De la conception au code

Passage UML : Java

De UML à Java

- Avec la modélisation UML, on identifie :
  - Les classes à utiliser dans le logiciel final
  - Les relations entre ces classes
  - Les méthodes à mettre dans les classes
  - On a une esquisse des algorithmes

- Il ne reste plus qu'à programmer en Java
  - Besoin de préciser public, private, etc.
  - Traduire les relations d'association, d'agrégation et de composition
  - Exploiter les séquences objets

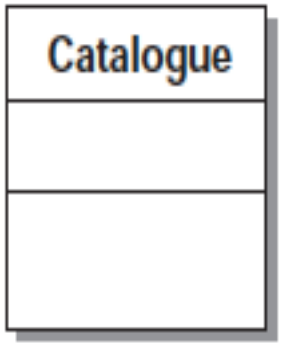
Visibilité : définitions

- La **visibilité** permet de définir précisément quels objets ont le droit d'accéder aux attributs et aux méthodes d'un autre objet
  - Un **attribut** d'une classe C est **visible** depuis une classe C' ssi toute instance de C' peut **lire ou écrire** cet attribut dans toute instance de C
  - Une **méthode** d'une classe C est **visible** depuis une classe C' ssi toute instance de C' peut **invoquer** cette méthode sur toute instance de C

Classes

- Par défaut chaque classe UML devient un fichier .java (Java) ou .cs (C#).

UML	Java

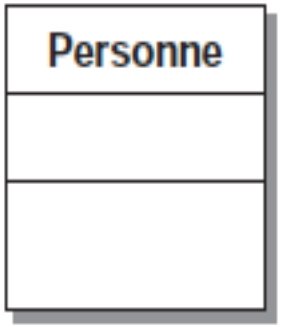
	`public class Catalogue {`
	`  …`
	`}`

	C#

	`public class Catalogue {`
	`  …`
	`}`

Classes abstaites

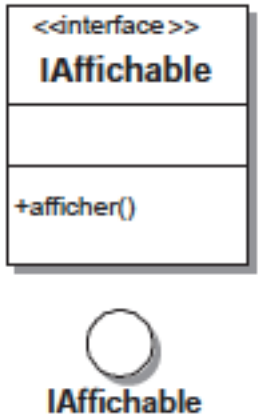
- Une classe abstraite est simplement une classe qui ne s'instancie pas directement mais qui représente une pure abstraction afin de factoriser des propriétés. Elle se note en *italique*

<table>
<tr><th>UML</th><th>Java</th></tr>
<tr><td rowspan="3"> 
</td><td>abstract public class Personne {
…
}</td></tr>
<tr><th>C#</th></tr>
<tr><td>abstract public class Personne {
…
}</td></tr>
</table>

Interface

- La notion UML d'interface (représentée sous ses deux formes graphiques) se traduit par le mot-clé correspondant aussi bien en Java qu'en C#.

UML	Java

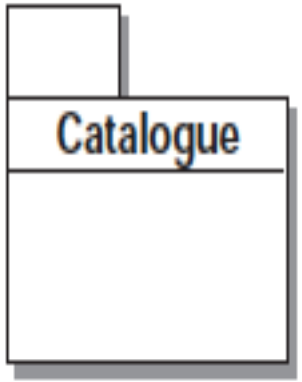
	`interface IAffichable {`
	`  void afficher();`
	`}`

C#	

`interface IAffichable {`
`  void Afficher();`
`}`	

Package

- Le package en tant que regroupement de classes ou d'interfaces existe aussi bien en Java qu'en C#, mais avec une syntaxe différente.

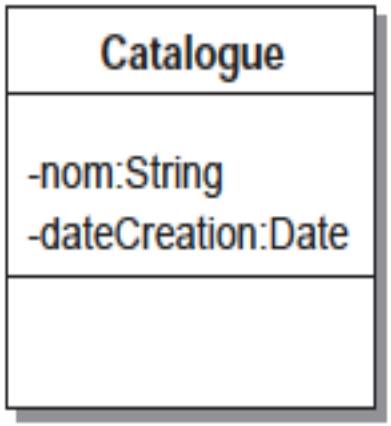
UML	Java
	`package catalogue;` …
	C#

	`namespace Catalogue`
	`{`
	…
	`}`

Attributs

- Les attributs deviennent des variables en Java et en C#
- Leur type est soit un type primitif (int , etc.), soit une classe fournie par la plate-forme (String, Date, etc.). Attention à ne pas oublier dans ce cas la directive d'importation du package correspondant.
- La visibilité des attributs est montrée en les faisant précéder par + pour public, # pour protégé (protected), - pour privé (private).
- Les attributs de classe en UML deviennent des membres statiques en Java ou en C#.
- Les attributs de type référence à un autre objet ou à une collection d'objets sont discutés dans la section « Association ».

