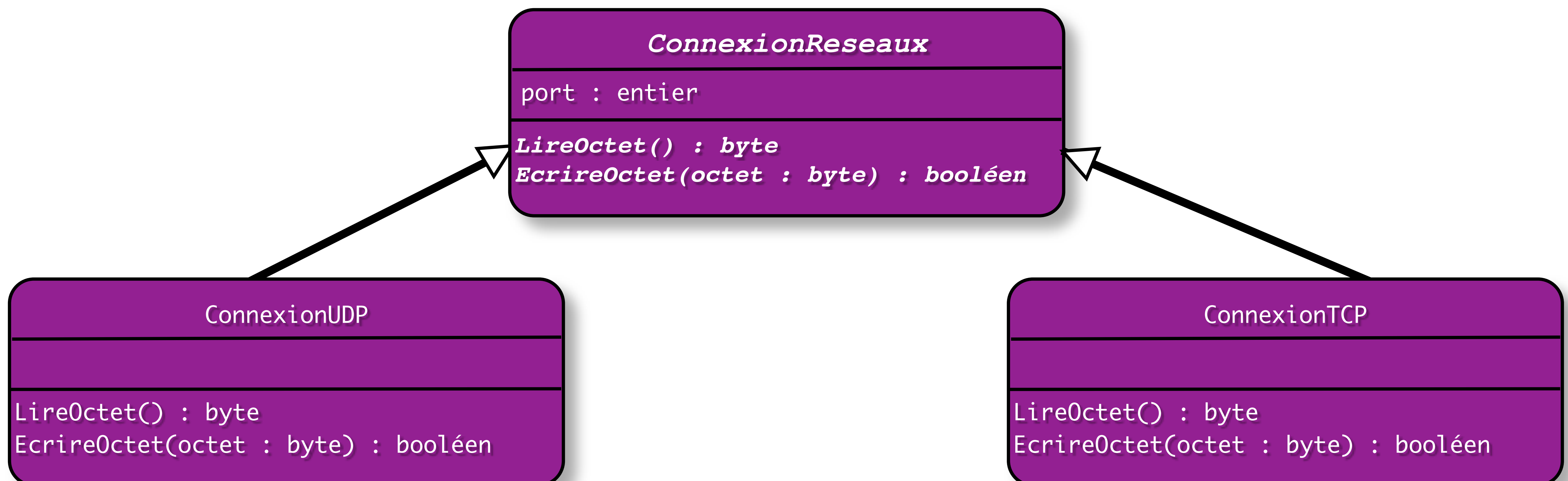



```
public abstract class Polygone {
    Points[][] sommets;

    abstract float CalculerSurface();
}
```

Utilisation des classes abstraites

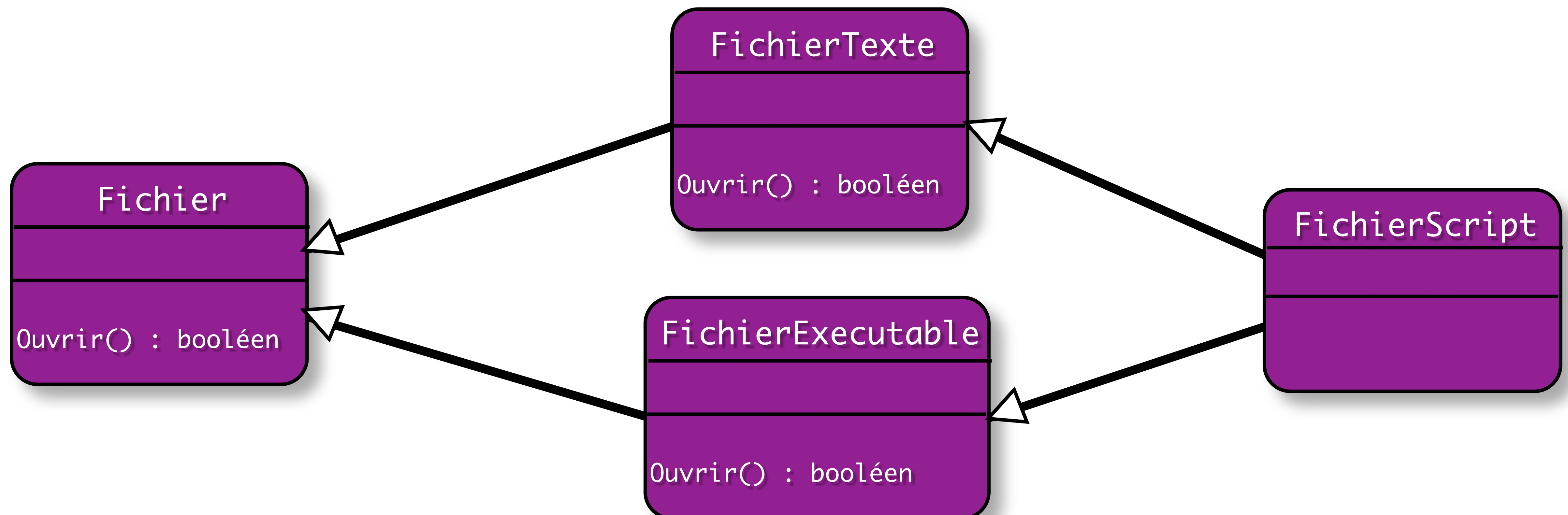
- Les classes abstraites permettent de définir un comportement général qui sera défini dans les classes filles.

