CNAM
UV 19357
Année 2003-2004

David Delahaye
David.Delahaye@cnam.fr

Conception et Développement Orientés Objets*

Cours 1 : Introduction

1 Présentation de la valeur

Ce cours s'adresse à toute personne ayant une expérience de la programmation. Son objectif est d'étudier les concepts fournis par les langages de programmation orientés objets. Les langages supports seront Java et Objective Caml. Ces deux langages seront d'abord présentés d'une façon informelle à partir d'exemples de façon à apprécier les pouvoirs expressifs de ces langages. Ensuite, les traits objets sont étudiés d'une façon plus approfondie du point de vue d'abord de la sémantique puis du typage. Le cours se terminera par l'utilisation des traits objets dans les méthodes de conception des logiciels et des spécifications.

2 Les paradigmes de programmation

On peut identifier 5 paradigmes de programmation :
- Programmation impérative (Pascal, Fortran, C++/C, ...)
- Programmation fonctionnelle (Lisp, ML, Scheme, ...)
- Programmation objet (Smalltalk, Eiffel, Java, ...)
- Programmation logique (Prolog, ...)
- Programmation parallèle (MPI, ...)

Les frontières entre ces différents paradigmes ne sont pas toujours franches pour un langage de programmation donné et il n'est pas rare qu'un langage soit fondé sur plusieurs de ces paradigmes. Par exemple, Java est un langage orienté objet mais le *corps* des fonctions ou procédures est bien décrit impérativement. Il en est de même pour Objective Caml qui est un langage fonctionnel mais qui dispose également de fonctionnalités objets.

3 Les concepts de la programmation objet

3.1 Notion d'objet

Un objet est composé de :
- données
- traitements agissant sur ces données

Exemple 3.1 *Un compte bancaire peut être représenté par :*

*Web : http ://deptinfo.cnam.fr/Enseignement/CycleProbatoire/CDI/Cours-OO/.

– *un solde (données)*
– *les opérations de dépôt et de retrait (traitements)*

Les données sont nommées par des variables appelées *variables d'instance* (ou *champs*). On appelle *état* d'un objet la valeur de ces variables d'instance à un moment donné (lors de l'exécution du programme). Les traitements sont décrits par des fonctions et des procédures que l'on appelle des *méthodes*. Un objet est donc constitué de variables d'instance et de méthodes.

Comment créer plusieurs comptes ? Une solution consiste à réécrire le code qui définit un compte pour autant d'instances que l'on souhaite. Ce n'est évidemment pas *pratique* sans compter qu'il y a duplication de code et la maintenance sera *très* difficile.

Certains langages apportent une réponse à ce problème par le biais non pas d'objets mais d'unités modulaires. Par exemple :
– Ada avec la notion de *package* générique
– Objective Caml avec la notion de module

Les langages objets apportent une solution différente avec la notion de classe qui va nous permettre de créer plusieurs objets. La définition préalable d'une classe avant toute création d'objets est d'ailleurs une étape obligatoire dans de nombreux langages objets où il est impossible de créer directement un objet.

3.2 Notion de classe

Une classe décrit des schémas d'objets. Du point de vue typage (on y reviendra plus précisément dans les autres chapitres de ce cours), une classe doit être vue comme le *type* des objets qui seront créés à partir d'elle. Une classe définit les variables d'instance (par leur nom et leur type avec éventuellement une valeur initiale) et les méthodes.

Exemple 3.2 *En utilisant une syntaxe* Java*, les comptes bancaires introduits dans la sous-section 3.1 peuvent être formalisés par la classe suivante :*

```
class Compte{
  int solde = 0;
  void depot (int n){
    solde = solde + n;
  }//depot
  void retrait (int n){
    solde = solde - n;
  }//retrait
  void print (){
    System.out.println(solde);
  }//print
}//Compte
```

où nous avons rajouté la méthode `print` *nous permettant d'afficher le solde du compte (nous verrons pourquoi de telles méthodes sont parfois nécessaires suivant les règles de visibilité données par la classe).*