UML



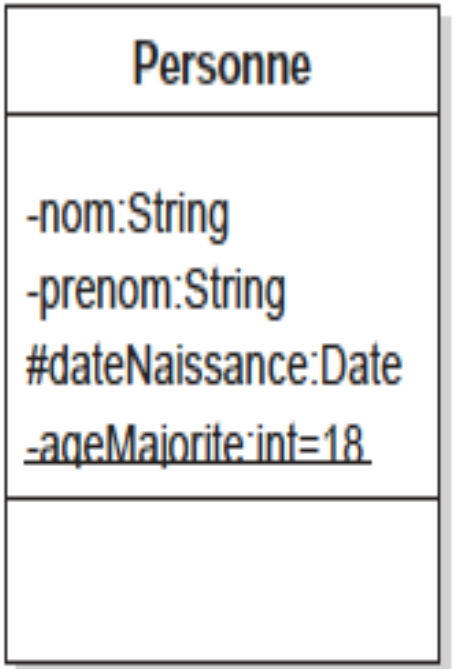
Java

```
import java.util.Date;

public class Catalogue {
    private String nom;
    private Date dateCreation;
    …
}
```

C#

```
using System;

public class Catalogue {
    private string nom;
    private DateTime dateCreation;
    …
}
```

UML



Java

```
abstract public class Personne {
    private String nom;
    private String prenom;
    protected Date dateNaissance;
    private static int ageMajorite = 18;
}
```

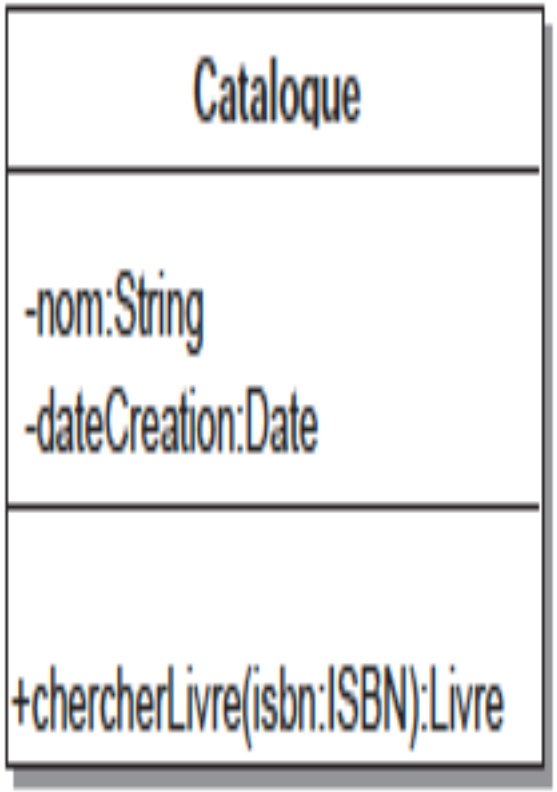
C#

```
abstract public class Personne {
    private string nom;
    private string prenom;
    protected DateTime dateNaissance;
    private static int ageMajorite = 18;
}
```

