

Architecture logicielle

POO – UML – Design Pattern

11 cours
1 partiel
1 contrôle intermédiaire

Objectifs :

- Acquisition des notions et du vocabulaire de la POO
- Séparer l'apprentissage des langages de celui des notions



POO, Design et UML : introduction

- La programmation = *Analyse* de problèmes, puis *conception* de solutions => *codage* et *maintenance* d'une application informatique de traitement de données.
- Programmation Orientée Objet : *méthode* de programmation dans laquelle les *données* et les *traitements* sont *intégrés* dans des composants, des briques nommés **Objets**, représentant des idées, des concepts et/ou des entités du monde réel.

Introduction suite...

Design pattern: (Patrons, Modèles de conception) Façon de résoudre le problème rencontré. Des solutions éprouvées, validées existent et doivent être appliquées dans des cadres bien définis. Ce n'est pas un algorithme, c'est une architecture, une trame de solution générique.

UML : langage graphique permettant d'échanger avec les autres intervenants, modélisant le comportement attendu, et les solutions mises en place.

Introduction fin

- Bonnes pratiques : Principales méthodes et organisations utilisées pour les architectures, méthodes AGILES, etc...

Introduction à la Programmation Orientée Objet

La solution procédurale

Afin de traiter les données, on appréhende le problème en raisonnant de façon logique, *d'un état initial vers un état final*.

Les données *entrent* dans l'algorithme, on *applique* les actions définies, et on *termine*.

La programmation procédurale tend à une capitalisation dans l'écriture du programme. On tente de trouver des parties qui se répètent, identiques, de trouver des comportements types. Ce seront des **procédures** et **fonctions**, qu'on pourra stocker dans des **bibliothèques**, puis réutiliser.

Les défauts du procédural

Principe de Wirth (inventeur du langage Pascal) :
Programme = algorithme + structure de données

=> **Forte dépendance** du programme aux données

=> **Peu de réutilisabilité**

=> **Peu d'interfaçage** entre applications

=> **Collaboration difficile** entre programmeurs...

=> **Evolution difficile**, effet de bords, régressions...

L'idée de la POO

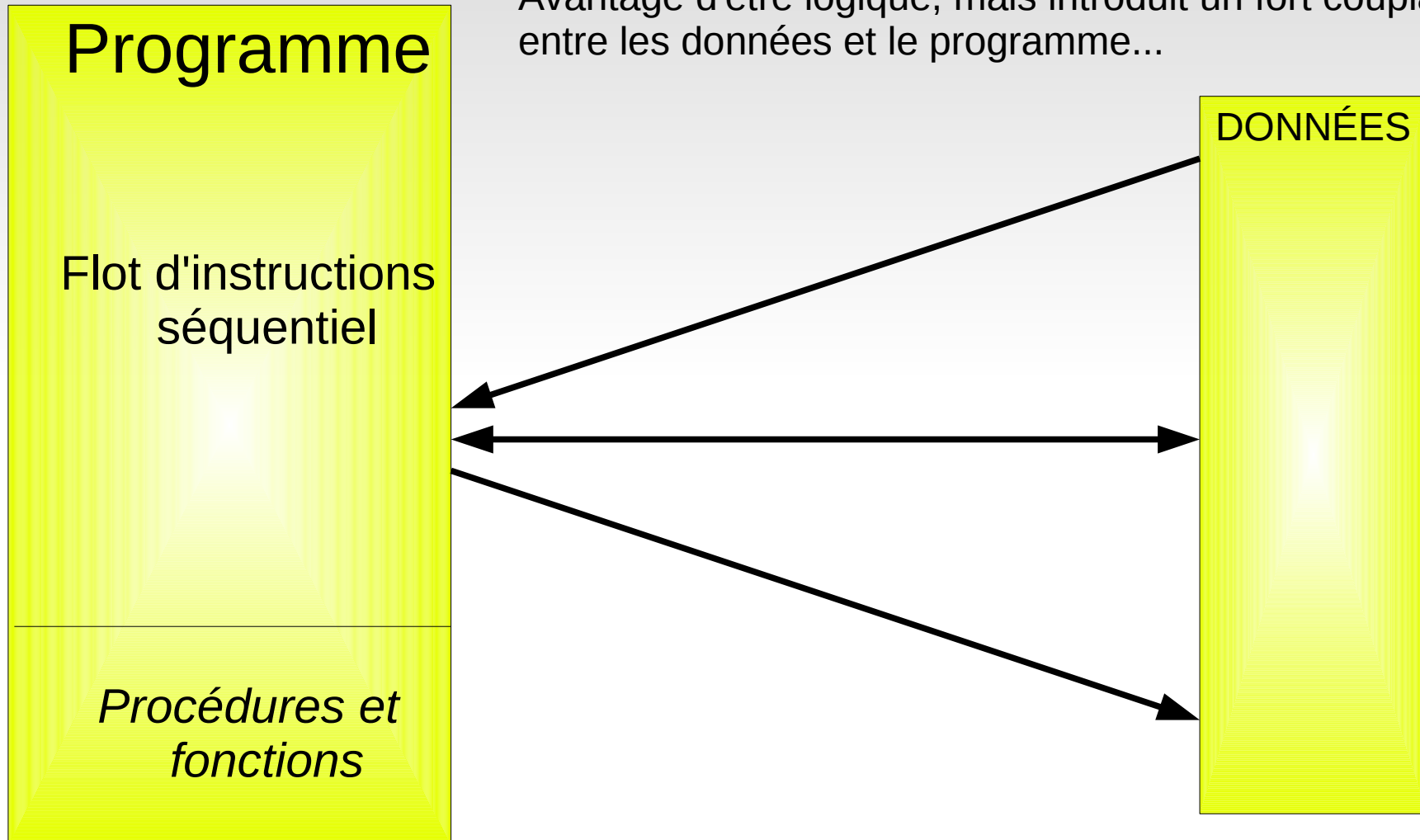
la Programmation Orientée Objet, c'est la tentative de réunir **les données ET les traitements** en une unité cohérente et maintenable. Dans le monde réel, tout est à la fois données et traitements...