- L'impossibilité d'instancier une classe abstraite évite les comportement impossibles (ici, il est impossible d'envoyer des octets sur un réseau sans en connaître le protocole...)



Héritages multiples

- Le mécanisme d'héritage est très puissant mais à certaines limites.
- Certains problèmes conduisent à des héritages multiples souvent difficiles à gérer.
- Java interdit l'héritage multiple contrairement à d'autre langages (comme C++,...) à la place il propose le mécanisme d'interfaces.



Interfaces

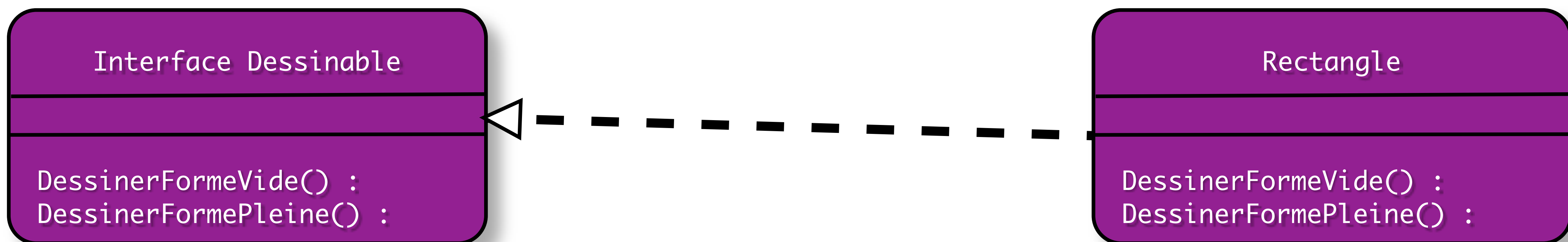
- Un interface est une liste de méthodes non implémentées.
- Une interface est déclarée en utilisant le mot clé `interface`.
- La liste des entêtes de méthodes est placée entre accolades.
- Une interface **ne** peut **pas** contenir d'attribut.

```
Interface Dessinable
----------------------
----------------------
DessinerFormeVide() :
DessinerFormePleine() :
```

```java
public interface Dessinable {
    void DessinerFormeVide();
    void DessinerFormePleine();
}
```

Interfaces

- Les classes peuvent *implémenter* une interface en utilisant le mot clé `implements` suivit du nom de l'interface (ou des interfaces).
- Toutes les méthodes de l'interface (ou des interfaces) doivent être implémentées.
- Dans un schéma UML, c'est une flèche en pointillés qui représente la relation d'implémentation.




```
public interface Dessinable {
    void DessinerFormeVide();
    void DessinerFormePleine();
}
```

```
public class Rectangle implements Dessinable {
    void DessinerFormeVide{
    ...
    }
    void DessinerFormePleine{
    ...
    }
}
```

Classes abstraites ou interfaces ?

- L’idée commune est le regroupement de fonctionnalités entre différents éléments.
- Les classes abstraites permettent de définir des attributs dont les classes filles hériteront.
- Une même classe ne peut hériter que d’une classe mère mais peut implémenter plusieurs interfaces.
- L’utilisation des interfaces n’impose pas une relation d’héritage (qui peut avoir des conséquences...).

