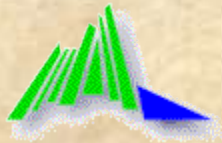


Programmation Orientée Objet
Langage Java™

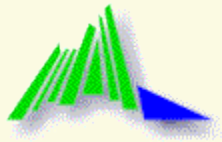
Manuel Munier

IUT des Pays de l'Adour - Mont de Marsan

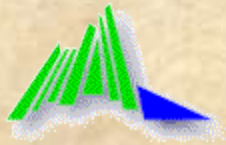
Département RT

2012-2013

Plan du cours



- Programmation Orientée Objet (POO)
  - modularité, encapsulation
  - objets, classes, instances, messages
  - héritage, liaison dynamique, généricité
- Langage Java
  - syntaxe du langage
  - classes et objets en Java
  - héritage, interfaces
  - paquetages, exceptions
  - architecture client/serveur (RMI)
  - Java et son environnement



Partie n°1

Programmation Orientée Objet

Plan

- Historique & Objectifs
- Notion d'objet
  - modularité, encapsulation
- Concepts de base
  - classes, instances, messages
- Concepts avancés
  - héritage, liaison dynamique, généricité

Historique

- Années 60 (début 70): C et Pascal organisent les programmes complexes: structures de données + fonctions + prog. principal
- Approche descendante, i.e. décomposition en sous-programmes
- L'accent est mis sur les traitements (les données ne servent que d'aliments aux fonctions)

Historique

- 1967: SIMULA introduit la notion de classe et d'objet
- Pour simuler le trafic d'un port, ses concepteurs créent des entités autonomes (bateau, port,…)
- Une entité regroupe à la fois une structure de données et les fonctions pour la manipuler: un **objet**

Historique

- En POO, un programme est un ensemble de petites entités autonomes qui interagissent et communiquent par messages
- L'accent est mis sur l'autonomie de ces entités et sur les traitements locaux
- ⇒ Programmer avec des objets nécessite un changement d'état d'esprit de la part du programmeur

Objectifs

- La prog. structurée (C, Pascal) est adaptée aux applications complexes mais peu évolutives
- Si les structures de données et les fonctions sont partagées par plusieurs prog., la modification d'une structure de données doit être répercutée sur tous les programmes
- Ex: 23 sept. 1999: "990923" → (23,09,1999)

Modularité

Objectifs

- 3 critères de qualité en Génie Logiciel
  - **fiabilité**: un composant fonctionne dans tous les cas de figure
  - **extensibilité**: on peut ajouter de nouvelles fonctionnalités sans modifier l'existant
  - **réutilisabilité**: les composants peuvent être réutilisés (en partie ou en totalité) pour construire de nouvelles applications
- En un seul mot: **modularité**

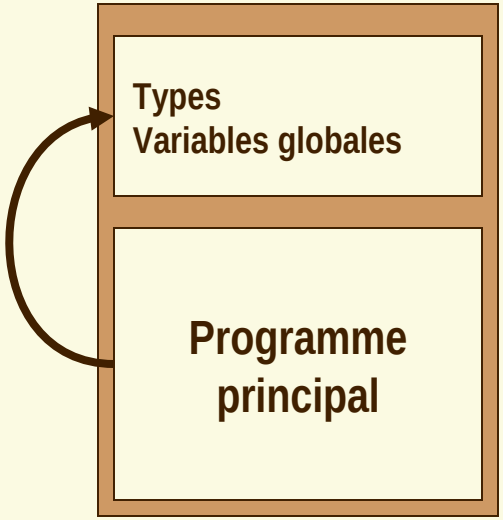
Modularité

Objectifs

- **Décomposition** d'un problème en sous-problèmes (modules). **Composition** = construction d'une application à partir de ces modules élémentaires
- **Compréhension**: la décomposition doit aider à comprendre le programme
- **Continuité**: en cas de changement des spéc., le nombre de modules à modifier doit être limité
- **Protection des données**: l'idéal est de n'accéder aux données d'un module qu'au travers des fonctions qu'il exporte

Approche brutale

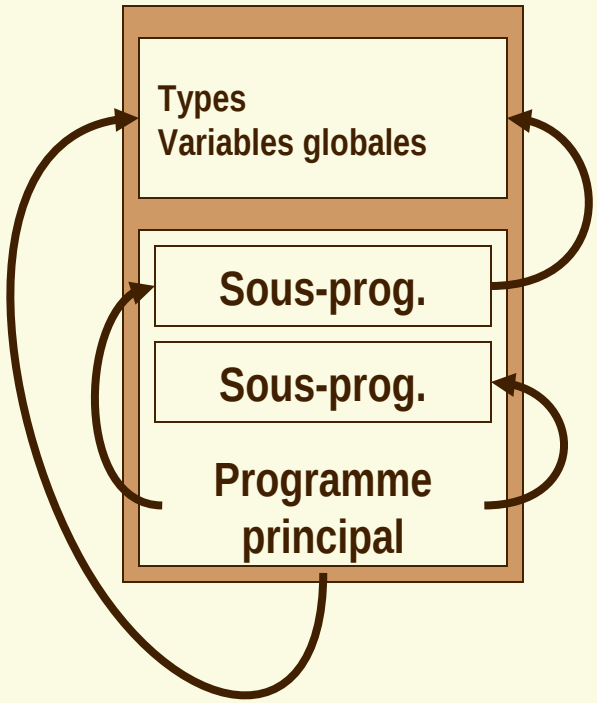
- Types et variables sont définis au début du fichier, puis un gros bloc d'instructions dépourvu de tout sous-programme
- Assembleur, programmation d'automates, basic,…
- C en 1ère année GTR ;-)