Une facture contient à la fois des données, et des calculs associés...

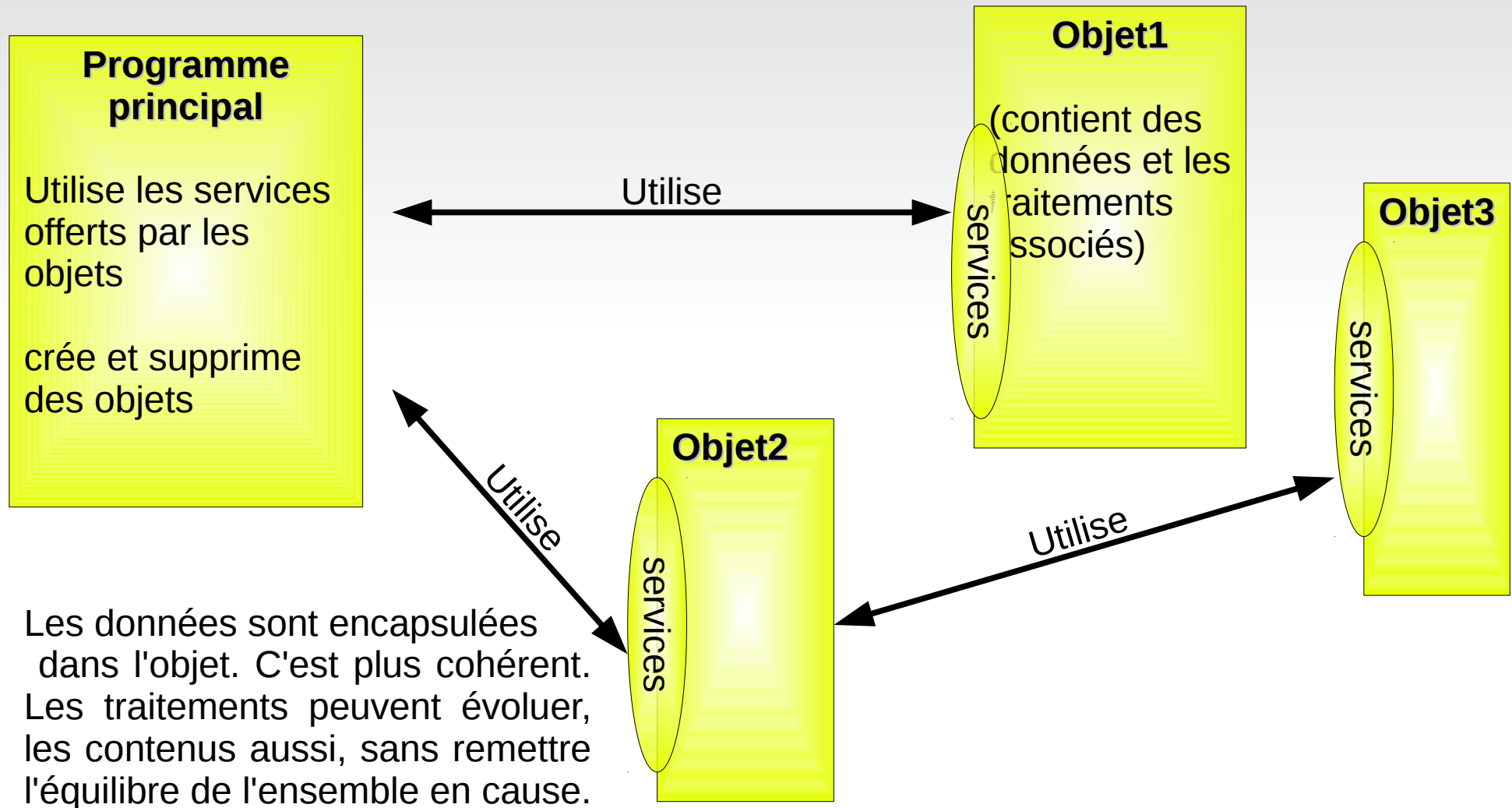
Toute chose est à la fois dotée de caractéristiques (des données, des valeurs) et des comportements (des actions, des réactions à des stimulus, des messages).

Procédural/structuré

Avantage d'être logique, mais introduit un fort couplage entre les données et le programme...



Programmation Objet



Problèmes des logiciels

La complexité :

- plus elle *grandit*, plus le système est *fragile*
- la complexité peut *dépasser* les capacités intellectuelles de l'homme
- Il faut mettre en place des méthodes *rigoureuses* pour se *protéger* de l'effondrement de l'ensemble
- Il faut pourtant *permettre l'évolution* de l'ensemble
- il faut *masquer* cette *complexité* à l'utilisateur

la conception

Comment concevoir un système ?

Le principe de base : *La décomposition...* Un problème est transformé en sous-problèmes, etc etc... vers une granularité de plus en plus fine.

Approche Structurée :

Orientée traitement : Raisonnement principal, modules fonctionnels, découpage progressif.

Approche OBJET :

Conception d'unités autonomes chargées de la gestion des données qu'elles comportent. C'est l'encapsulation. On délègue à quelque chose la gestion en détail → il en a la responsabilité...

L'approche Objet





Une **Classe** est une abstraction permettant de modéliser des éléments du monde réel. Un objet n'existe pas de façon isolée. Il offre des *services*, et utilise des *services*.

Les *services* sont le miroir de l'activité de l'objet. Par exemple, un objet "Compte en banque" permet de connaître le solde, le nom du titulaire, d'obtenir son rib.. On peut effectuer un crédit, un débit sur cet objet, obtenir la liste des opérations

L'objet est plus qu'un regroupement de données et de traitement : il est *cohérent*, modèle exact et complet de la perception de l'utilisateur dans le référentiel de son utilisation (il n'est cependant pas universel... Un compte en banque, ce n'est pas la même chose pour un particulier et pour son banquier. Il est cohérent pour celui qui la conçu)

Des objets et leurs services.