Ainsi, une classe va servir de *modèle* pour la création d'objets. En reprenant notre exemple, la création d'objet en Java se fait par le mot-clé `new`. Un objet de type `Compte`

sera donc créé par l'expression `new Compte()`. Cet objet pourra être lié à une variable que l'on a déclarée de la manière suivante :

```
Compte martin;
martin = new Compte();
//ou directement: Compte martin = new Compte();
```

L'expression `new Compte()` doit être vue comme l'appel à une méthode particulière de la classe `Compte`, appelé *constructeur*, qui permet la création (et l'initialisation) des objets de la classe `Compte`. Nous reviendrons plus en avant sur cette notion de constructeur par la suite. On dira qu'un objet créé à partir d'une classe est une *instance* de cette classe. Ainsi, dans note cas, `martin` est une instance de `Compte`.

Une fois un objet créé, nous pouvons appeler ses méthodes qui vont modifier son état. L'appel des méthodes se fait généralement en préfixant avec le nom de l'objet (car il ne faut pas oublier qu'il y a autant de méthodes que d'objets et utiliser simplement le nom d'une méthode pour l'invoquer est ambigu). En Java, avec notre exemple, on peut réaliser les opérations suivantes :

```
martin.depot(500);
martin.retrait(100);
martin.print();
```

3.3 Héritage

On appelle *héritage* la possibilité de créer une nouvelle classe en enrichissant ou en raffinant une classe déjà existante. Cet enrichissement/raffinement peut porter sur :

- les données : ajout de nouvelles variables d'instances
- les méthodes :
  - ajout de nouvelles méthodes
  - redéfinition de méthodes déjà présentes

On appelle *super-classe* (ou *classe mère*) la classe dont on souhaite hériter et *sous-classe* (ou *classe fille*) la classe qui hérite.

Exemple 3.3 *En reprenant notre exemple sur les comptes bancaires, on peut vouloir former une nouvelle classe représentant les comptes d'épargne qui permettent d'augmenter le solde suivant un certain taux. Plutôt de réécrire une nouvelle classe avec les mêmes variables d'instance et les mêmes méthodes que la classe* `Compte` *puis de la compléter avec ce qui est spécifique aux comptes d'épargne, il suffit simplement de créer une nouvelle classe qui va hériter de la classe* `compte`*. En* Java*, l'héritage se fait à l'aide du mot-clé* `extends` *:*

```
class CompteEpargne extends Compte{
  int taux = 5; //exprimé en %
  void interet(){
    solde = solde + ((solde * taux)/100);
  }//interet
}//CompteEpargne
```

Ainsi, les objets de la classe `CompteEpargne` *posséderont également la variable d'instance* `solde` *et les méthodes de la classe* `Compte`. *Ces objets posséderont, en plus (par rapport à la classe* `Compte`*), la variable d'instance* `taux` *et la méthode* `interet`. *Voici un exemple d'opérations que l'on peut réaliser sur les objets de* `CompteEpargne` *:*

```
CompteEpargne dupont = new CompteEpargne();
dupont.depot(1000);
dupont.interet();
dupont.print();
```

Attention, remarquez que le temps n'est ici pas formalisé et c'est à votre banquier d'appeler la méthode `interet` *chaque fois que cela sera nécessaire (tous les mois, tous les ans, ...).*

Exemple 3.4 *Toujours avec la classe* `Compte`, *on peut décider de* sécuriser *la méthode* `retrait` *de manière à n'autoriser un retrait uniquement si le solde est suffisant. Pour ce faire, on va hériter de la classe* `Compte` *et redéfinir la méthode* `retrait` *(même spécification mais corps différent) :*

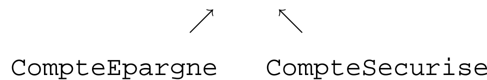
```
class CompteSecurise extends Compte{
  void retrait(int n){
    if (solde >= n)
      solde = solde - n;
    else{
      System.out.print("Provision insuffisante:");
      print();
    }//if
  }//retrait
}//CompteSecurise
```