Approche fonctionnelle

Objectifs

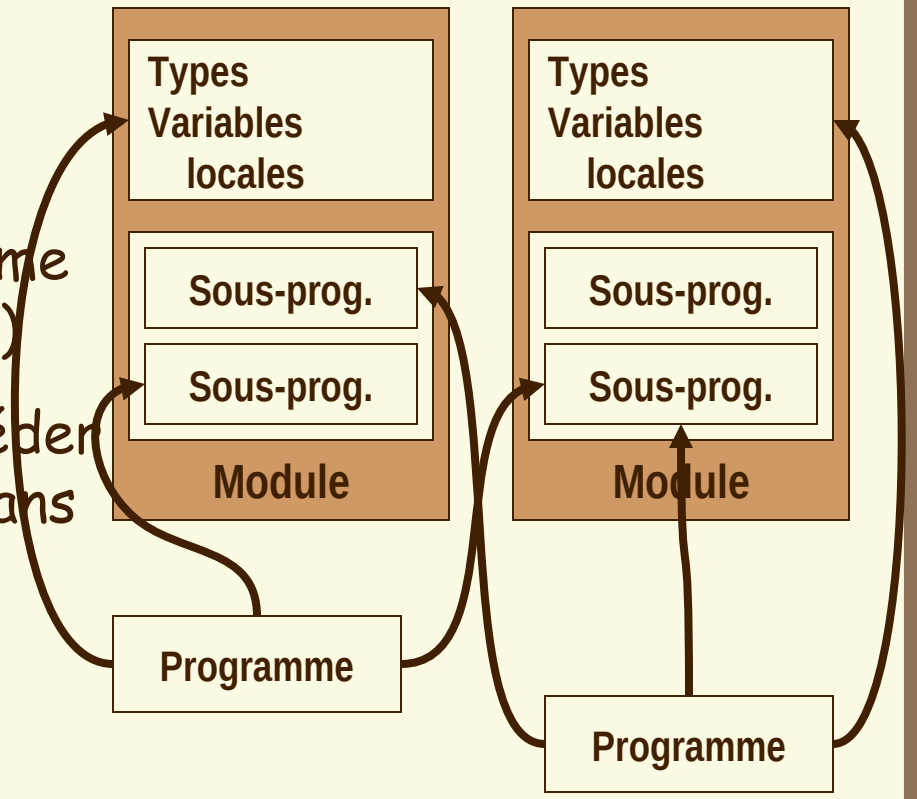
- La modularité ne concerne que les traitements, i.e. utilisation de fonctions
- Plus le programme devient complexe, plus il est difficile de s'y retrouver car tout se trouve dans le même fichier
- C en fin de 1ère année GTR



Approche structurée

Objectifs

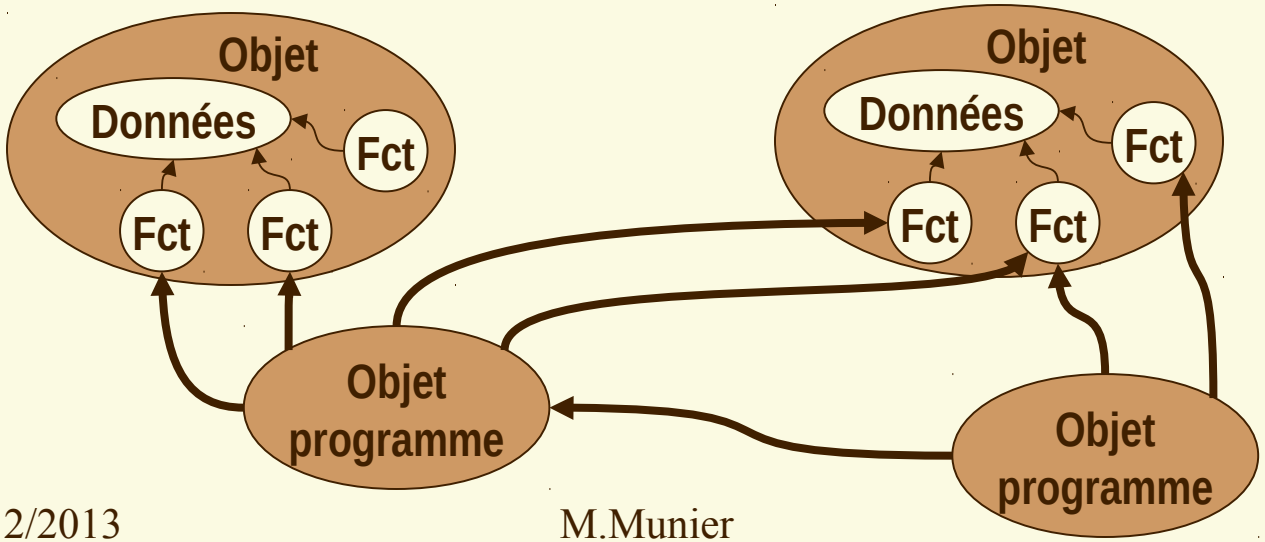
- Approche par abstractions
- Découpage en modules
- Ex: type **`pile`**
- Utilisation via le mécanisme d'import (**`#include`** en C)
- Pb: on peut toujours accéder aux données du module sans utiliser ses fonctions



