www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Cours de programmation par objets au travers du langage Java

1ère partie : concepts de base

Table des matières

1. 1. Introduction **3**
1.1 Historique 3
1.2 Informations sur Java 3
1.3 Aspects généraux 3
1.4 Son mode de fonctionnement 4
1.5 Notion de machine virtuelle 4
2. Caractéristiques générales du langage Java **5**
2.1 Syntaxe 5
2.2 Types de données 6
2.3 Variables 7
3. L'aspect objet du langage Java **7**
3.1 Concepts de base des langages objets 7
3.2 Objet, classes et instance 7
3.3 Création d'une classe 9
3.4 Création d'objet 10
3.5 Envoi de message et méthodes 10
3.6 Héritage 12
3.7 Interfaces et implémentation 13
3.8 Relations et liens 14
3.9 Graphe de collaboration 19
4. Autres aspects du langage Java **20**
4.1 Packages 20
4.2 Applications et applets 20
5. Les tableaux **21**
6. Quelques particularités .... Erreur! Signet non défini.
7. Les packages et classes principales **21**
7.1 Package Java.lang 21

1.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

1. Introduction

1.1 Historique

Début des langages objets dans les années 60 avec Simula 67, dérivé d'Algol, conçu en Norvège à Oslo par Dahl et Nygaard. Simula est un langage destiné à faire de la simulation par évÈnements discrets en représentant directement les entités du monde réel (grue, bateau, guichet de banque, client,..) par une structure informatique munie d'un état et d'un comportement.

En 72. A partir de Sketchpad (système de manipulation d'éléments graphique) et de Simula, Alan Kay invente Smalltalk (trad. parlote ou babillage), écrit en Lisp. Il invente à ce sujet la notion d'envoi de message. Un programme est alors conçu comme un ensemble d'objets informatiques qui communiquent par envois de messages. Puis en 76, Smalltalk 76, avec la notion de métaclasse. Enfin, en 80, Smalltalk 80 fait son apparition en déclinant de manière particulièrement harmonieuse la notion du "tout objet". Smalltalk est à l'heure actuelle la référence en termes de langages concevant tout en termes d'objets.

Dans les années 70 et 80, de nombreux langages objets conçus comme des extensions de Lisp (Flavors), puis intégrés à Lisp (CLOS).

En 86-87, apparition de C++, qui s'impose de plus en plus dans le monde industriel.

1.2 Informations sur Java

On trouver une mine de renseignements sur le Web et sur Java.

Web: www.javasoft.com, java.sun.com

Tutoriel en français: www.inf.enst.fr/~charon/CoursJava

Sur yahoo: catégorie: informatique, langage de programmation, Java

1.3 Aspects généraux

1. Langage portable (disponible sous Unix (Solaris, Linus,..), Mac et Windows 95)
2. Machine virtuelle : Compilation -> interprétation
3. Syntaxe semblable à C++
4. Tout « presque » objet :
   - Pas de variables globales (autres que les noms de classes)
   - Mais il existe encore des entités primitives « non-objets » (les caractères, les nombres et les booléens).
5. Bibliothèque importante de classes
6. Gestion graphique intégrée et portable
7. Structure par « packages »
8. Exceptions
9. Threads

10.Gestion de communications sous le Web

11.en cours d'évolution (version 1.1 plus puissante, intégration des objets distribués par exemple).

1.4 Son mode de fonctionnement

1.4.1 Notion d'Applet, d'applications et de machine virtuelle

Un premier exemple d'application « bonjour » : Attention, les noms de fichiers doivent avoir le nom des classes correspondant (on ne peut avoir qu'une classe « externe » par fichier).

```
/**
•   fichier source : SalutApp.java
•   The SalutApp class implements an application that
•   simply displays "Salut les copains!" to the standard output.
*/
import java.io.* ;
class SalutdApp {
   public static void main(String[] args) {
      System.out.println("Salut les copains !") ; //imprime la chaîne
   }
}
```

Pour compiler l'application :

```
javac HelloWorldApp.java
```

Pour exécuter l'application :

```
java HelloWorldApp
```

et on voit apparaître :

```
salut les copains !
```

La version graphique (fichier Salut.java)

```
import java.applet.Applet ;
import java.awt.Graphics ;

public class Salut extends Applet {
public void paint(Graphics g) {
g.drawString(« salut les copains ! », 50, 25) ; //affiche la chaîne
   }
}
```

puis on crée le fichier Salut.html :

```
<HTML>
<HEAD>
<TITLE> A Simple Program </TITLE>
</HEAD>
<BODY>

Voici le résultat du programme :
<APPLET CODE= »Salut.class » WIDTH=150 HEIGHT=25 ALIGN=MIDDLE>
</APPLET>
</BODY>
</HTML>
```

Et il n'y a plus qu'à charger ce fichier dans un browser capable d'exécuter du Java (Netscape 2.0 et suivant, Microsoft Explorer 3.0, Hot Java, etc...).

1.5 Notion de machine virtuelle

{{à finir}}

2. Caractéristiques générales du langage Java

2.1 Syntaxe

Syntaxe semblable à celle du C. Attention, comme dans C, les ' ;' sont des terminateurs d'instructions et non des séparateurs d'instructions comme en Pascal (en particulier il y a toujours un ' ;' à la fin d'une affectation qu'il y ait un else ou non derrière).