On peut remarquer ici qu'évidemment un objet peut appeler ses propres méthodes (ici la méthode `print` *appelée dans la méthode* `retrait`*). Nous verrons qu'avec la redéfinition, se pose le problème de savoir quelle méthode est réellement appelée (celle de la super-classe ou de la sous-classe) et qu'il existe des moyens pour préciser ces appels. On peut, par exemple, réaliser les opérations suivantes :*

```
CompteSecurise durand = new CompteSecurise();
durand.depot(200);
durand.retrait(300);
```

La relation d'héritage entre classes définit une hiérarchie. Lorsqu'une classe n'hérite que d'une seule autre classe, on dit que l'héritage est *simple*. La hiérarchie des classes est alors un arbre. Lorsqu'une classe hérite de plusieurs autres classes, on dit que l'héritage est *multiple*. La hiérarchie des classes est alors un treillis. Certains langages ne permettent pas l'héritage multiple (prétendant que l'usage en est plutôt anecdotique). C'est le cas de Java, par exemple. D'autres l'autorisent comme C++, Eiffel ou Objective Caml. Dans notre exemple, nous n'avons que de l'héritage simple (nous utilisons Java) et l'arbre d'héritage est le suivant :

`Compte`



où la relation A $\leftarrow$ B signifie que B hérite de A.

À noter que Java rajoute systématiquement un niveau supplémentaire avec la racine Object, qui fournit un certain nombre de méthodes et constructeurs de base (nous y reviendrons lors des séances réservées à Java).

3.4 Encapsulation des données et messages

Un concept majeur de la programmation orientée objet est l'*encapsulation des données*. En effet, on peut voir un objet comme une structure qui encapsule les variables d'instance qui ne sont pas manipulées directement par l'utilisateur mais via les méthodes. Ainsi, les variables d'instance sont bien des variables globales du point de vue des différentes méthodes de l'objet mais sont bien des variables locales à l'objet du point de vue de l'utilisateur de l'objet. Cette portée limitée des variables d'instance est extrêmement pratique car on peut créer autant d'objets que l'on souhaite sans jamais avoir de conflits ou d'ambiguïté en référençant une de ses variables (il faudra toujours préfixer par l'objet).

L'encapsulation des données est différente suivant les langages et dépend des règles de visibilité adoptées par défaut par ces langages. On distingue généralement 3 règles de visibilité (on donnera, à titre d'exemple, le mot-clé Java correspondant) pour les différents composants d'une classe (variables d'instance et méthodes) :

– publique (public) : visible de partout
– protégée (protected) : visible à l'intérieur de la classe et des sous-classes
– privée (private) : visible que pour les méthodes de la classe

Dans certains langages, il y a aussi une notion de composant *ami* (il n'y a pas de mot-clé correspondant) visible par toutes les autres méthodes des classes définies dans le même fichier ou package. On n'abordera pas cette notion qui est un peu *ad hoc* et n'a pas réellement sa place dans le paradigme objet en général.

En Java, par défaut, tout est publique, même les variables d'instance, ce qui tend à dire que, par défaut, Java ne respecte pas vraiment la notion d'encapsulation des données propre au paradigme objet puisque tout utilisateur de l'objet peut accéder mais aussi modifier les variables d'instances de l'objet. En Objective Caml, par défaut, le choix est plus fidèle au paradigme puisque les variables d'instances sont privées et les méthodes publiques.

Exemple 3.5 *Dans notre exemple précédent, il serait judicieux de* protéger *un peu plus la variable d'instance* solde *pour éviter des manipulations directes sans passer par les méthodes appropriées. Pour ce faire, le code de la classe devient :*

```
class Compte{
  protected int solde = 0;
  void depot (int n){
    solde = solde + n;
  }//depot
  void retrait (int n){
    solde = solde - n;
  }//retrait
```

```
  void print (){
    System.out.println(solde);
  }//print
}//Compte
```