Approche par objets

Objectifs

- On associe les données et les opérations qui les manipulent ⇒ plus grande indépendance entre programmes, opérations et données
- On force les programmes à manipuler les données d'un objet au travers des opérations de cet objet



Conception

Objectifs

- La modularité a un coût: à partir du cahier des charges, il faut identifier les objets (ou les abstractions)
- En amont de la phase de prog. ont ainsi été développées des méthodes d'analyse et/ou de conception orientées objet
- Ex: UML (the Unified Modeling Language)

Langages à objets

Objectifs

- SMALLTALK (1980)
  - à l'origine des LOO; langage faiblement typé, interprété
- C++ (1982)
  - successeur du C; fortement typé, hybride (POO et prog. structurée)
- EIFFEL (1988)
  - langage fortement typé, pur OO; concept de pré et post conditions (utilisé surtout en GL)

Langages à objets

Objectifs

- Java (1995)
  - langage fortement typé
  - les sources sont compilés en bytecode Java, lequel est ensuite interprété par une machine virtuelle Java
  - avantage: le bytecode est indépendant du matériel et du système
  - l'environnement Java de base (le JDK) intègre de nombreuses bibliothèques de classes en standard: réseau, processus, interface graphique, appel de fonctions distantes,...

Plan

- Historique & Objectifs
- Notion d'objet
  - modularité, encapsulation
- Concepts de base
  - classes, instances, messages
- Concepts avancés
  - héritage, liaison dynamique, généricité

Encapsulation

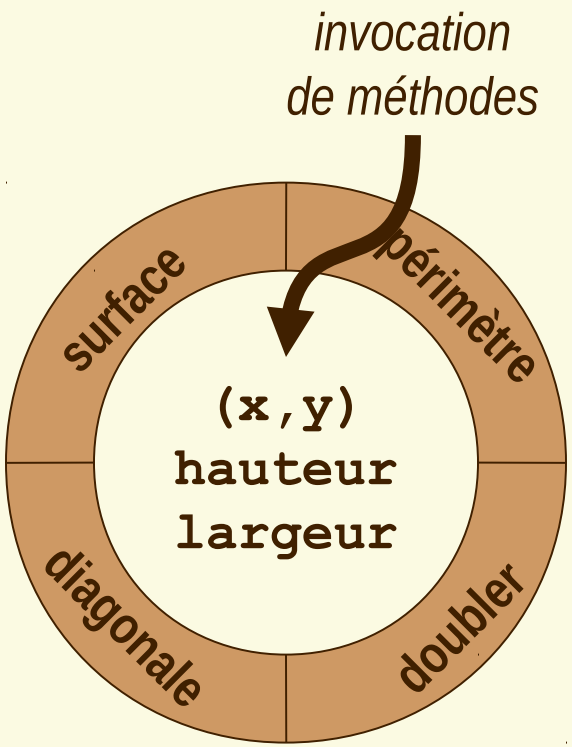
- Un **objet** est une **entité autonome** regroupant à la fois des données (attributs) et des opérations permettant de manipuler ces données (méthodes) ⇒ **encapsulation**
- Les **attributs** décrivent l'**état interne** de l'objet (aspect statique)
- Les **méthodes** décrivent le **comportement** de l'objet (aspect dynamique)

Abstraction de données

- Un objet n'est accessible qu'au travers de ses opérations visibles (les méthodes de son interface externe)
- Son implémentation (les structures de données et les attributs associés) est cachée
- On peut donc changer l'implémentation d'un objet sans affecter les prog. qui l'utilisent

Exemple d'objet

- Un objet rectangle expose 4 méthodes:
  - surface
  - périmètre
  - diagonale (longueur)
  - doubler (dim. × 2)
- Grâce à l'encapsulation et à l'abstraction de données, on pourrait remplacer les attributs par `{(x1,y1),(x2,y2)}`



Conclusion

- L'encapsulation favorise …
  - la modularité,
  - l'indépendance,
  - la réutilisation
- … des sous-systèmes en séparant
  - l'**interface** d'un composant (liste des services offerts)
  - de son **implémentation** (structures de données et algorithmes)

Plan

- Historique & Objectifs
- Notion d'objet
  - modularité, encapsulation
- Concepts de base
  - classes, instances, messages
- Concepts avancés
  - héritage, liaison dynamique, généricité

Classes

- Une **classe** est un moule pour fabriquer des objets ayant la même structure et le même comportement
- Une classe est composée
  - d'attributs qui décrivent la structure des objets qui seront produits
  - de méthodes, i.e. d'opérations qui leur sont applicables