Utilisation des classes abstraites et des interfaces

- Les classes abstraites permettent de factoriser une partie du code (exactement comme dans l'héritage "normal").
- Les classes abstraites et les interfaces peuvent être utilisées pour définir un paramètre (ou plusieurs) dans une définition de méthode.
- Le type exact du paramètre n'est pas connu, mais on sait que telle ou telle fonction est implémentée.
- Les interfaces sont très utilisées pour la programmation événementielle (interfaces graphiques, ...)

```
boolean EcrireChaine(InterfaceReseaux eth){
        ...
        eth.EcrireOctet()
        ...
   }
```

Interdire l'héritage

- Il est possible d'interdire l'héritage d'une classe en utilisant le modificateur `final` lors de la déclaration.

- La classe `String` est ainsi spécifiée pour des raisons de sécurité (manipulation de nom de fichiers, ...)

- De même, il est possible d'interdire l'héritage d'une méthode en utilisant le mot clé `final`. Les classes filles ne peuvent plus modifier la méthode.

```
public final class ClasseFinale {
    ...
}
```

```
public class ClasseNonFinale {
    final void Methode(){
        ...
    }
}
```

Visibilité

- La visibilité d'un attribut ou d'une méthode est définie lors de la déclaration. Elle permet d'autoriser (ou non) l'accès des éléments à d'autres classes.

- Quatre visibilités sont possibles :
  - `private` (représenté par - en UML)
  - `protected` (représenté par # en UML)
  - `public` (représenté par + en UML)
  - "*friendly*" qui est la visibilité par défaut si rien n'est spécifié.
- Le modificateur de visibilité doit être placé avant la déclaration de méthode ou d'attribut.

```
...
private int UnAttribut
...
public void UneMethode(double unArgument)
```

Encapsulation

- Les modificateurs de visibilité sont à la base de l'encapsulation.

- L'encapsulation permet de cacher les attributs aux autres classes et de n'y accéder que par des méthodes (les assesseurs ou *getters/setters*).

  - Avant de modifier un attribut, il est possible de vérifier sa validité.
  - La modification de certains attributs peut être interdite ("*lecture seule*").
  - Il est possible de modifier complètement la manière dont est stocké l'information sans changer les méthodes d'accès.

```
public class ConnexionReseau {
    private String IP;
    public String getIP() {
        return IP;
    }
    public void setIP(String uneIP) {
        // Test pour vérifier si l'IP est
valide avant de la modifier
    }
}
```

```
public class ConnexionReseau {
    ...
    private boolean connected;
    ...
    public boolean isConnected() {
        return connected;
    }
}
```