2.1.1 Expressions

La syntaxe d'une expression est la suivante :

```
<expr> ::= (<expr>) |
           <expr> <op-bin> <expr> |
           <op-unaire> <expr> |
           <ident> |
           <litteral> |
           <envoi de message avec retour> // voir plus loin dans le cours
```

2.1.2 Opérateurs

Les mêmes qu'en C :

2.1.2.1 Relationnel

```
==, !=, >, <, >=, <=,
```

2.1.2.2 Arithmétique

```
+, *, -, /, % (reste d'une division entière, modulo)
++, -- (même règles qu'en C)
```

2.1.2.3 Logique

```
&&, ||, !
```

Opérateurs de manipulation de bits

```
op1 >> op2      shift bits of op1 right by distance op2
op1 << op2      shift bits of op1 left by distance op2
op1 >>> op2     shift bits of op1 right by distance op2 (unsigned)
&      op1 & op2       bitwise and
|      op1 | op2       bitwise or
^      op1 ^ op2       bitwise xor
~      ~ op            bitwise complement
```

2.1.3 Instructions

2.1.3.1 affectation

Comme dans C :

```
<var> = <expr>;
```

il existe aussi la possibilité d'associer un opérateur binaire à une affectation :

```
<var> += <expr>;
```

est équivalent à

```
<var1> = <var1> + <expr>;
```

Note : Il est généralement préférable d'éviter ce raccourci de notation, car il nuit généralement à la compréhension des programmes sans véritablement leur apporter d'avantages concluants.

2.1.3.2 Structures de contrôle

```
if <expr> <instr1> else <instr2>
if <expr> <instr>

Switch <expr> {
case <cste1> : <instr1> //penser au Break ! !
      ...
case <csten> : <instrn>
default : <instr par defaut>
}

while <expr> <instr>
do <instr> while <expr>

for(<init> ; <expr tant que> ; <increment>) <instr>
```

2.1.3.3 Différences avec C++

- Pas de Goto ! !
- Pas de Macros…
- Pas de struct et d'unions
- Pas de pointeurs

2.2 Types de données

Deux types de données : les types primitifs (qui ne sont pas de véritables objets), et les types par référence.

Les types primitifs (commencent par une minuscule)

Type	Taille/format	Description
`byte`	1 octet	entier d'un octet
`short`	2 octets	entier (sur 16 bits)
`int`	4 octets	entier (sur 32 bits
`long`	8 octets	entier (sur 64 bits)
`float`	4 octets	flottant sur 32 bits
`double`	8 octets	flottant sur 64 bits
`char`	2 octets (+ de 34000 car !)	caractère (Unicode)
`boolean`	1 bit	vrai ou faux (true, false)

Syntaxe (à la C) :

```
<type> <variable> = <expression>
```

Les types « par référence » qui portent sur des classes d'objets.

2.3 Variables

Types de variables :

- variable locale
- paramètre de méthodes
- attribut d'objet
- (+ paramètres d'exception, voir plus loin)

Il n'y a plus de variable globale ! ! On les remplace avantageusement par des attributs statiques de classes (voir plus loin).

Les variables peuvent être initialisées au moment de leur déclaration.

```
Ex :
int age = 1 ;
```

Portée lexicale des variables : chaque bloc ({... }) peut avoir ses propres variables locales.

3. L'aspect objet du langage Java

3.1 Concepts de base des langages objets

Ils sont au nombre de trois:

1. le concept de structuration qui lie entre eux les notions d'objet, de classe et d'instance
2. le concept de communication au travers de l'envoi de message et des méthodes
3. le concept d'héritage.

3.2 Objet, classes et instance

3.2.1 Définitions

Objet: un objet ets une structure informatique caractérisée par un état et un ensemble d'opérations exécutables par cet objet qui caractérise son comportement.

Objet = état + opérations applicables.

Exemple:

```
une Fenêtre
   etat: position, taille, bordure, titre, couleur,...
   opérations: dessiner, déplacer, agrandir...

un Icône
   etat: position, taille, pixels,...
   opérations: dessiner, déplacer, agrandir ...

un PersonnageDeJeuDeRôle
   etat: pièce, force, ptsDeVie, habileté, armes, pouvoirs,...
   opérations: allerNord, allerSud, ouvrirPorte, prendre, ...

un Fichier
   état: nom, directory, id, protections, propriétaire, contenu,.
   opérations: ouvrir, fermer, détruire, lireCar, écrireCar,...
```

On regroupe un ensemble de caractéristiques communes aux objets sous la forme de structures formelles, appelées *classes*:

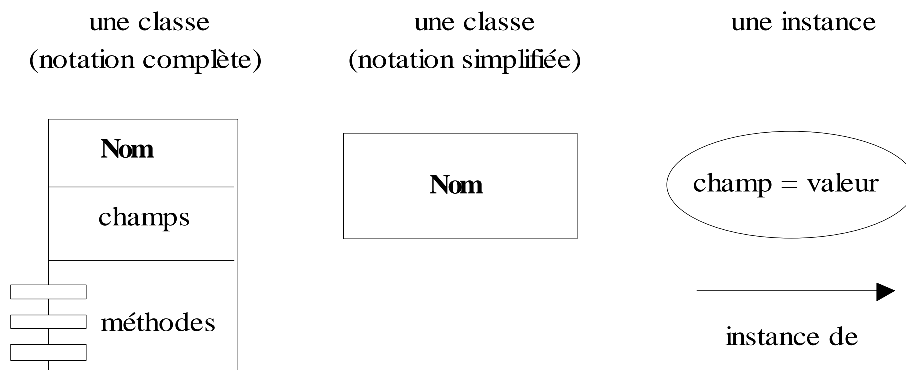
Définition: une ***classe*** est un modèle de la structure statique (les **champs** ou **attributs**) et du comportement dynamique (les **opérations** ou **méthodes**) des objets associés à cette classe, que l'on appelle ses instances.