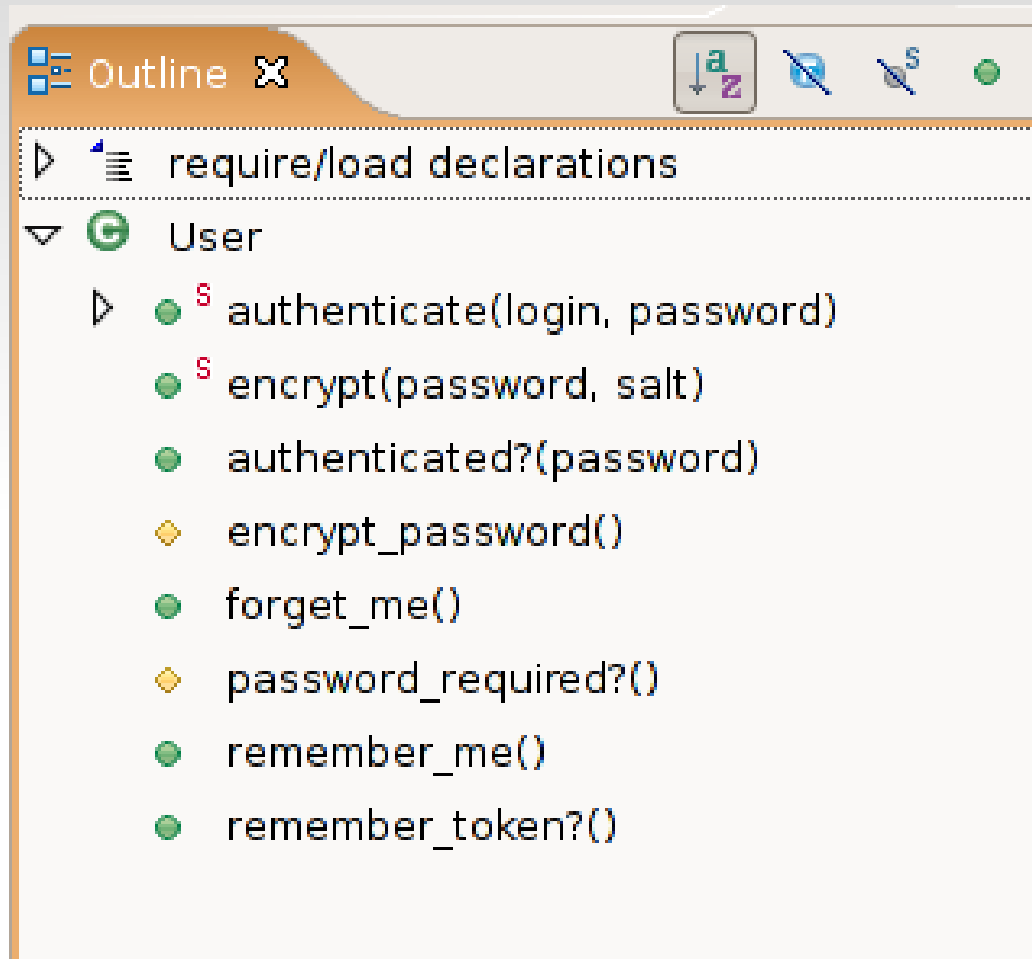
2	Amusement Parks On Fire	venus in cancer	Amusement Parks On Fire	3:37
9	the arcade fire	rebellion(lies)	title	5:10
9	the arcade fire	rebellion(lies)	title	5:10
4	Art Brut	Fomed a band	Bang bang Rock'n'Roll	2:59
1	BATTLE	Tendency	Unknown Album (20/12/2...	4:22
1	Belle & Sebastian	Step Into My Office, Baby	Dear Catastrophe Waitress	4:12
2	Belle & Sebastian	Dear Catastrophe Waitress	Dear Catastrophe Waitress	2:22
3	Belle & Sebastian	If She Wants Me	Dear Catastrophe Waitress	5:05
4	Belle & Sebastian	Pizza, New York, Catcher	Dear Catastrophe Waitress	3:03
5	Belle & Sebas		Dear Catastrophe Waitress	3:22
6	Belle & Sebas		Dear Catastrophe Waitress	5:26
7	Belle & Sebas		Dear Catastrophe Waitress	3:08
8	Belle & Sebas		Dear Catastrophe Waitress	3:34
9	Belle & Sebas			
10	Belle & Sebas			
11	Belle & Sebas			
12	Belle & Sebas			
1	Björk			
2	Björk			
3	Björk			
4	Björk	vokuro		
5	Björk	Öll Birtan		
6	Björk			

Colonnes...
Ajouter à la liste de lecture
Note
 Copier
 Supprimer
Supprimer du disque
 Recherche de morceaux
Tout sélectionner
Aucune sélection
 Éditer les informations du morceau

Dans un gestionnaire de musique (ici Banshee), les méthodes accessibles avec un clic droit. Il y a des services offerts par le morceau et d'autres offerts par la liste de lecture...

Le concept d'Objet offre une manipulation plus intuitive, puisque comme dans la vie courante, les fonctionnalités sont offertes par les objets eux-mêmes...

Des objets et leurs services.



Dans l'outil de développement Eclipse, un objet écrit en Ruby nous présente ses méthodes. Il s'agit dans cet exemple d'un objet permettant d'implémenter l'authentification. Les noms des méthodes sont assez explicites. Les couleurs indiquent les restrictions d'accès à ce service (vert = pour tous, jaune = limité)

Un objet, qu'est ce que c'est ?

Un objet, qu'est ce que c'est ?

Un objet est un ensemble de codes informatiques qui représente à la fois des procédures et des données.

En Objet, la procédure s'appelle souvent une **méthode**, et une donnée, un **attribut**.