Exemple: classe `Compte`

- Un compte bancaire est défini par deux attributs: **`crédit`** et **`débit`**
- Du fait de l'encapsulation, on doit définir des méthodes pour manipuler ces attributs: **`déposer`**, **`retirer`**, **`donnerSolde`**
- Une classe est définie en deux temps:
  - sa **spécification** (son interface)
  - son **implémentation** (corps des méthodes)

Exemple: classe Compte

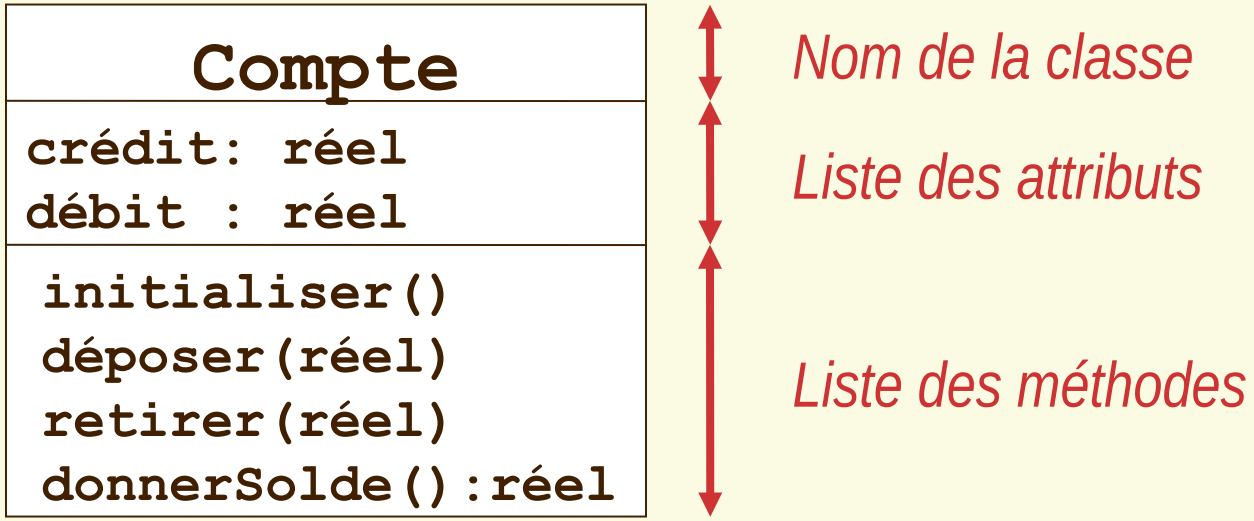
- Spécification en pseudo-langage objet:

```
classe Compte: spécification
    attributs:
        crédit, débit: réel;

    méthodes:
        initialiser();
        déposer(réel);
        retirer(réel);
        donnerSolde(): réel;
fin Compte
```

Exemple: classe Compte

- Représentation sur le graphe de classes:



Exemple: classe Compte

- Implémentation = corps des méthodes:

```
classe Compte: implémentation
    méthode initialiser()
        début
            crédit = 0.0;
            débit  = 0.0;
        fin

    méthode déposer(montant: réel)
        début
            crédit = crédit + montant;
        fin
```

Exemple: classe Compte

```
    /* suite implémentation classe Compte */

    méthode retirer(montant: réel)
        début
            débit = débit + montant;
        fin

    méthode donnerSolde(): réel
        début
            retourner(crédit - débit);
        fin

fin Compte /* fin de l'implémentation */
```

Tout est objet

- En prog. structurée, on distingue:
  - types simples (littéraux): réels, entiers,…
  - types composés: tableaux, structures, fichiers
- En POO, la notion d'objet sert de concept fédérateur à tous les composants logiciels:
  - les entiers peuvent être des objets créés à partir de la classe **`Nombre`**
  - idem pour la classe **`ChaîneDeCaractères`** dont les objets sont eux-mêmes composés d'objets de la classe **`Caractère`**

Instances

- Une **instance** est un objet créé à partir d'une classe (cf. moule) par un mécanisme appelé **instanciation** et matérialisé par l'opérateur **`new`**
- Pour les programmeurs C, le **`new`** peut être vu comme une sorte de **`malloc`**:
  - allocation mémoire
  - création de l'objet
  - retour d'une référence (pointeur) sur cet objet

Instance

- Exemple:

```
/* mon_compte est une instance de Compte
   */
mon_compte = new Compte();
```

- Lors de l'instanciation d'un nouvel objet, la méthode **`initialiser`**, si elle est définie dans la classe, est automatiquement appelée pour fixer l'état initial de l'objet
- Cette méthode particulière est appelée **constructeur**

Messages

- En POO, les instances communiquent entre elles par **envoi de messages** (ou invocation de méthodes)

```
<receveur>.<sélecteur>[(<liste d'arguments>)]
```

ou

```
<receveur> <= <sélecteur>[(<liste d'arguments>)]
```

- Envoyer un message à un objet, c'est lui dire ce qu'il doit faire (déclencher l'exécution d'une méthode)