Remarque: les classes n'ont pas nécessairement d'existence physique dans la machine.

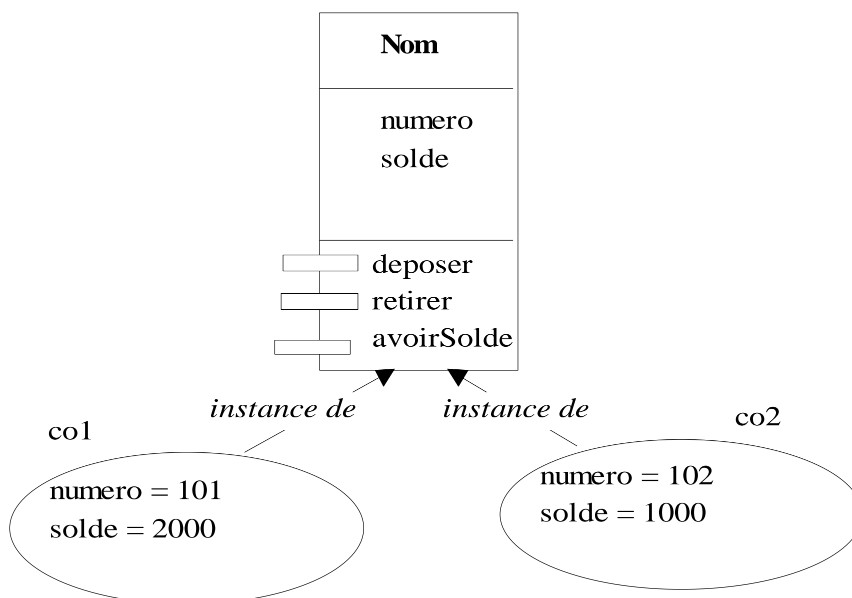
Définition: une ***instance*** est un objet, occurrence de la classe qui possède la structure définie par la classe et sur lequel les opérations définies dans la classe peuvent être appliquées.

3.2.2 Notation graphique d'une classe

Remarque : La notation utilisée est UML avec de légères variantes : les variantes portent essentiellement sur la définition des instances qui sont représentées ici par un ovale (alors qu'elle sont représentées sous la forme d'un rectangle en UML pratiquement comme les classes ce qui conduit à une certaine confusion) et surtout qui peuvent être placées dans le même diagramme qu'en UML (ce qui est impossible en UML).



3.2.2.1 Exemple



3.3 Création d'une classe

Syntaxe :

```
<déclaration> { <corps> }
```

3.3.1 La partie déclaration :

```
<modifier> class <nom> extends <nom de superclasse>
<modifier> ::= public | final | abstract
```

- `public` indique que la classe est acessible en dehors du package (voir plus loin)
- `final` indique que la classe ne peut plus avoir de sous-classes (classe hyper-concrète)
- `abstract` indique que la classe ne peut pas être instanciée.

3.3.2 Le corps d'une classe

```
<corps> ::= [ <variable> | <methode> | <constructeur> ]*
```

Le corps d'une classe est composé d'une suite de définition de champs et de méthodes.

Déclaration d'un champs

```
<modifier>* <type> <nom> = <expr>
```

Déclaration d'une méthode :

```
<modifier>* <type résultat> <nom>(<type1> <param1>,...,<typen> <paramn>)
```

Les modifieurs :

```
<modifier> : := < portée> | <allocation> | <modifiabilité>
<portée> : := public | protected | private protected | private
<allocation> : := static
<modifiabilité> : := final
```

- `public` : accessible de partout
- `protected` : accessible de partout à l'intérieur d'un package
- `private` : accessible uniquement de la classe où la variable a été définie.

`static` : indique que la variable est en fait globale à toutes les instances de la classe (ou de ses sous-classes). Elle est directement accessible par la classe. Ex : `CompteCrédit.taux`

- `final` : indique que la valeur de la variable ne peut pas être changée (représente en fait une constante).

Les constantes sont généralement déclarées comme `public static final`.

Pour les méthodes, on a aussi les modifiers :

- `abstract` : ne contient aucun code, et implique que la méthode sera définie dans les sous-classes. La classe ne peut plus être instanciée. (Si toutes les méthodes sont `abstract`, on pourra considérer la classe comme une « interface »).
- `synchronized` (pour l'utilisation avec des threads)

Un exemple de description de classe :

```
class Compte {
   int numero ;
   int solde=0 ;
```

```
      // constructeur
         Compte(String s, int c) {
            client = s; solde = c;
         }
}
```

3.4 Création d'objet

3.4.1 Constructeurs

Comme en C++ grosso modo. La différence essentielle vient de ce que toutes les instanciations se font à l'aide de la commande `new`. Attention, pour les programmeurs C..

```
Ex :
Compte c1 = new Compte(101,2000);
```

3.5 Envoi de message et méthodes

3.5.1 Envoi de message en Java

En java, l'envoi de message s'exprime sous la forme d'une notation '.' comme dans Eiffel et les Pascal Objet (et d'une certaine manière en C++ si l'allocation n'est pas faite dans le tas).