- *Un attribut* est souvent typé (par exemple un entier, un booléen, un Objet), et porte un nom.
- *Une méthode* est aussi typée (c'est à dire qu'elle renvoie quelque chose de typé, ou bien elle ne renvoie rien), porte un nom et peut attendre des arguments.

Des attributs...

Dans cet exemple de code Java, l'image que le programmeur se fait d'un `MicroProcesseur`...

Ce *Processeur* dispose d'un *compteur ordinal*, sous forme d'entier, d'un *accumulateur* (entier aussi), et d'une chaîne de caractère *Etat*.

Il dispose de 5 *registres* (en gros, des emplacements pour stocker des valeurs qui sont ici des entiers).

Un `boolean` permet de passer ce processeur en mode bavard.

```
public class Processeur {  
    boolean verbose=false;  
    int compteurOrdinal=0;  
    int accumulateur=0;  
    String etat=" ";  
    int registre[] = new int[5];  
    ...  
}
```

...et des méthodes

...Puis les méthodes : Ce processeur (du moins, tel que le perçoit ce programmeur, offre un ensemble de fonctionnalités, de comportements : des méthodes..).

Par exemple, définir (set) un attribut (Etat en l'occurrence), récupérer des valeurs (celle de l'accumulateur ici), ou extraire l'adresse mémoire indiquée dans une instruction récupérée par le processeur...

```
public class Processeur {  
    .... (contenu de la page précédente)  
    public int adresseMem(String argument) {  
        /* cette methode se charge de récupérer l'adresse mémoire */  
        String adresse=argument.substring(1);//on récupere après $  
        return(Integer.parseInt(leResultat));  
    }  
  
    public void setEtat()    {etat = ""; }  
  
    public void setEtat(String valeur)    { etat = etat + "-" +valeur;    }  
  
    public int getAccumulateur()    {return accumulateur;    }  
    ....  
}
```

Structure d'un programme objet

Dans la programmation structurée, le problème posé est décomposé jusqu'à obtenir un bloc d'instructions, parfois capitalisé sous forme de procédure ou de fonction.

Dans la démarche objet, la décomposition conduit à concevoir des objets

Le logiciel s'articulera autour d'objets, qui inter-agiront les uns avec les autres. **Le programme est une structure d'objets qui collaborent entre eux.**

Lors de l'exécution du programme, les objets sont créés, et établissent entre eux des liens selon les services qu'ils offrent. Ils seront modifiés, détruits, etc..

Avantages/Inconvénients

Les objets peuvent être *améliorés*, *corrigés*, *mis à jour* sans perturber le fonctionnement de l'ensemble.. Certains langages acceptent même des modifications dynamiques des objets pendant le déroulement de l'application.

Il existe parfois des mécanismes qui permettent à un objet *d'inspecter* un autre objet afin d'en **découvrir** les méthodes offertes.

Par contre, le problème rencontré par les débutants en objet est souvent lié à la *perte apparente de la logique séquentielle* de l'application.

Concept objet : la Classe

Dans les langages Objet, le programmeur va souvent utiliser ou créer des **Classes** :

La **Classe** est l'usine qui engendre les objets. La **Classe** est le patron (le modèle) qui définit les attributs et les methodes d'un ensemble d'objets identiques.

La **Classe est comme une chaine de montage, qui fabrique les objets**. Ainsi, tous les objets fabriqués ont le même comportement. Mais ils auront des attributs spécifiques à chacun, et une vie propre.

Ainsi, la chaine de fabrication des Scénics produit des voitures ayant toutes le même comportement, mais pas la même couleur, pas le même propriétaire, et qui ne seront pas utilisées de la même façon (fréquence, type de conduite, décoration...).

Plein d'objets issus d'une Classe...

Le processus de fabrication d'un objet s'appelle **l'instanciation**.

On **instancie** donc un objet (spécifique, précis, celui là et pas un autre) à partir de la classe (qui elle est générique)..

C'est la chaine de montage de Voiture (la Classe) qui a produit Ma Voiture (la mienne, celle là, précisément). Elle ressemble aux autres... mais c'est la mienne..

Instancier = construire un objet à partir de la classe (via le **constructeur**, méthode d'instanciation de l'objet, qui va donc le fabriquer : cet objet aura les méthodes attendues, et les attributs aussi. Cependant, les valeurs de ses attributs lui seront propres).

Un objet, sa naissance, sa
mort....
Et entre les deux...

L'instanciation : le constructeur

Il s'agit de générer un *objet* du type décrit par la *classe*. La méthode qui se charge de cette instanciation s'appelle le **constructeur**.

Exemple en Java

```
maVoiture=new Voiture("rouge");  
saVoiture=new Voiture("bleue");
```

Exemple en Ruby

```
maVoiture=Twingo.new('rouge')  
saVoiture=Twingo.new('bleue')
```

Fin de vie : le destructeur

Lorsque l'objet devient inutile, il doit disparaître. Certains langages sont capables de s'apercevoir de l'inutilité d'un objet (il n'est plus référencé, ceux qui s'en servaient n'ont gardé aucun lien sur lui), et le font disparaître de la mémoire (principe du *Garbage Collector*, qui passe régulièrement, en tâche de fond, pour faire le ménage).

Il est cependant possible de fournir, et même d'utiliser un *destructeur* (en C++ par exemple, mais plus rare si on a un GC), méthode qui va détruire l'instance de l'objet, en implémentant certainement des traitements (vérification, logs, etc).

Se servir d'un objet

Un fois l'objet instancié, ses **méthodes** seront accessibles, souvent grâce à la **notation pointée**.

(nota : on pourrait aussi essayer d'accéder à ses attributs, mais cela viole le principe d'encapsulation)

Il s'agit d'un message envoyé à l'objet, qui nous répond alors.

Exemple en Java

```
maVoiture.accelere(30);
```

```
bornes=saVoiture.getKilometres();
```

```
maVoiture.depasse(saVoiture);
```

```
//notez ici un objet qui travaille avec un autre objet
```

Pourquoi protéger les attributs ? *(ou comment introduire les méthodes de façon discrète...)*

Pour assurer la pertinence des données !