Exemple


```
programme exempleCompte
début
    C1   : Compte;        /* référence ! */
    reste: réel;

    C1 = new Compte();  /* nouvelle instance */

    C1.déposer(10460.0);
    C1.retirer(3160.0);
    reste = C1.donnerSolde();

    afficher(reste);
fin exempleCompte;
```

envoi de messages à l'objet C1, instance de la classe Compte

Composition

Messages

- Si une méthode retourne une réf. objet, on peut envoyer des messages à cet objet

```
classe Banque: spécification
    attributs:
        comptes: Dictionnaire;
    méthodes:
        initialiser();
        ouvrirCompte(numCompte);
        déposer(numCompte,réel);
        retirer(numCompte,réel);
        donnerSolde(numCompte): réel;
fin Banque
```

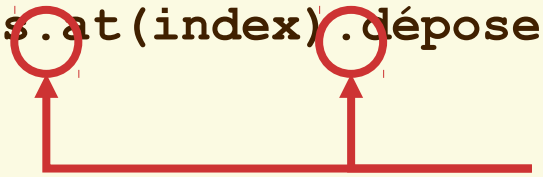
```
add(index,refObj)
at(index): refObj
```

Composition


```
classe Banque: implémentation
  méthodes initialiser()
    début
      comptes = new Dictionnaire();
    fin

  méthode ouvrirCompte(index: numCompte)
    début
      comptes.add(index,new Compte());
    fin

  méthode déposer(index: numCompte,montant: réel)
    début
      comptes.at(index).déposer(montant);
    fin
```



composition

Composition


```
méthode retirer(index: numCompte,montant: réel)
  début
    comptes.at(index).retirer(montant);
  fin

méthode donnerSolde(index: numCompte): réel
  début
    retourner(comptes.at(index).donnerSolde());
  fin

fin Banque
```

composition

composition

Référence `self`

- Les objets ont parfois besoin de s'envoyer des messages à eux-mêmes (appel d'une autre méthode de la classe)
- Il existe pour cela une référence objet particulière: **`self`**
- **`self`** représente le «je»: un objet invoque une de ses propres méthodes

Plan

- Historique & Objectifs
- Notion d'objet
  - modularité, encapsulation
- Concepts de base
  - classes, instances, messages
- Concepts avancés
  - héritage, liaison dynamique, généricité

Héritage

- L'**héritage** est un mécanisme permettant à une nouvelle classe de posséder automatiquement les attributs et les méthodes de la classe dont elle **dérive**
- Exemple: un compte épargne est un compte classique, capable également de produire des intérêts
- On dérive la classe `CompteEpargne` de la classe `Compte`

Exemple: CompteEpargne

- On ne caractérise que les différences avec la classe **Compte**

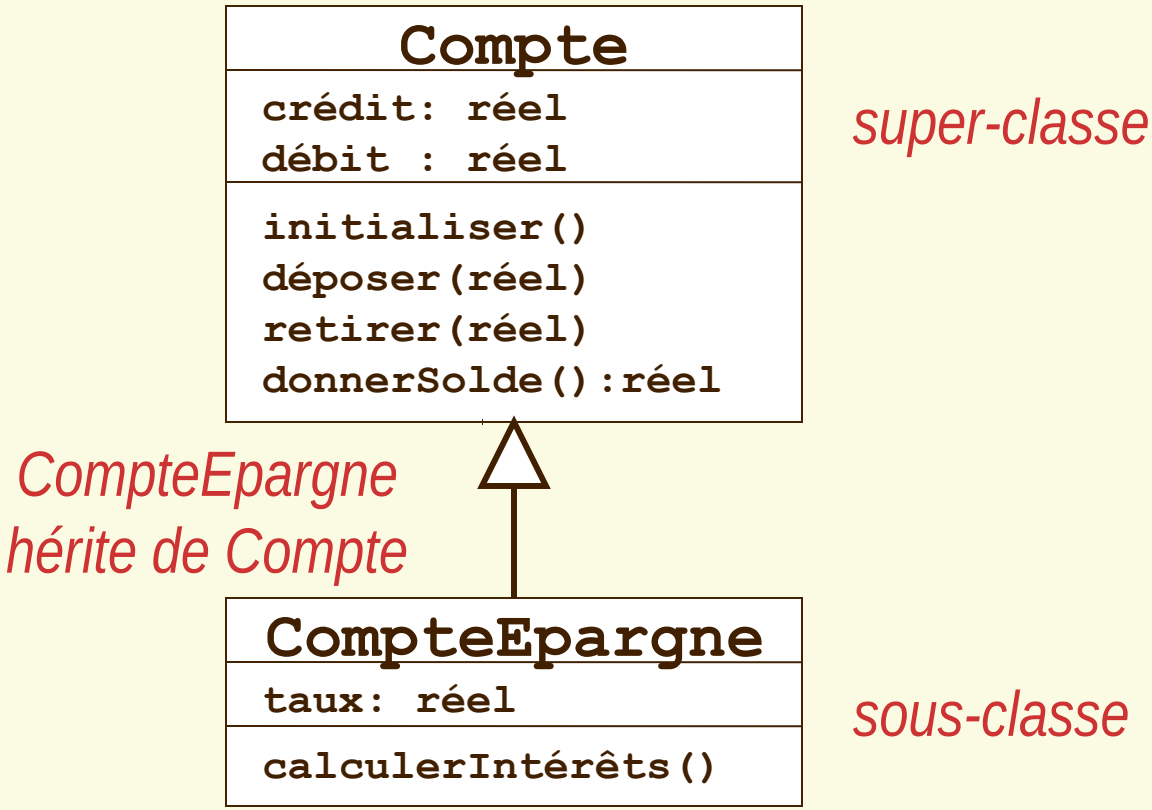
```
classe CompteEpargne: spécification
    sous-classe de Compte

    attributs:
        taux: réel;

    méthodes:
        calculerIntérêts();
fin CompteEpargne
```