Syntaxe :

```
<envoi-message> ::= <obj>.<sel>(<arg1>,..,<argk>)
```

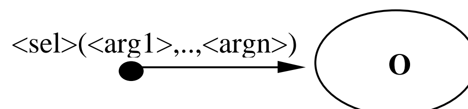
où `<obj>` est un objet receveur du message et `<sel>` est un identificateur qui correspond au nom d'une méthode définie dans la classe de `<obj>` (ou dans l'une de ses sur-classes, voir la section suivante sur l'héritage).

Attention : l'envoi de message peut prendre la forme d'une expression ou d'une instruction. Dans le premier cas, l'envoi de message retourne une valeur, dans le second, la méthode ne retourne aucun résultat.

Exemple :

```
c1.deposer(100);
int a = c1.avoirSolde();
```

3.5.2 Représentation graphique d'un envoi de message à une instance



3.5.3 Définition de méthodes

Exemple:

```
class Compte {
   int numero;
   int solde=0;

   // constructeur
   Compte(int n, int c) {
         numero = s; solde = c;
   }

   // définition des méthodes
   void deposer(int v) {
      solde = solde + v;
```

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

```
    }

    void retirer(int v) {
       solde = solde + v;
    }

    int avoirSolde() {
       return(solde);
    }

    // note on verra par la suite une meilleure technique d'affichage
    void afficher() {
       System.out.println("Compte: " + numero +
                          ", solde : " + avoirSolde())
    }
    ... // autres méthodes ...
 }
```

Utilisation des méthodes :

```
Compte co1 = new Compte(101,2000);
Compte co2 = new Compte(102,1000);

co1.deposer(1000);
co2.deposer(500);

co1.afficher();
co2.afficher();
```

Le fonctionnement du programme donnera:

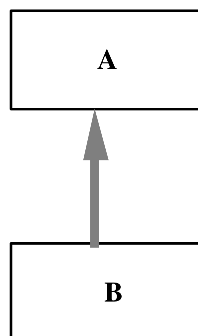
```
Compte: 101, solde : 3000
Compte: 102, solde : 1500
```

3.6 Héritage

Définition: mécanisme de recopie virtuelle par défaut des caractéristiques de la classe.

Les champs sont recopiés vers la sous-classe, mais pour les méthodes, tout se passe comme si on les recopiait d'une classe vers sa sous-classes. Mais si une sous-classe redéfinit ses méthodes, celles-ci ne sont pas recopiées.

3.6.1 Représentation graphique



Exemple: compteCredit est un compte rémunéré. Possède un taux de rémunération et une méthode CalculInterets.

```
class CompteCredit extends Compte {
      float tauxRemuneration = 0;
```

```
        void calculInterets() {
            solde = solde + solde * tauxRemuneration;
        }

    // constructeurs:
        CompteCredit(int n, int c, float t) {
            super(n,c);     // utilise le constructeur de la super-classe
            tauxRemuneration = t;
        }
}
```

Soit

```
CompteCredit co3;

co3 = new CompteCredit(101, 500, 0.05);

co3.calculInterets(); // calcul les intérêts
                  [co3 : crédit = 515]
```

En revanche:

```
co1.calculInterets(); // produit une erreur
```

3.6.2 Redéfinition de méthodes

On peut redéfinir des méthodes qui ont été définies dans des superclasses. Cela permet de masquer l'héritage des méthodes :

Exemple. Supposons que l'on veuille réaliser une classe `ComptePayant` qui facture à 5F le côut d'un retrait :

```
Classe ComptePayant extends Compte {
        void retirer(int v) {
            solde = solde + v + 5;
        }
}
```

soit

```
ComptePayant co4 ;

co4 = new ComptePayant(104, 1000);

co4.retirer(200); //on retire 205 F en fait
```

3.6.3 La variable 'this'

La variable `this` représente le receveur du message. Lorsqu'on envoie un message de la forme:

$o.m(a_1,..,a_k)$

alors la variable `this` prend la valeur `o`. Généralement, cette variable est omise lorsqu'elle se trouve en position de receveur de message (elle n'est en fait utilisée que lorsqu'on cherche à avoir une référence à l'objet lui-même).

La méthode `calculInterets` de `CompteCrédit` peut alors être définie ainsi:

```
dans CompteCrédit :
    void calculInterets() {
        this.deposer(avoirSolde()*tauxRémunération);
    }

ou:
    void calculInterets() {
        deposer(avoirSolde()*tauxRémunération);
    }
```

On a alors supprimé tous les accès directs aux variables d'instance. Celles-ci sont "encapsulées" par les méthodes `déposer`, `retirer` et `avoirSolde`.

3.6.4 Redéfinition de méthodes avec accès aux méthodes définies dans les sur-classes

Lorsqu'on redéfinit une méthode, on peut avoir besoin de faire appel à des méthodes définies dans des classes situées au-dessus. Pour cela on utilisera en Java le mot clé `super` à la place de `this`.