Si on passe par une méthode, celle-ci va assurer un nombre plus ou moins important de vérifications afin de contrôler la validité de cette donnée. L'exemple typique concerne les vérifications de date..

//accès direct à l'attribut

~~*maVoiture.dateAchat='24/11/1970';*~~ *NON*

//définition de la valeur de l'attribut via une méthode

maVoiture.setDateAchat('24/11/1970'); Mieux

Exemple de méthode setter

L'accès aux attributs est très fortement protégé par des méthodes d'accès, qu'on appelle **getter et setter...**

La méthode `void setDateAchat(String laDate)` (méthode qui ne renvoie rien, et attend une chaîne de caractères) fonctionnera certainement de la façon suivante:

- `my_phone.setDateAchat('bidule');` refusera (pas une date)
- `my_phone.setDateAchat('31/02/2004');` refusera (impossible)
- `my_phone.setDateAchat('17/05/2034');` refusera (date future)
- `my_phone.setDateAchat('17/05/2014');` fonctionnera (vraie date, plausible)

L'encapsulation

On voit l'intérêt des méthodes, qui permettent d'implémenter des procédures de vérification. Si le domaine de validité de l'attribut évolue, il suffit de faire évoluer la méthode, et l'impact sur l'ensemble de l'application est quasi-nul.

C'est ce principe qui pousse à ne pas directement travailler avec les attributs, mais plutôt avec les méthodes correspondantes (le getter qui renvoie la valeur de la variable, et le setter qui permet de la définir).

On dit qu'on sépare **l'interface** (publique, accessible...C'est ce qu'on verra, ce qui est en contact avec l'extérieur) de **l'implémentation** (le code, inaccessible et évolutif)

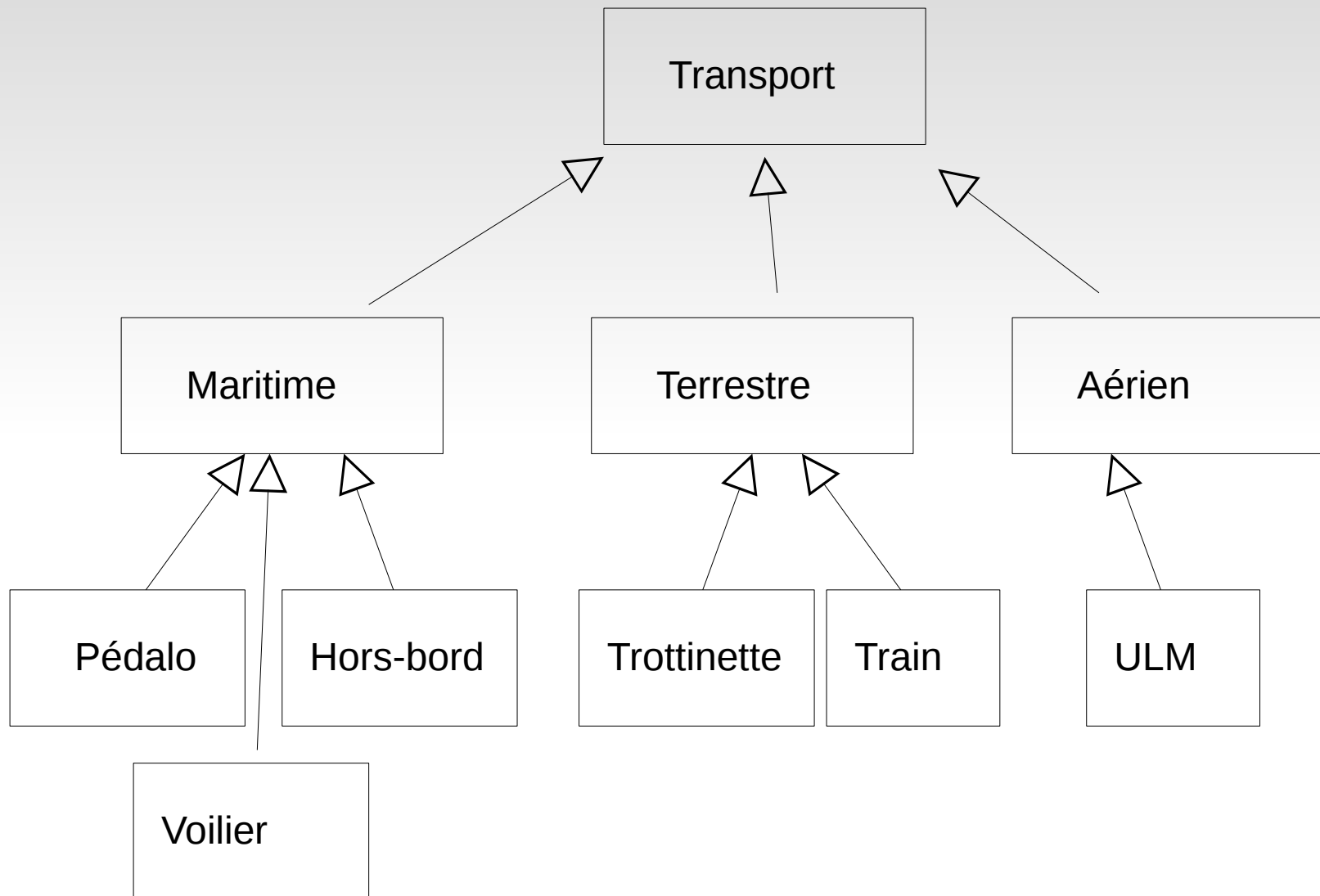
Des objets qui se ressemblent ?

L'héritage

Deuxième concept associé à la programmation objet : c'est la **généralisation/spécialisation**.

L'héritage permet la construction d'une arborescence de Classes en allant vers une description de plus en plus fine (une **spécialisation**).