Exemple: CompteEpargne

- Représentation sur le graphe de classes:



Exemple: CompteEpargne

- Toutes les instances de **CompteEpargne** disposent implicitement des attributs **crédit** et **débit**, ainsi que des méthodes **déposer**, **retirer** et **donnerSolde** (hérités de la classe **Compte**)

```
classe CompteEpargne: implémentation
  méthode calculerIntérêts()
    début
      crédit = crédit + self.donnerSolde()*taux;
    fin
fin CompteEpargne
```

Enrichissement

- Une classe A est incomplète
  - On définit une classe B qui hérite de A et on lui ajoute de nouvelles fonctionnalités (attributs et/ou méthodes)
  - On étend les fonctionnalités de la classe existante: **enrichissement** d'attributs et de méthodes

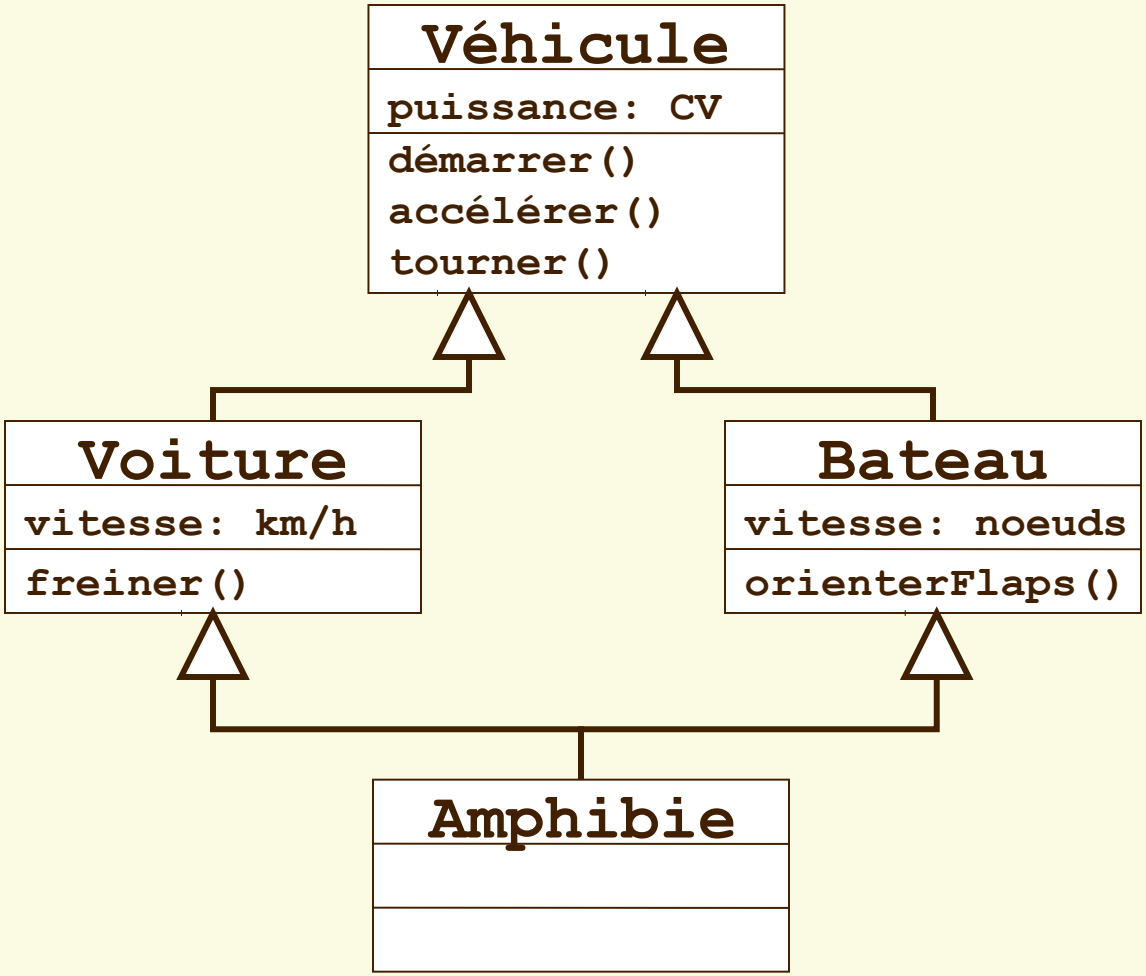
Substitution

- Une classe A est satisfaisante, sauf dans certains cas particuliers
  - On définit une classe B qui hérite de A et on y redéfinit certaines méthodes de A
  - On parle de **substitution**
  - Priorité est donnée aux attributs et aux méthodes redéfinis dans B par rapport à ceux et celles de même nom dans A
  - **Attention**: on modifie le **corps**, pas la **signature** d'une méthode !!!