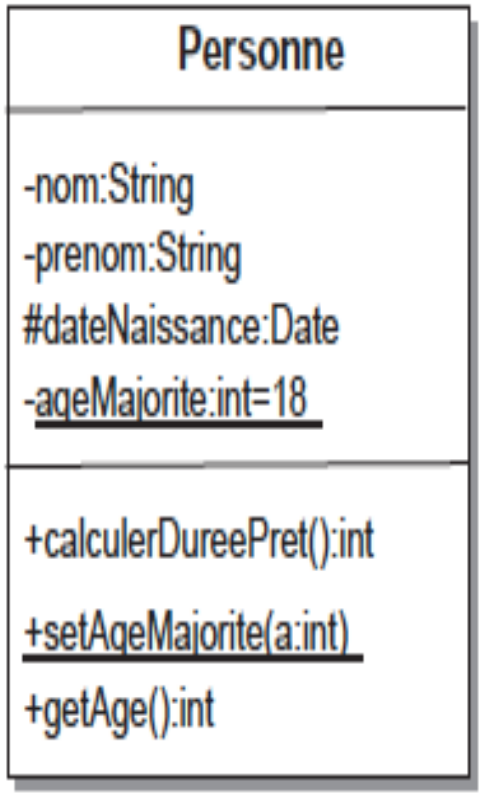
Opérations

- Les opérations deviennent des méthodes en Java et en C#.
- Leur visibilité est définie avec les mêmes conventions que les attributs.
- Les opérations de classe deviennent des méthodes statiques ; les opérations abstraites (en italique) se traduisent par le mot-clé correspondant en Java ou en C#.

UML	Java

	public class Catalogue {
	private String nom;
	private Date dateCreation;
	public Livre chercherLivre(ISBN isbn) {
	...
	}
	...
	}

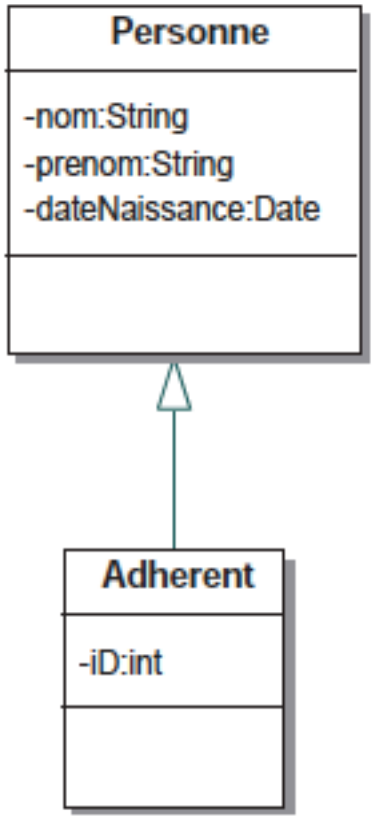
UML	Java

	abstract public class Personne {
	private String nom;
	private String prenom;
	protected Date dateNaissance;
	private static int ageMajorite = 18;
	public abstract int calculerDureePret();
	public static void setAgeMajorite(int aMaj) {
	…
	}
	public int getAge() {
	…
	}
	}

Les relations

- Généralisation : héritage

UML



Java

```
public class Adherent extends Personne {
    private int iD;
}
```

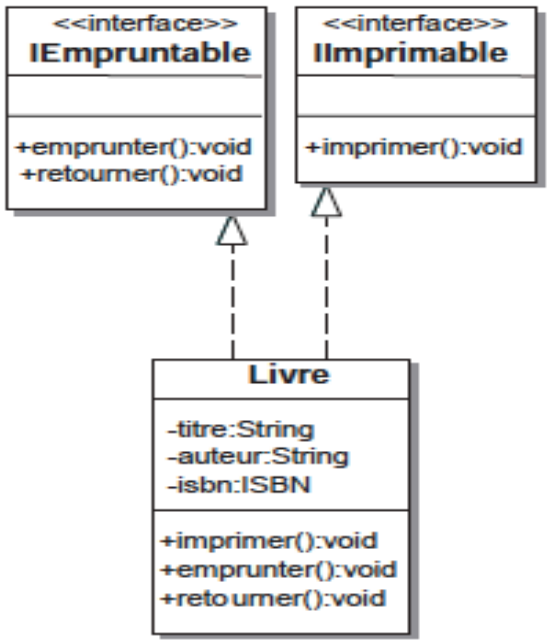
C#

```
public class Adherent : Personne {
    private int iD;
}
```

Réalisation : interfaces

- Une classe UML peut implémenter plusieurs interfaces. Les langages Java et C# proposent directement ce mécanisme, mais pas l'héritage multiple entre classes.
- C# utilise la syntaxe du C++ pour l'héritage. Le même mot-clé s'utilise pour l'héritage d'implémentation et d'interface contrairement à Java qui spécifie extends et implements
- .