Chaque Sous-Classe 'hérite' des caractéristiques de sa (ses) super-Classe(s) (*méthodes et attributs*), tout en l'étendant (*parfois très peu, parfois beaucoup*).



L'héritage 2

Ce mécanisme permet à la fois une **écriture élégante** (*un sous-Classe ne va décrire que ses différences par rapport à ses ancêtres : Code plus court, plus lisible, compact*) tout en **permettant un traitement générique** (*on pourra traiter toutes sortes de sous-Classes différentes pourvu qu'on s'intéresse à leur ancêtre commun et à un attribut/une méthode de l'ancêtre : **comportement générique***).

Héritage simple, héritage multiple

Selon les langages, une classe peut dériver (**hériter**) d'une ou de plusieurs super-Classes. Dans le cadre de **l'héritage simple**, on obtient une arborescence de Classes, ce qui est facile à appréhender.

D'autres langages permettent **l'héritage multiple (C++)** (une classe a plusieurs ancêtres, apportant certains avantages de transversalité, mais compliquant un peu les choses (*conflits possibles de noms de méthodes et d'attributs, d'où des comportements plus difficiles à conceptualiser*))

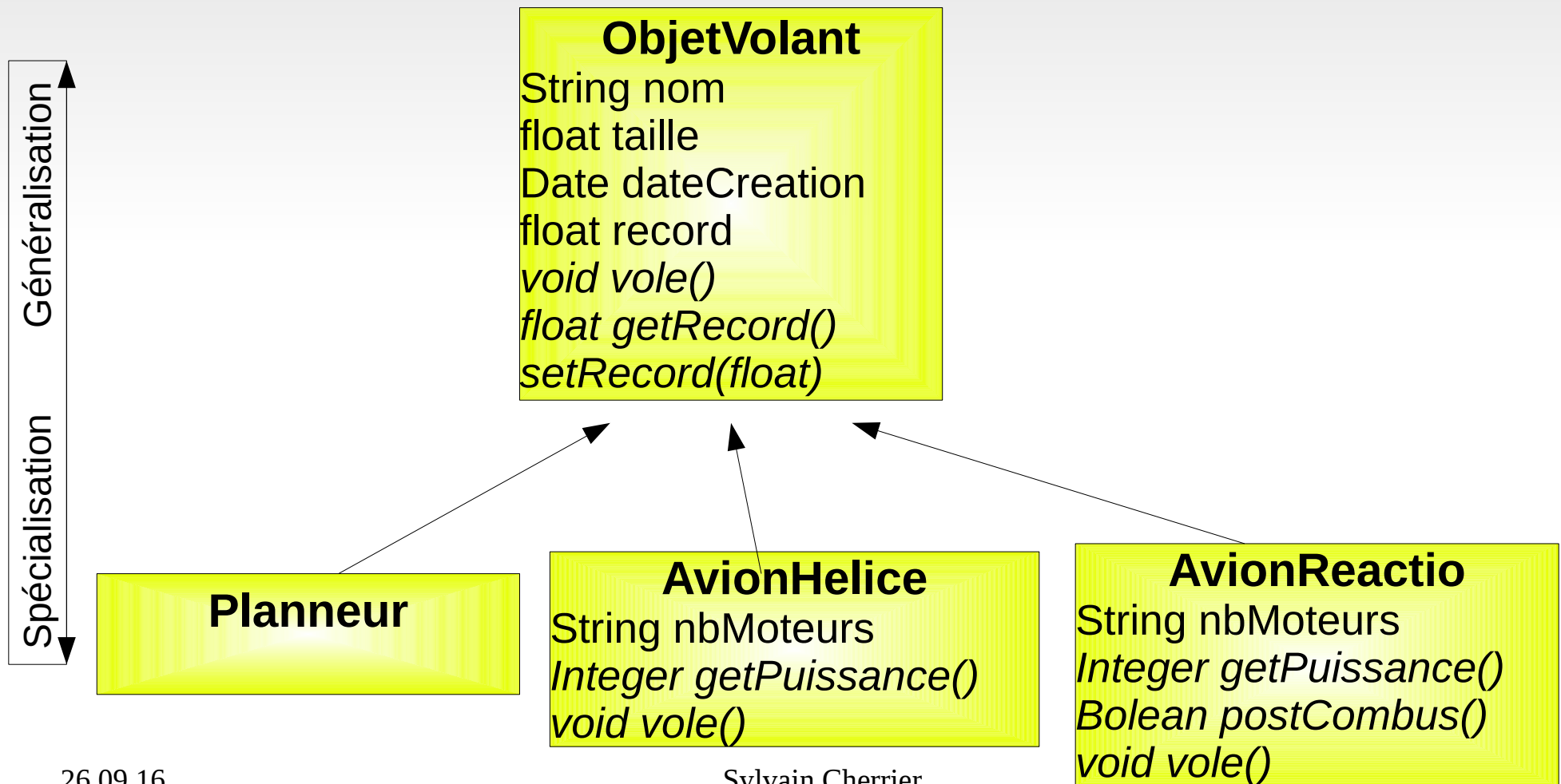
D'autres solutions

- Certains langages implémentent des solutions Objets différentes
- Javascript utilise la programmation orientée *prototype* pour réaliser la conception de nouveaux types d'objets (il n'y a pas de classe).
- Cela permet d'être plus dynamique
- Création à la volée de nouveaux types d'objets, changement de la hiérarchie d'héritage en cours d'utilisation...

Héritage : effets sur les attributs et méthodes

Héritage simple : Saut en hauteur

Une arborescence de Classes, permettant la gestion des objets volants.