Synthèse

- L'héritage induit un nouveau style de programmation: on procède par raffinages successifs de classes prédéfinies
- Il existe 2 formes d'héritage
  - **héritage simple**: une sous-classe n'hérite que d'une seule classe
  - **héritage multiple**: une sous-classe hérite de plusieurs classes (pas supporté par tous les LOO)

Héritage multiple



Héritage multiple

- L'héritage multiple peut introduire des **ambiguïtés** !
  - La classe **`Amphibie`** **hérite deux fois** de la méthode **`démarrer`** définie dans la classe **`Véhicule`**
  - Les classes **`Voiture`** et **`Bateau`** **définissent chacune** de leur côté un attribut **`vitesse`** avec un type différent
  - Les classes **`Voiture`** et **`Bateau`** **vont chacune redéfinir** la méthode **`tourner`** laquelle sera invoquée au niveau de la classe **`Amphibie`** ?

Héritage multiple

- Pour lever les ambiguïtés:
  - certains langages laissent ça «au hasard»
  - d'autres langages définissent un ordre de priorité par rapport au graphe de classes
  - le mieux est de résoudre ces ambiguïtés «à la main», i.e. de redéfinir l'attribut `vitesse` et la méthode `tourner` dans la classe `Amphibie`
- C'est en raison de ces problèmes (soit-disant) que certains langages ont choisi de ne pas supporter l'héritage multiple

Polymorphisme

- Une méthode est dite **polymorphe** si elle réalise des actions différentes selon la nature de l'objet qui reçoit le message
- Il y a 3 formes de polymorphisme:
  - surcharge de méthodes
  - méthodes prédominantes
  - méthodes virtuelles

Surcharge

Polymor-phisme

- Une méthode est dite surchargée s'il existe plusieurs implémentations associées à son nom

```
classe Vecteur3D: spécification
  attributs:
    x,y,z: réel;

  méthodes:
    modifierCoordonnées(réel,réel,réel);
    modifierCoordonnées(Vecteur3D);
    modifierCoordonnées(tableau[1..3] de réel);
fin Vecteur3D
```

Surcharge

Polymor-phisme

```
programme clientV3D
début
  t : tableau[1..3] de réel = {5.5,10,3.2};
  v1: Vecteur3D = new Vecteur3D();
  v2: Vecteur3D = new Vecteur3D();
  v3: Vecteur3D = new Vecteur3D();

  /* 3 manières différentes de modifier les */
  /* coordonnées d'un objet Vecteur3D       */

  v1.modifierCoordonnées(10,11.5,15.8);
  v2.modifierCoordonnées(v1);
  v3.modifierCoordonnées(t);
fin clientV3D
```

Surcharge

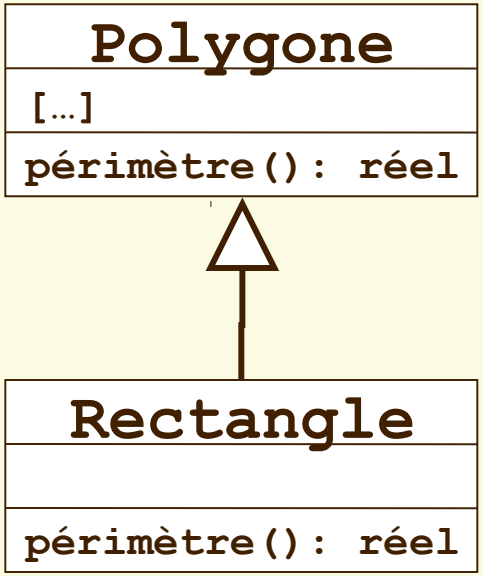
Polymor-phisme

- C'est le type des arguments qui détermine l'implémentation de la méthode à utiliser
- Etant donné que le type des arguments est connu lors de la compilation, c'est le compilateur qui effectue le lien entre le message à envoyer et l'implémentation à utiliser
- On parle lors de **liaison statique**

Prédominance

Polymor-phisme

- La prédominance de méthode est introduite par l'héritage quand la sous-classe redéfinit une méthode de sa super-mère
- La méthode **`périmètre`** redéfinie par la classe **`Rectangle`** est plus efficace que celle définie pour un polygone en général



Prédominance

Polymor-phisme

- Quand on invoque la méthode **`périmètre`** sur un objet de la classe **`Rectangle`**, on regarde tout d'abord si elle a été redéfinie
  - si oui, la méthode redéfinie est prioritaire sur celle définie dans la super-classe
  - si non, on invoque la méthode de la super-classe
- Le choix de l'implémentation est réalisé à l'exécution: on parle de **liaison dynamique**