Ici, nous utilisons `protected` *plutôt que* `private` *afin que les méthodes de* `CompteEpargne` *et de* `CompteSecurise` *puissent utiliser* `solde`*. Par contre, dans la classe* `CompteEpargne`*, la variable d'instance* `taux` *peut être déclarée* `private` *puisqu'il n'y a pas, a priori de sous-classe et l'utilisateur n'a pas de raison de pouvoir y accéder.*

De manière générale, on veillera à respecter les règles suivantes :

1. Définir les variables d'instance en `private` ou éventuellement en `protected` si elles doivent être utilisées par les héritiers directement.
2. Prévoir (si nécessaire) des fonctions d'accessibilité en lecture et écriture pour chaque variable d'instance.
3. Limiter au minimum nécessaire (suivant les applications visées) le nombre des méthodes publiques de la classe.
4. Prévoir un certain nombre de méthodes privées (`private` ou `protected`) pour faciliter les tests, la réutilisation ou l'extension de la classe.

Une autre notion majeure de la programmation orientée objet est le principe de message (introduit par Smalltalk dans les années 70). Les objets interagissent et communiquent entre eux par l'envoi de messages. Les messages sont définis par 3 composants de base :

- l'objet à qui s'adresse le message
- le nom de la méthode à déclencher
- les paramètres nécessaires à la méthode

3.5 Liaison retardée et coercion (sous-typage)

Dans une hiérarchie de classes, des méthodes peuvent être redéfinies. Il n'est pas toujours possible de déterminer statiquement quel code doit être exécuté lorsqu'une méthode redéfinie est invoquée.

Par exemple, soit le code suivant :

```
class Banque {
  public Compte ouverture(){
    Compte C;
    String reponse;
    DataInputStream In = new DataInputStream(System.in);
    System.out.println("Voulez-vous un compte securisé?");
    reponse=In.readLine();
    if (reponse == "oui")
      C = new CompteSecurise();
    else
      C = new Compte();
    return C;
  }//ouverture
  ...
```

```
    Compte durand = ouverture();
    durand.depot(300);
    durand.retrait(400);
    durand.print();
  ...
}//Banque
```

La classe de l'objet durand dépend d'une valeur entrée par l'utilisateur, qui n'est donc connue qu'à l'exécution. La méthode `retrait` n'est pas la même selon la classe (`Compte` ou `CompteSecurise`). Son code ne pourra être déterminé que lors de l'exécution. La liaison du nom au code est donc dynamique, on dit qu'on a alors une *liaison retardée* (ou *liaison dynamique*). Par contre, la méthode `print` est la même dans les deux classes et la liaison nom-code est déterminée statiquement à la compilation. La liaison retardée peut être vue comme une méthode pour simuler du polymorphisme.

Ce principe de liaison retardée a une influence sur le typage qui se doit (pour permettre ce mécanisme dynamique) d'introduire une notion de coercion (ou de sous-typage). En effet, on peut remarquer dans le code précédent que la variable `C` est de type `Compte` et qu'une affectation avec un objet de type `CompteSecurise` ne pose pas de problèmes. Cela signifie que dans les langages objets, un objet de type sous-classe peut-être vu comme un objet de type super-classe. Ce sous-typage permet des liaisons dynamiques à l'exécution mais peut être également source de confusion dans certains langages objets.

3.6 Classes abstraites ou virtuelles

Il est généralement possible de créer des classes abstraites (ou virtuelles) pour lesquelles il manque le corps de certaines méthodes (on ne connaît que leur spécification, typiquement leur type). Il n'est pas possible de créer d'instances de classes abstraites et pour les utiliser (en créant des objets), il suffit d'en hériter en *concrétisant* les méthodes abstraites (i.e. en donnant la définition de leur corps).