UML



Java

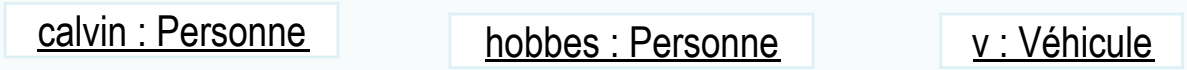
```
public class Livre implements
 IImprimable, IEmpruntable {
    private String titre;
    private String auteur;
    private ISBN isbn;
    public void imprimer(){
    …
}
    public void emprunter(){
    …
}
    public void retourner(){
    …
 }
}
```

C#

```
public class Livre :
 IImprimable, IEmpruntable {
    private string titre;
    private string auteur;
    private ISBN isbn;
    public void Imprimer(){
    …
}
    public void Emprunter(){
    …
}
    public void Retourner(){
    …
 }
}
```

Illustration de la visibilité

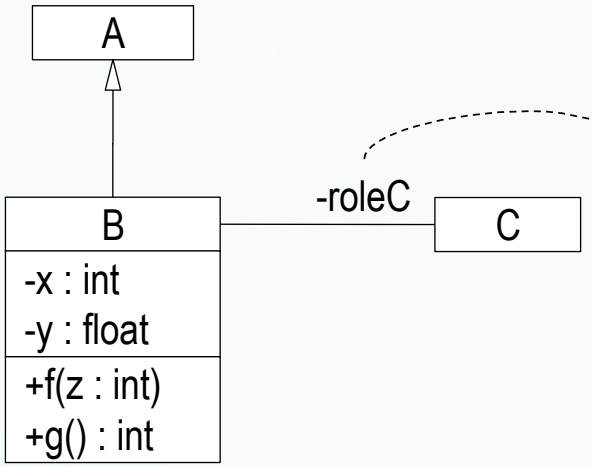
- Reprenons la relation Personne / Véhicule



- Soit calvin et hobbes deux instances de Personne et v une instance de Véhicule
  - calvin à le droit d'accéder à :
    - hobbes.nom, hobbes.age, hobbes.manger(), hobbes.dormir()
    - v.rouler()
    - pas v.immatriculation
  - etc.

Traduction des classes

- Seul piège : les attributs de la classe Java sont ceux de la classe UML + les rôles des classes associées




```
public class B extends A {
    private int x;
    private float y;
    private C roleC;

    public void f(int z) {
        …
    }
    public int g() {
        … return …
    }
}
```

Traduction des associations et navigation

- Une association UML est bidirectionnelle : les deux rôles se traduisent par des attributs dans les classes Java


```
public class A {
    private B roleB;
}
```

```
public class B {
    private A roleA;
}
```


```
public class A {
    private B roleB;
}
```

```
public class B {
    // il n'y a plus de A
}
```

- On peut rendre l'association unidirectionnelle avec une flèche de **navigation** :

Traduction des compositions

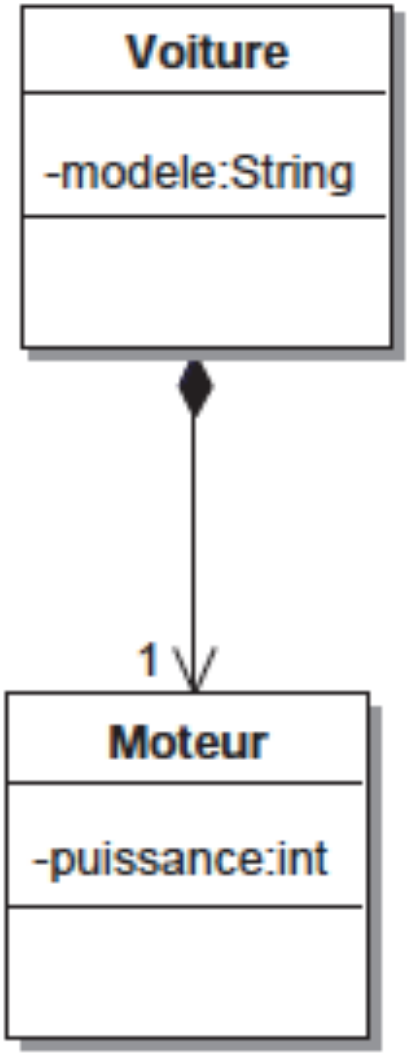
- On peut souvent les ramener à des relations unidirectionnelles


```
public class A {
    private B roleB;
}
```

```
public class B {
    // il n'y a plus de A
}
```

- Règle à respecter en Java : « la destruction du tout entraîne celle des parties »
  - Les composants doivent être créés dans le constructeur de A
  - Aucune méthode de A ne doit renvoyer de référence de roleB
  - Sinon le ramasse-miettes ne détruit pas roleB quand il détruit A

UML	Java

	public class Voiture {
	private String modele;
	private Moteur moteur;
	private static class Moteur {
	private int puissance;
	}
	…
	}