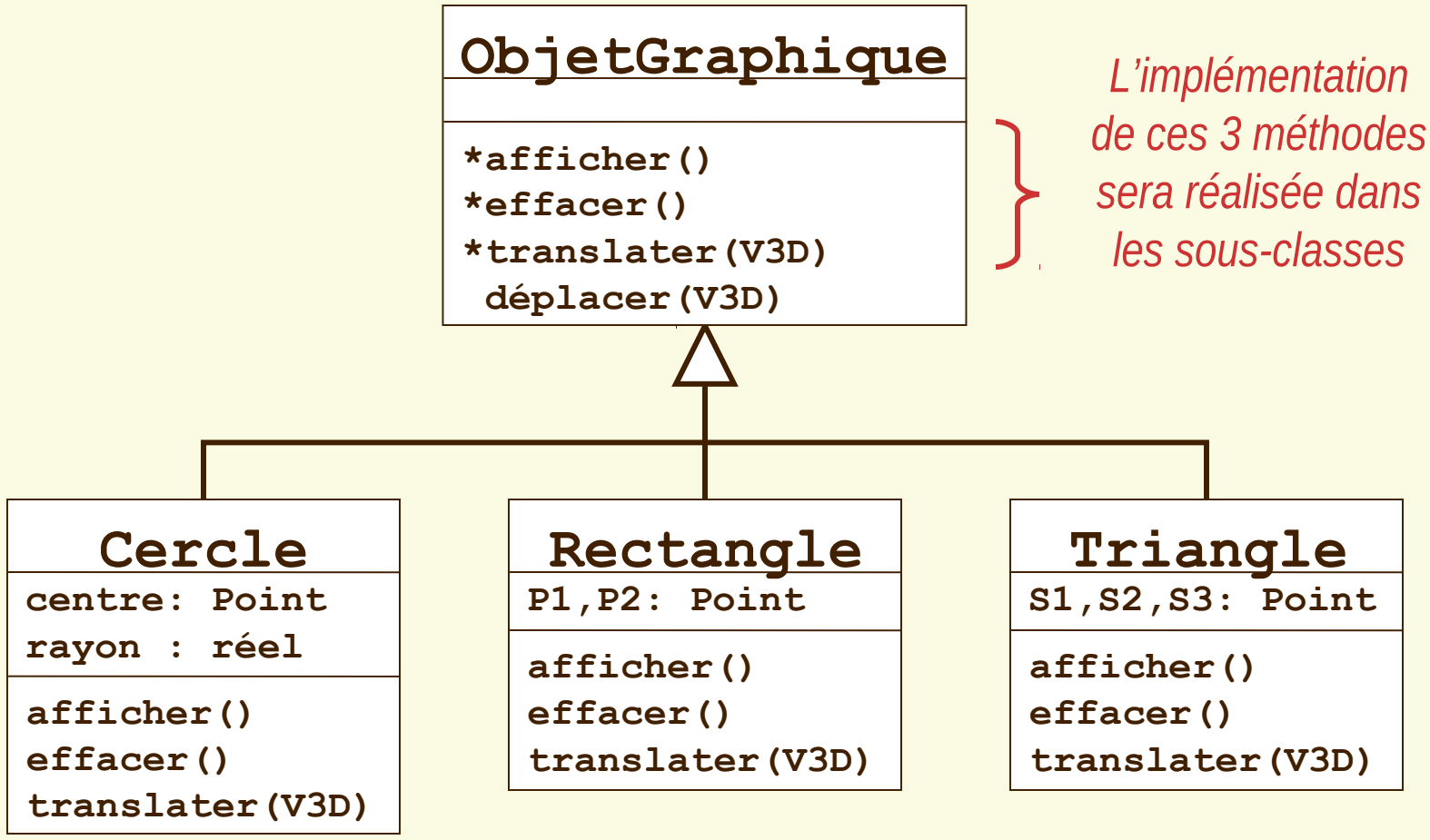
Méthodes virtuelles

Polymor-phisme

- Héritage + `self` ⇒ méthodes virtuelles
- L'implémentation d'une méthode virtuelle (ou abstraite) n'est pas réalisée dans la classe elle-même mais dans ses sous-classes
- Une classe disposant de méthodes abstraites est appelée classe abstraite

Méthodes virtuelles

Polymor-phisme



Méthodes virtuelles

Polymor-phisme

- Si les méthodes **afficher**, **effacer** et **translater** sont spécifiques à la forme, la méthode **déplacer** est générique

```
classe ObjetGraphique: implémentation
  méthode déplacer(v: Vecteur3D)
    début
      self.effacer();
      self.translater(v);
      self.afficher();
    fin
fin ObjetGraphique
```

*Choix de l'implémentation à l'exécution par liaison dynamique en fonction de la classe **réelle** de l'objet*

Celle de new, pas la super-classe

Généricité

- On peut également concevoir des classes polymorphes: c'est la généricité

```
classe Liste<T>  /* les éléments de la liste  */
                 /* seront des instances de T */
  méthodes:
    longueur(): entier;
    ième(entier): T;
    insérerEnQueue(T);
    supprimer(entier);
fin Liste
```

- Instanciation: `l = new Liste<ObjetGraphique>;`

Conclusion

- Historique & Objectifs
- Notion d'objet
  - modularité, encapsulation
- Concepts de base
  - classes, instances, messages
- Concepts avancés
  - héritage, liaison dynamique, généricité