Exemple: La redéfinition de la méthode `retirer` dans la classe `ComptePayant` aurait pu être écrite ainsi :

```
dans ComptePayant:
   void retirer(int v) {
      super.retirer(v);
      super.retirer(5);
   }
```

3.6.5 Les constructeurs

{{à finir}}

3.7 Interfaces et implémentation

Une interface est une classe dont toutes les variables sont `static` et `final` (c'est à dire des constantes) et toutes les méthodes sont `abstract` (donc non implémentées).

```
Ex :
public interface <nom> extends <une autre interface>
```

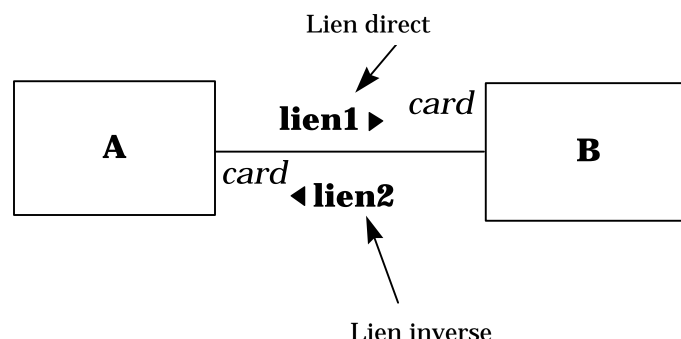
Une classe peut implémenter un ensemble d'interfaces :

```
<modifier> class <nom>
extends <uneClasse>
implements <interf1>,.., <interfn>
```

3.8 Relations et liens

Les relations expriment des connections entre objets. Une relation est composée de deux liens: un lien direct et un lien inverse.

3.8.1 Notation graphique:



Remarque: On note les deux liens directement dans la même association si cela est possible. Mais on ne note parfois qu'un seul lien si le lien inverse est évident ou n'est pas pertinent.

Exemple: une personne peut être propriéraire de plusieurs appartements, et un appartement a un propriétaire qui est une personne:

Les associations décrivent des liens entre classes qui seront ensuite *instanciées* au niveau des instances.

La cardinalité indique la manière de prendre en compte le domaine des valeurs de la relation.

Lorsque la cardinalité (qui se trouve en bout de flêche pour les liens directs, en base de la flêche pour les liens indirects) est de 1, on dit que le lien est *univalué*, si la cardinalité est '*' (on utilise parfois la lettre 'n' à la place de '*'), on dit qu'il est *univalué*.

Lorsqu'un lien est multivalué, on peut lui associer une marque indiquant la forme que prend cette multivaluation. S'agit-il d'un ensemble non-ordonné ou bien d'une suite ordonnée de valeur ?. Les marques que l'on considèrera ici sont les suivantes :

- {set} : marque par défaut. Indique qu'il s'agit d'un ensemble non-ordonné de valeurs.
- {ord} : indique qu'il s'agit d'une suite ordonnée de valeurs.
- {clé} : indique qu'il s'agit d'un ensemble de valeurs indexées par une clé (numérique ou symbolique).

On appelle *base* la classe de départ d'un lien R, et *tête* sa classe d'arrivée. Exemple: la base du lien propriétaire est `Logement`, sa tête est `Personne`.

Par exemple, la relation `propriétaire` peut être vue comme une fonction de Appartement dans Personne, et la relation inverse, `propriétaire de` peut être considérée comme une fonction de Personne dans `Collection(Logement)`, où `Collection(X)` désigne une collection des parties de `X`.

propriétaire: Appartement fi *Personne*

propriétaire de: Personne fi *Appartement*

telles que: si l'on considère que Jean possède deux appartements, appart1 et appart2, alors on a bien:

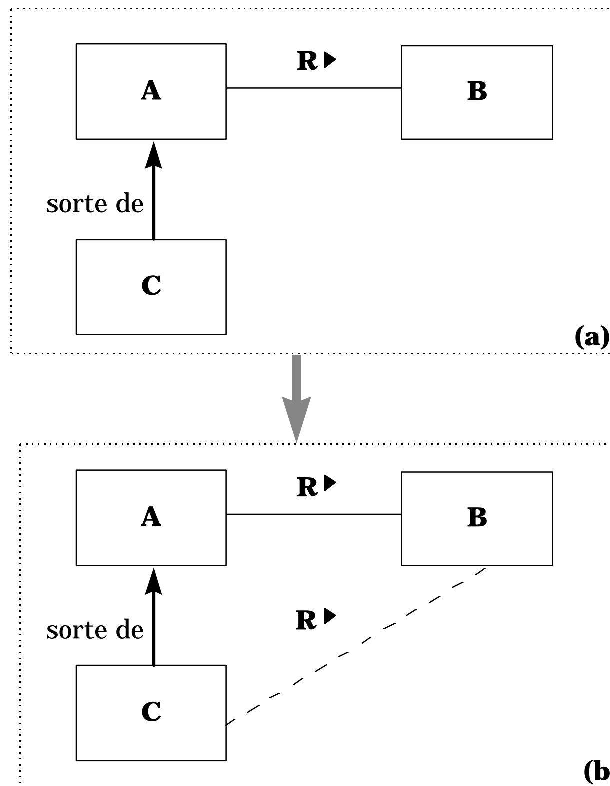
propriétaire(appart1) = Jean

propriétaire de(Jean) = {appart1, appart2}

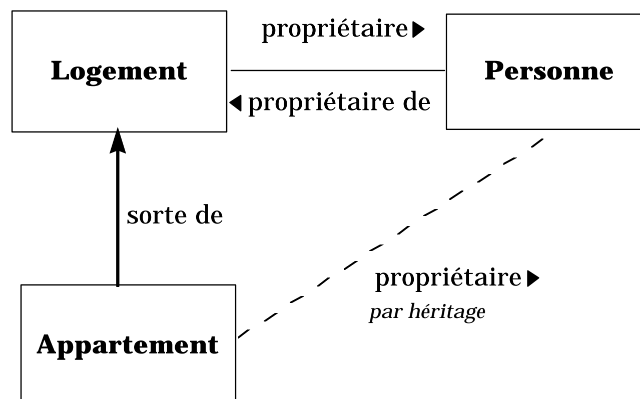
Remarque: on est parfois amené à séparer les liens directes et indirectes d'une relation. Le lien constitue donc la notion primitive, la relation une notion composée.