Exemple 3.6 *Prenons comme exemple une classe représentant les opérations binaires :*

```
abstract class OpBin{
  int a=0,b=0;
  abstract int operation();
  int oppose(){
    return -(operation());
  }//oppose
  void modifier(int a0,int b0){
    a = a0;
    b = b0;
  }//modifier
}//OpBin
```

La classe `OpBin` *possède deux variables d'instance représentant les opérandes. Elle exporte les méthodes* `operation` *(l'opération binaire proprement dite),* `oppose` *(retournant l'opposé) et* `modifier` *(permettant de modifier les opérandes). La méthode* `operation` *est abstraite (déclarée* `abstract` *en* Java*), rendant par là-même la classe*

`OpBin` *abstraite, car cette méthode représente n'importe quelle opération binaire (addition, soustraction, ...). On peut rendre cette classe concrète en créant des sous-classes qui donneront un corps à* `operation` *:*

```
class Plus extends OpBin{
  int operation(){
    return a+b;
  }//operation
  Plus (int a,int b){
    modifier(a,b);
  }//Plus
}//Plus

class Moins extends OpBin{
  int operation(){
    return a-b;
  }//operation
  Moins(int a,int b){
    modifier(a,b);
  }//Moins
}//Moins
```

Ces deux classes `Plus` *et* `Moins` *donnent un corps à* `operation` *(addition et soustraction). Elles sont donc concrètes et on peut créer des instances de ces classes. Les méthodes* `Plus` *et* `Moins` *sont des constructeurs respectifs de ces classes permettant d'initialiser les opérandes (par d'autres valeurs que* `0`*). On peut utiliser ces deux nouvelles classes de la manière suivante :*

```
OpBin op;
op = new Plus(1,2);
System.out.println(op.operation());
System.out.println(op.oppose());
op = new Moins(3,2);
System.out.println(op.operation());
System.out.println(op.oppose());
```

On peut remarquer que la liaison retardée joue un rôle primordial dans le cas de classes abstraites puisque statiquement, les méthodes abstraites ne sont liées à aucun code (contrairement à la redéfinition où il y a quand même un code même si ce n'est pas forcément le bon). Les méthodes abstraites seront donc forcément liées dynamiquement (sauf si l'objet est déclaré comme étant du type de la sous-classe concrète).

On peut voir également que dans le cas de classes abstraites, la relation d'héritage ne se conçoit plus réellement comme un enrichissement mais plutôt comme un raffinement. Dans notre exemple, on part d'opérations binaires générales, pour arriver à des opérations binaires bien particulières comme l'addition et la soustraction. Ainsi, l'héritage peut être vu non seulement comme un enrichissement lorsque l'on ajoute des variables d'instances ou des méthodes mais aussi comme un raffinement lorsque l'on précise la sémantique de certaines méthodes.

3.7 Avantages et limites de la programmation objet

Du point de vue du génie logiciel, la programmation orientée objets a de nombreux avantages qui résident principalement en 3 mots-clés : fiabilité, maintenabilité et réutilisabilité. En particulier, on a :
- l'encapsulation des données (protection entre composants)
- une abstraction entre implantation et utilisation
- une meilleure compréhension du logiciel (vision modulaire)
- une réutilisation aisée (directe ou par héritage)
- un code *réduit* (héritage, liaison retardée)
- une propagation des modifications *longitudinales* (par héritage)

La programmation a néanmoins ces limites dues essentiellement à sa sémantique et son typage. Entre autre, on a :
- une sémantique difficile à formaliser
- un code pas toujours efficace (liaison retardée)
- un typage difficile, dynamique et donc coûteux
- une explosion *rapide* des classes (outils adaptés nécessaires)

3.8 Un petit état de l'art

Le tableau 1 donne un petit historique des langages de programmation orientés objets en étudiant ces différents langages suivant l'encapsulation, l'héritage, la liaison retardée et le typage.

Langage OO	Encapsulation	Héritage	Liaison retardée	Typage
Simula (1967)	Non	Oui	Oui	Oui
Smalltalk (1970)	Oui	Oui	Oui	Oui[1]
C++ (1986-87)	Oui	Oui	Oui	Oui
Eiffel (1987-88)	Oui	Oui	Oui	Oui
Java (1995)	Oui	Oui	Oui	Oui[2]
Objective Caml (1996)	Oui	Oui	Oui	Oui[2]

TAB. 1 – Historique des langages OO.

[1] Notion de méta-classe.
[2] Typage fort.