Un programme utilisateur de nos classes

Un autre objet (*Compétition par exemple*) pourra utiliser nos classes. Quelques exemples.

```
ObjetVolant s1,s2; //déclaration
```

```
Planneur fb1;
```

```
s1=new ObjetVolant(); s2=new Planneur(); //oui un planneur est un objet volant
```

```
fb1=new AvionReaction(); //NON !!! ça n'est pas cohérent !!
```

```
fb1=new Planneur(); //oui
```

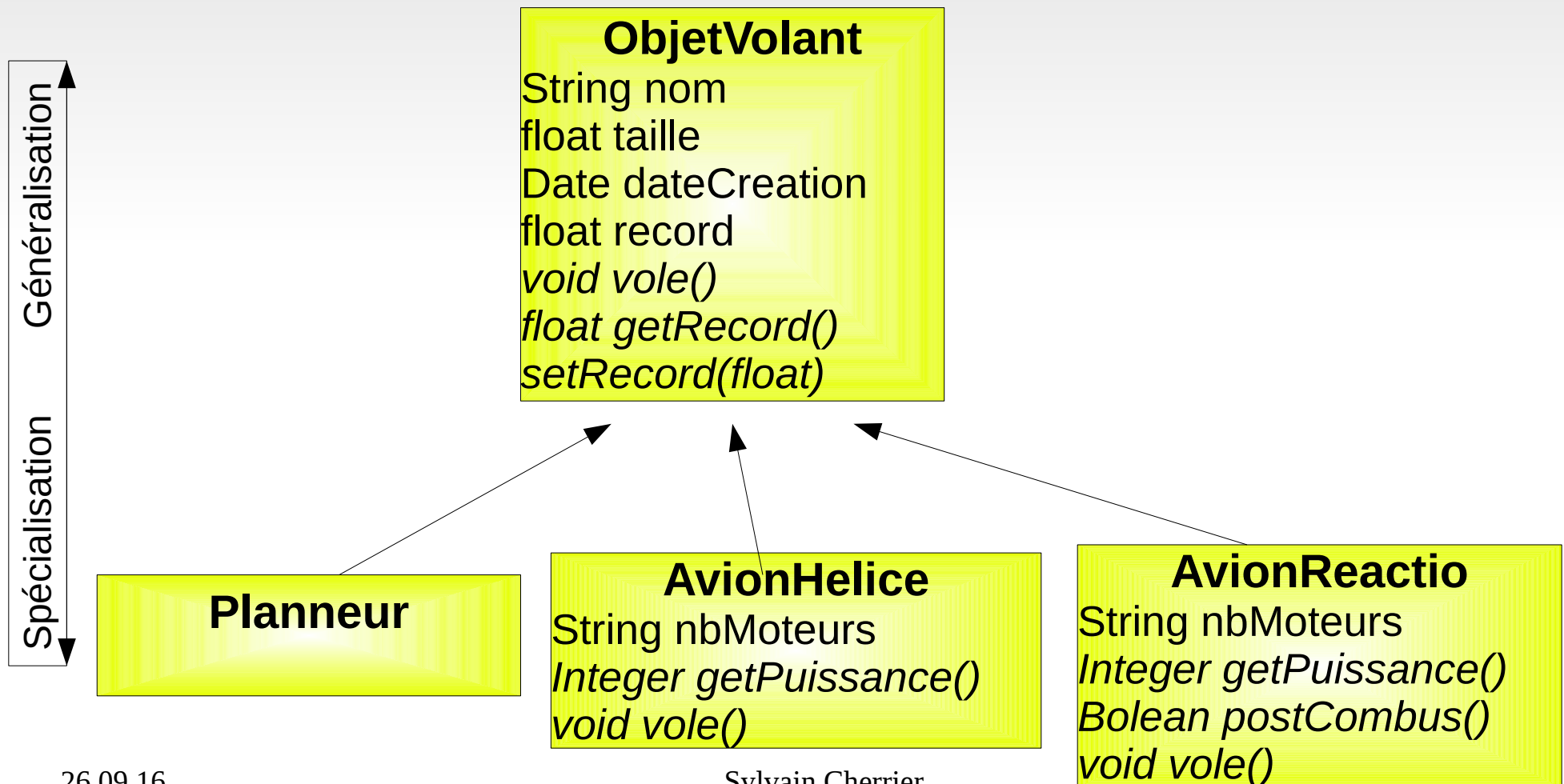
```
s1.vole();s2.vole(); //oui
```

```
float record=max(s1.getRecord(),s2.getRecord());
```

```
//oui, si max(float,float) existe dans le langage, et renvoie bien un float. Ce code est il bien objet ??
```


Héritage simple : Saut en hauteur

Une arborescence de Classes, permettant la gestion des objets volants.



Comportement général et spécialisation :

Le polymorphisme

Un comportement (une méthode) d'une super-Classe peut être modifié (**redéfini**) dans une sous-Classe. Cette sous-Classe se comporte d'une façon différente de la super-Classe pour cette même action. Cette **redéfinition**, qui permet de spécialiser le comportement hérité d'un ancêtre, correspond au **polymorphisme**. C'est la façon de sauter qui est redéfinie ici.

Planneur	AvionHelice	AvionReaction
<i>void vole()</i>	String nbMoteurs <i>void vole()</i> <i>Integer getPuissance()</i>	String nbMoteurs <i>void vole()</i> <i>Integer getPuissance()</i>

Redéfinition

Lors de l'appel d'une méthode, le langage recherche la plus adaptée, celle qui correspond à l'objet traité. On travaille au plus proche, puis on remonte dans l'arborescence des Classes jusqu'à trouver la méthode qui s'applique.

```
ObjectVolant s1,s2; AvionReaction fb1;  
s1=new ObjetVolant(); s2=new AvionReaction(); //oui un  
AvionReaction est un ObjectVolant  
fb1=new AvionReaction(); //oui  
s1.vole();s2.vole(); //oui, on appelle la méthode redéfinie  
float record=fb1.getRecord(); //ici, on remonte pour trouver la méthode  
float record2=s2.getRecord(); // Idem ici, le véritable objet est un  
AvionReaction, spécialisation de ObjectVolant.. AvionReaction ne redéfinit pas  
cette méthode, on appelle donc celle de l'ancêtre
```

Surcharge

Autre dispositif intéressant, la **surcharge**. Ici, on **décline** le comportement de la méthode **selon les arguments fournis**. Cela permet d'appeler une méthode avec différents arguments.