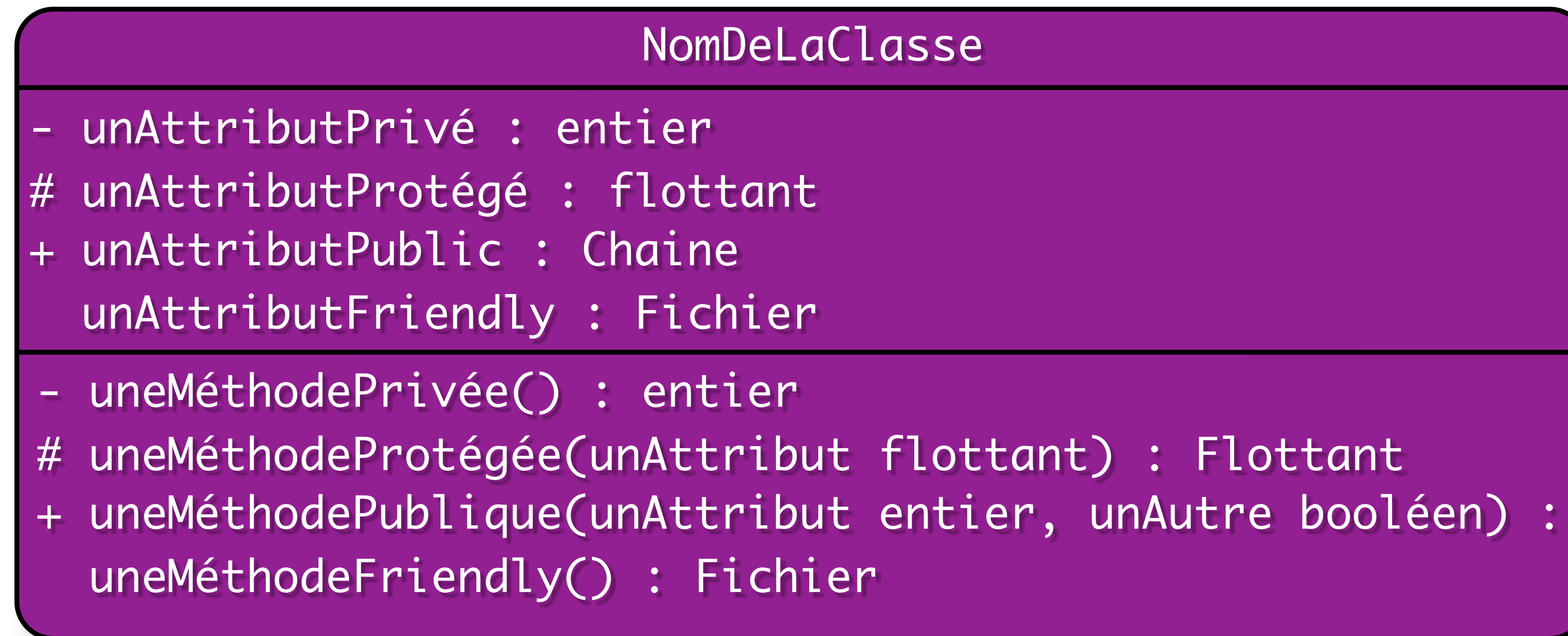
Visibilité

- Le tableau ci dessous résume les différentes combinaisons.

Visibilité	Même classe	Classe fille	Même paquetage	Autre cas
`private`	✓	✗	✗	✗
`protected`	✓	✓	✓	✗
sans (“friendly”)	✓	✗	✓	✗
`public`	✓	✓	✓	✓

- Ces différents modificateurs permettent de définir précisément l’accès aux différents éléments.
- Les éléments non visibles existent quand même, ils sont simplement inaccessibles !
- Un élément déclaré `private` ne pourra donc être lu/écrit que dans la classe ou il est déclaré. Dans toutes les autres classes, les attributs (ou méthodes) déclarés `private` des objets ne seront pas accessibles.

Synthèse du passage de UML à Java




```java
public class NomDeLaClasse {
    private int unAttributPrivé;
    protected float unAttributProtégé;
    public String unAttributPublic;
    Fichier unAttributFriendly;

    private int uneMéthodePrivée(){...}
    protected float uneMéthodeProtégée(float unAttribut){...}
    public void uneMéthodePublique(int unAttribut, boolean unAutre){...}
    File uneMéthodeFriendly(){...}
}
```

Constante en Java

- Il n’a pas de mot clé particulier pour définir une constante en Java.

- A la place, on utilise un attribut statique final publique :
  - le modificateur `static` permet de définir un attribut partagé par toute les instances de la classe.
  - le modificateur `final` empêche la modification de cette valeur.
  - le modificateur `public` permet d’accéder à l’attribut sans problème de visibilité.

```
public static final double PI = 3.141592653589793;
```

Opérateur particulier

- L'opérateur `instanceof` permet de tester l'appartenance à une classe (ou à une classe mère) ou l'implémentation d'une interface.

- Le casting d'une classe vers une autre est possible dans le sens *«qui peut le plus peu le moins»*, il est basé sur les relations d'héritage. Cette possibilité est très utilisée dans les collections (ensemble d'objets).

- A l'intérieur d'une classe, le mot clé `this` permet de référencer l'objet courant (*«moi»*), il est utilisé :

  - pour lever des ambiguïtés entre un nom de variable dans une interface de méthode et un attribut

  - pour désigner l'objet courant (très utilisé dans les auditeurs des interfaces graphiques au prochaine module...)

```
//...
    public void MaFonction (Object unObjet){
    //..
        if (unObjet instanceof MaClasse){
            unObjet.MaMethode();
        }
    //...
}
```

```
//...
    public void MaFonction (Object unObjet){
        MaClasse monObject
        if (unObjet instanceof MaClasse){
            monObjet = (MaClasse) unObjet
        }
    //...
}
```

Et la notation `System.out.println("...")` ???

- Il suffit de tout décomposer pas à pas :
  - `System` n'a jamais été instanciée, c'est donc une classe.
  - `println()` est une méthode (car il y a des parenthèse) appliqué sur l'objet `out`
  - `out` est un attribut (car il n'y a pas de parenthèse) de la classe `System`