Traduction des multiplicités *

- Relations 1-vers-plusieurs


```
public class A {
    private B[] roleB;
}
```

```
public class B {
    private A roleA;
}
```

- roleB peut être un tableau de B
- Mais aussi une liste de B, un dictionnaire, etc.
  - voir cours à venir sur les structures de données (Java Collections)
- Possibilité de limiter la relation de manière unidictionnelle

Traduction des multiplicités *

- Relations plusieurs-vers-plusieurs

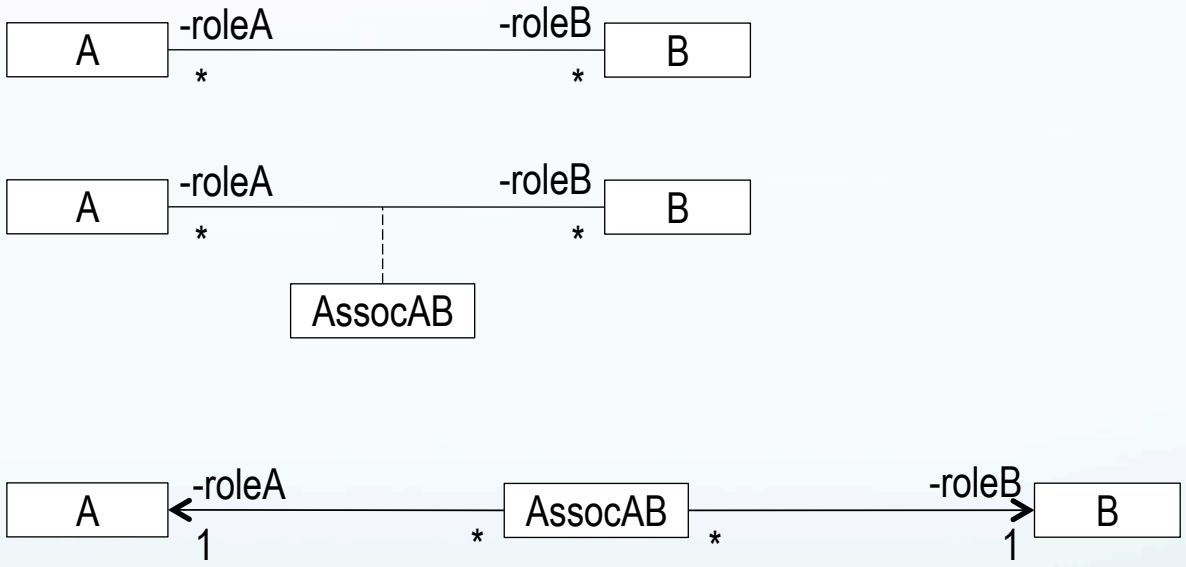

```
public class A {
    private B[] roleB;
}
```

```
public class B {
    private A[] roleA;
}
```

  - Ici aussi, possibilité d'utiliser des listes plutôt que de simples tableaux

Traduction des multiplicités *

- Relations plusieurs-vers-plusieurs
  - Possibilité d'introduire une **classe d'association**



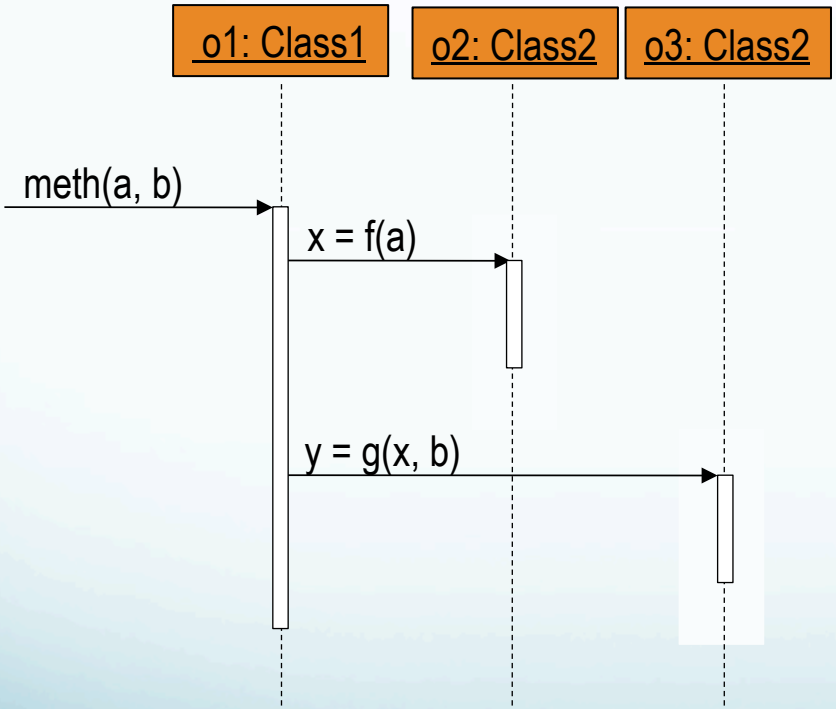

```
public class A {
   // plus de roleB
}
```