Imaginons par exemple une Classe qui représente l'idée qu'un programmeur se fait d'un Point dans l'espace en 2D. Cette classe doit nous fournir le service suivant : nous dire si oui ou non un autre Point est proche de lui, selon une marge...

Exemple Surcharge

```
public class Point {  
    Private int x,y; //des Attributs, un point à, ici, un x et un y...  
  
    public Point(int _x, int _y) { //constructeur  
        x=_x; y=_y;  
    }  
    public boolean proche(int leX,int leY, int marge) {  
        return Math.abs(leX-x)<marge&&Math.abs(leY-y)<marge;  
        /* ici, on explique comment marche la  
           méthode proche()*/  
    }  
    public boolean proche(Point p,int marge) {  
        return proche(p.x,p.y,marge); //surcharge..  
        /* on appelle la méthode de CETTE classe, on  
           reste à la même "profondeur" dans l'arborescence... */  
    }  
}
```

Il est temps de jouer....

Exercices

Essayez de donner les **caractéristiques** d'une Classe représentant un Compte bancaire (voir le formalisme). Un utilisateur doit pouvoir connaître le *solde du compte*, et effectuer des *opérations* sur ce compte (nota : un compte, comme un porte monnaie, ne connaît pas toutes les opérations qui ont été effectuées : Il ne connaît que sa situation actuelle... *Quel objet pourrait être chargé de cette mémoire?*)

Classe

```
nomAttribut1 : type  
nomAttribut2 : type  
nomAttribut3 : type
```

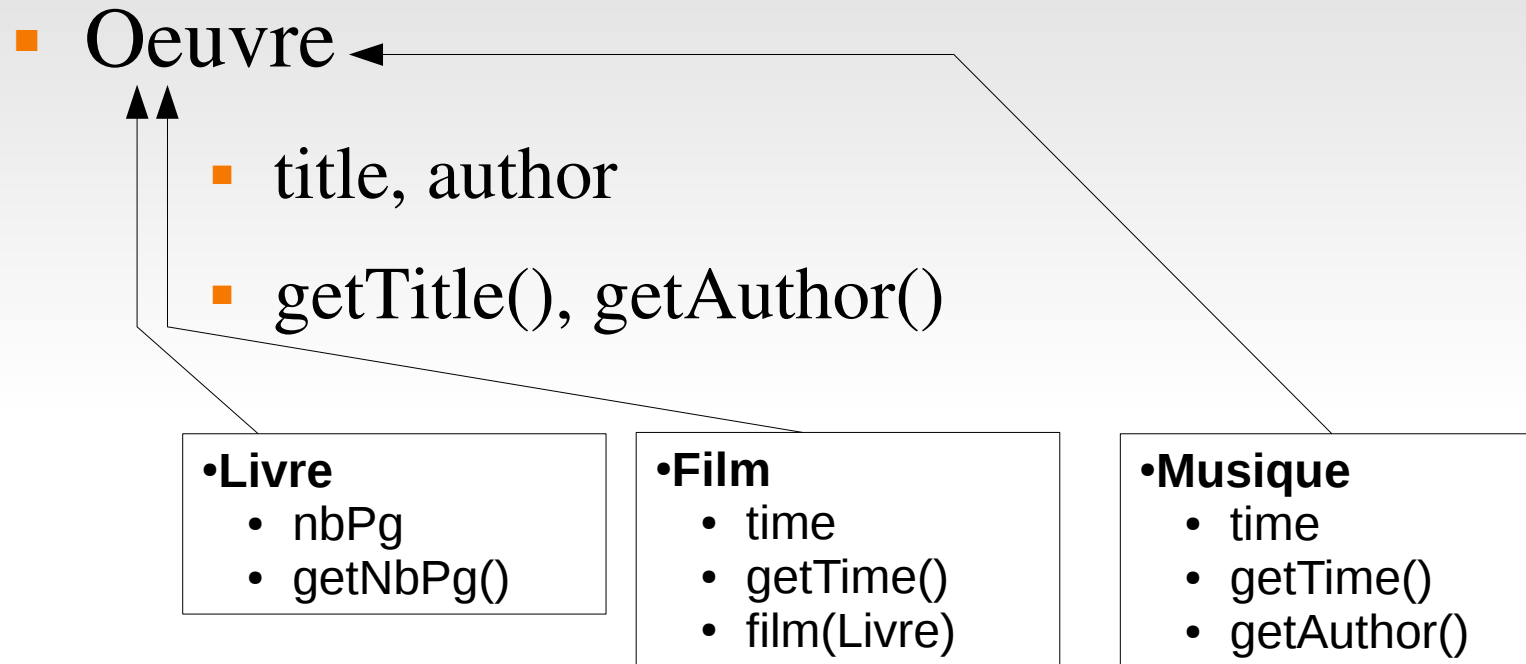
```
nomMéthode1(args..) : type  
nomMéthode2(args...) : type
```

Exercices...

On vous demande de représenter aussi un **Compte Rémunéré** (selon un certain taux, il rapporte des intérêts). Le client vous demande de pouvoir connaître le montant des intérêts générés, et le taux de ce compte (en plus des opérations habituelles d'un compte). (indice : Faut il tout ré-écrire ?)

Proposez une solution plus élégante.

Exercice



Héritage ? Surcharge ? Redéfinition ?

Exercices

- Essayez de construire la Classe qui permettrait de gérer des morceaux de musique (des Titres). Un titre a un nom, une durée. Il a peut être été classé dans les charts.
- Construisez ensuite la Classe Album. Un album contient des titres (peu importe comment c'est organisé pour le moment), il faut connaître le nom de l'Album, son auteur, etc... On doit pouvoir récupérer le nombre de Titres de l'album, la liste des titres (sous forme d'une longue chaîne), et un Titre selon son numéro de plage...