3.8.2 Héritage et liens

On peut propager la base des liens de la manière suivante :



Exemple: si l'on crée une classe Logement qui généralise la notion d'appartement, on a alors:

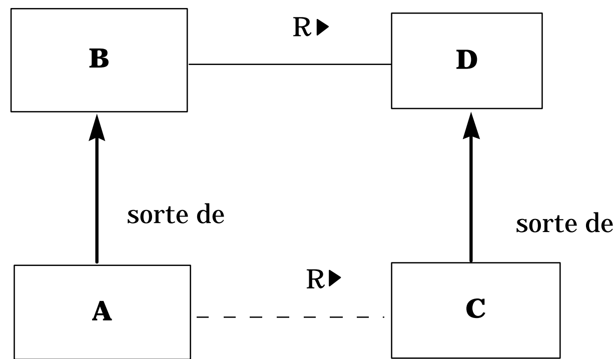


3.8.3 Cohérence dans l'héritage

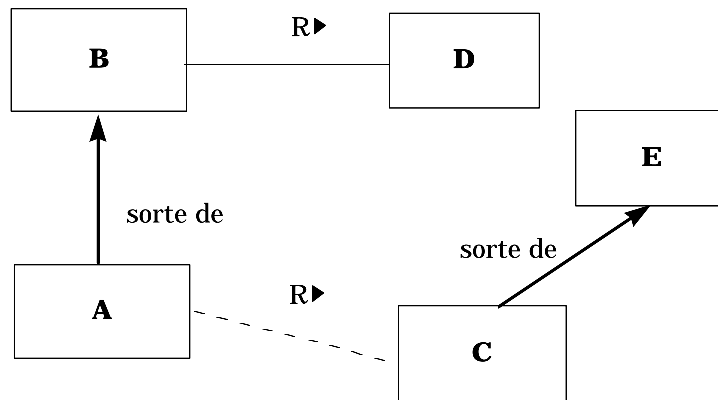
On considère qu'un lien de nom R entre deux classes B et D est valide si:

1) il n'a jamais été défini en ayant pour base une sur-classe de B,

2) il a été défini en ayant pour base une sur-classe de B, et pour tête une sur-classe de D.



Graphe cohérent

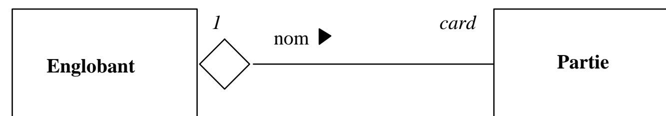


Graphe incohérent

3.8.4 Les agrégations (part of)

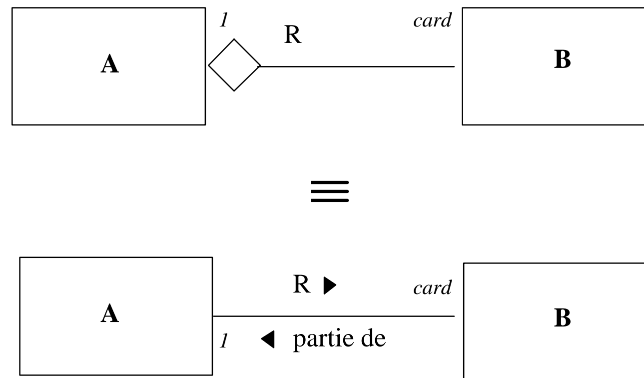
On distingue, parmi, l'ensemble des relations, les agrégations qui indiquent qu'un objet fait partie intrinsèque d'un autre objet. Exemple: le moteur fait partie de la voiture, le bras d'un corps, le clavier le CPU d'un micro-ordinateur, les feuilles d'un arbre, etc....

Ce type de relation est tellement important qu'on le note d'une manière particulière:



Par définition les agrégations sont telles qu'ils sont nécessairement de type 1-*: un objet contenu ne peut faire partie que d'un seul contenant à la fois. Si l'on a besoin de décrire des relations de cardinalité *-* (ex: une association comprend plusieurs membres, mais un membre peut appartenir à plusieurs associations), alors il sera nécessaire de les représenter sous la forme d'une association quelconque.

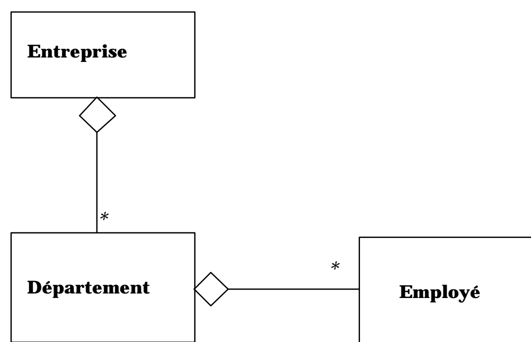
Par définition on a:



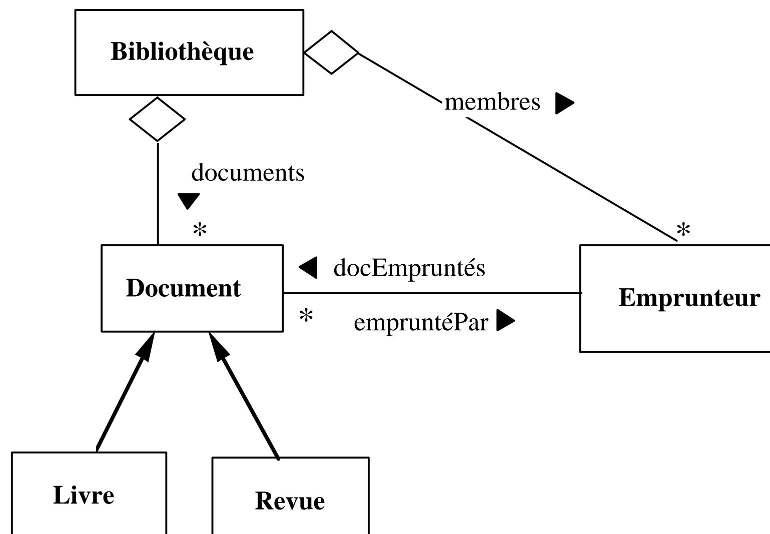
où R/R' représentent des relations telles que "contient/partie-de", "éléments/membre de", "composant/composant de", etc...

Cela signifie donc que les agrégation sont des relations ordinaires qui n'introduisent pas de sémantique différente par rapport aux associations. Ces agrégations sont utilisés simplement parce qu'elles améliorent l'intelligibilité d'une modélisation.

Exemple1 : représentation des unités d'une entreprise



Exemple2 : gestion d'une bibliothèque



Exercices:

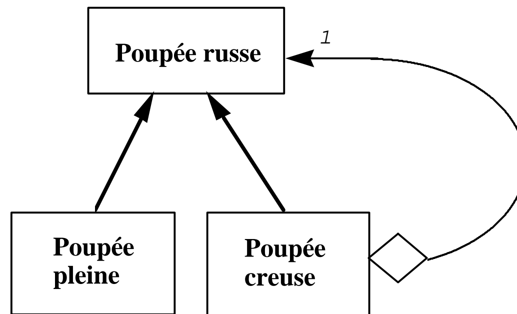
1. Introduire la classe Bibliothécaire représentant le personnel de la bibliothèque et les liens qu'ils ont avec les autres classes.

2. Définir des classes de documents qui ne peuvent pas être empruntés et modifier l'ensemble des classes en conséquence.

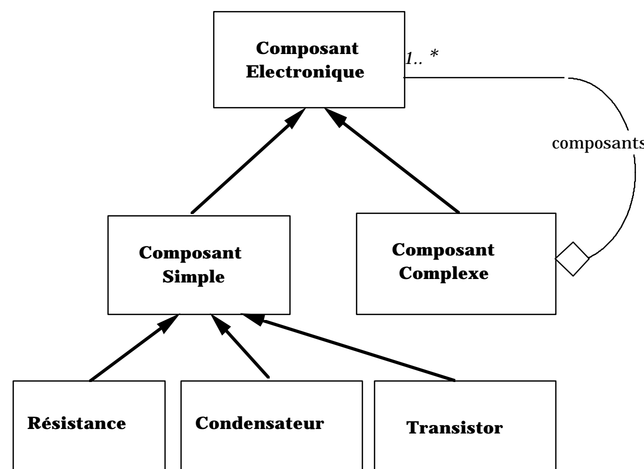
3.8.5 Récursivité des liens partitifs

Les liens partitifs (comme n'importe quel lien d'ailleurs, hormis les liens d'héritage) peuvent être récursifs, c'est à dire contenir des objets de la même classe qu'eux.

Exemple1: les poupées russes:



Exemple 2 : les composants électroniques

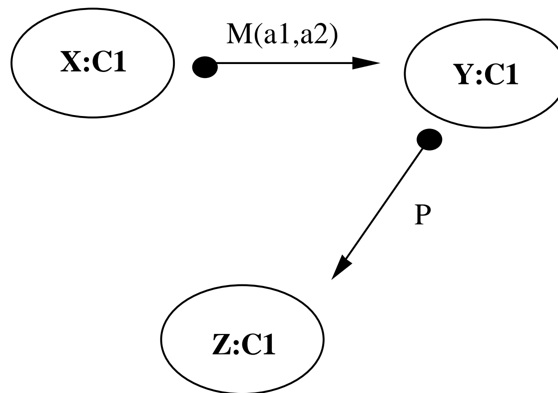


Remarque : la composition récursive est une structure très classique que l'on rencontre très souvent et notamment chaque fois que l'on doit représenter des structures récursive dont la taille n'est pas définie à l'avance.

3.9 Graphe de collaboration

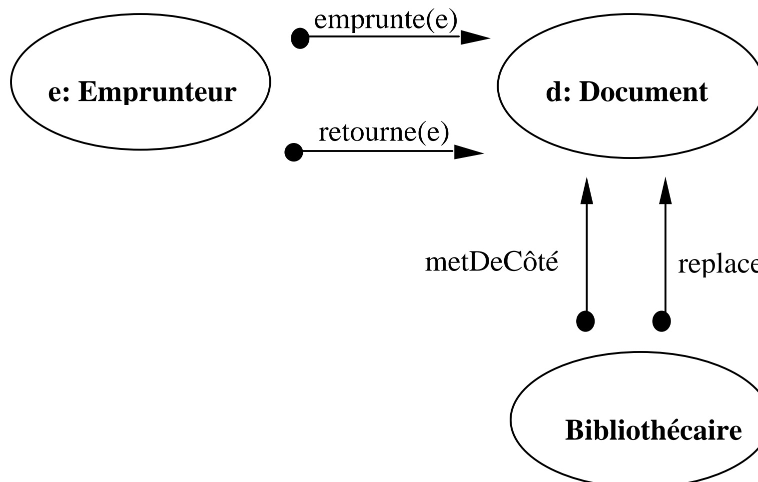
Principe: définir les communications qui existent entre objets au niveau global. Les graphes de collaboration indiquent qui communique avec qui. Ils sont utilisées pour analyser et concevoir les structures de communications entre objets.