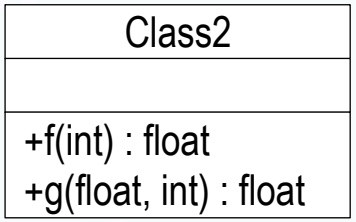
```
public class AssocAB {
   private A roleA;
   private B roleB;
}
```

```
public class B {
   //plus de role A
}
```

Traduction des séquences objets

- Les messages reçus par un objet correspondent à des méthodes de sa classe

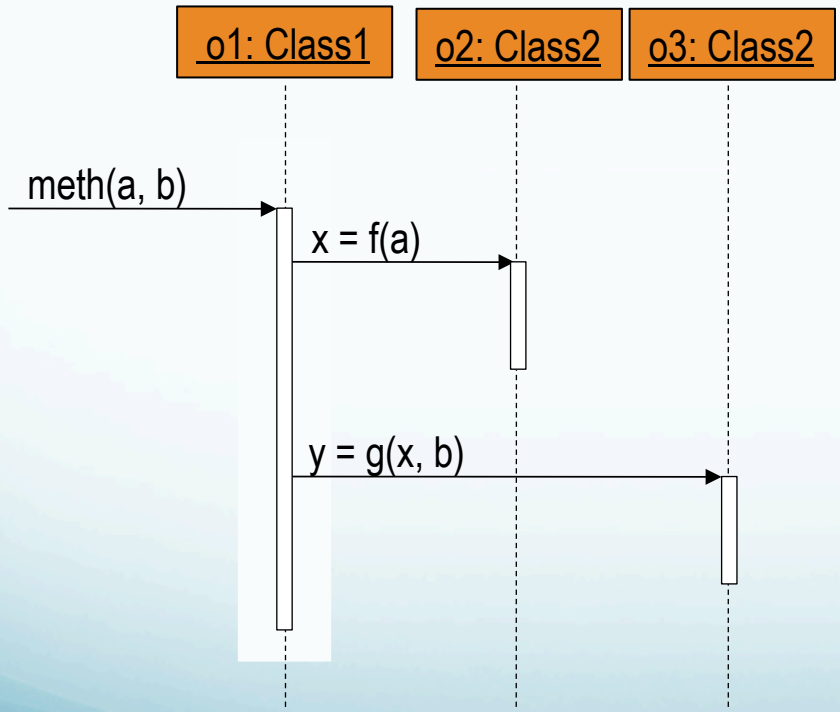


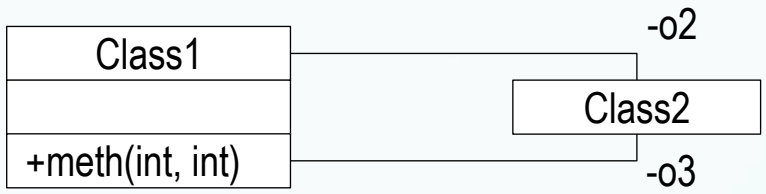



```
public class Class2 {
    public float f(int m) {
        … return …
    }
    public float g(float z, int n) {
        … return …
    }
}
```

Traduction des séquences objets

- La séquence objet permet aussi de donner un aperçu de l'algorithme d'une méthode






```
public class Class1 {
    private Class2 o2;
    private Class2 o3;
    public void meth(int m, int n) {
        float x = o2.f(m);
        float y = o3.g(x, n);
        …
    }
}
```

Bilan

- En partant des cas d'utilisation et des classes du domaine on écrit des diagrammes de séquence objets
- Ces séquences objets permettent de sélectionner et d'affiner les classes qui resteront dans le diagramme de classes final
- Le diagramme de classes final est prêt à programmer directement en Java