On représente les graphes de collaboration sous la forme d'un ensemble d'instances quelconques. Ces instances sont dénommées par une variable suivie du nom de la classe. On représente les messages sous la forme de flêches dont la base comprend un rond. Par exemple pour indiquer qu'un objet X de la classe C1 envoie un message M avec les arguments a_1,a_2 à un objet Y de la classe C2, et que cet objet Y envoie un message P sans arguments à un objet Z de la classe C3, on dessine la figure suivante:



Les instances portent des variables car elles représentent des instances quelconques, au même titre qu'en mathématique on parle d'élément quelconque d'un ensemble. S'il n'existe qu'un seul objet représentatif pour une classe donnée et que cet objet n'est pas passé en argument, on peut omettre la variable.

3.9.1 Exemple: la gestion de bibliothèque



4. Autres aspects du langage Java

4.1 Packages

Les packages sont des unités de programmes indépendant. Un package peut contenir un nombre quelconque de fichiers.

On déclare un package en le faisant précéder de la mention :

```
package <nom>
```

On peut importer un package par la mention :

import <nom de package> ;

Quand un package contient plusieurs classes public (par exemple s'il est composé de plusieurs fichiers comme le package java.util), on écrit :

import <nom de package>.*

exemple :

```
import java.awt.* ; // pour charger l'interface graphique
import java.io.* ; // pour charger le module d'entrée sortie standard.
```

Exemple : si l'on désire importer la classe Date :

```
import java.util.date ;  // importe la classe Date du package java.util
import java.util.* ; // importe toutes les classes du package java.util
```

4.2 Applications et applets

4.2.1 Application

Les applications sont lancées par la méthode main associée à la classe de l'application, de type :

```
public static void main(String[] args)
```

Les arguments sont placés dans le tableau args.

4.2.2 Applets

Les applets répondent aux messages suivants :

- `init()` : appelée une fois à la création de l'applet.
- `start()` : appelée une fois après l'init, et appelée à chaque fois que l'applet a été stoppée.
- `stop()` : appelée lorsqu'on change de page du browser par exemple,
- `destroy()` : appelée lorsqu'on quitte le browser.
- `paint()` : appelée chaque fois que le navigateur doit se redessiner.

Remarque: on reparlera des Applets dans le cadre des interfaces graphiques…

5. Les tableaux

Les tableaux sont des objets « référencés » qui ont un sens particulier pour le compilateur. En tant qu'objets référencés, ils sont donc créés par l'opérateur new.

```
char s[] = new char[30] ;
Compte tableauCompte[] = new Compte[n] ;
```

Le premier élément d'un tableau est 0. Le dimensionnement se fait lors de la création du tableau (et non pas lors de la déclaration).

Les tableaux multi-dimensionnés sont créés ainsi :

```
int i[][] = new int[3][4] ;
```

On peut aussi écrire les tableaux en plaçant les crochets après le type : les deux lignes suivantes sont équivalentes.

```
int iarray[] ;
int[] iarray ;
```

De même pour les méthodes :

```
byte    f( int n )[] ;
byte[]  f( int n ) ;
```

On accède normalement aux tableaux par les indices. On peut placer (comme pour les variables) des objets appartenant à une sous-classe de la classe de définition du tableau.

```
tableauCompte[3] = new CompteCrédit("Dupond", 500);
```

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

La longueur d'un tableau est obtenu à partir de l'attribut `length`.

Ex : affichage de tous les éléments d'un tableau :

```
for(int i = 0 ; i < a.length ; i++)
   tableauCompte[i].afficher() ;
```

6. Les packages et classes principales

6.1 Package Java.lang

6.1.1 Classe : Java.lang.Object

méthodes principales :

- `boolean equals(Object o)`
- `String toString()`
- `Object clone()`

```
Ex :
Truc a = new Truc() ;
Truc b = a ; // a et b pointent vers le même objet
System.out.println(a == b) ; // imprime true
Truc c = a.clone() ; // a et b pointent vers des objets différents
System.out.println(a == c) ; // imprime false
```

- `String toString()` : retourne une chaîne représentant l'objet (pour être redéfinie dans les sous-classes).
- `Class getClass()` ; retourne la classe (en fait le descripteur de la classe de l'objet (instance de Class).

6.1.2 Classe : Java.lang.String

Les chaînes de caractères jouent un rôle intermédiaire entre les entités primitives et les objets référencés.

Les objets instances de String sont des chaînes de caractères dont le contenu ne peut être modifié (mais on peut combiner des chaînes et sélectionner des sous-chaînes).

```
String s1 = "bonjour" ; // crée un chaîne s
String s2 = s + "ça va ?" ; //crée une chaîne par concaténation.
String s3 = new String(34) ; //crée une chaîne représentant le nombre 35
String s4 = s3 + 56 ; //crée la chaîne 3456
```

Mais :

```
String s5 = 4 ;// provoque une erreur à la compilation
String s6 = new String(s1) ; // crée une chaîne qui est une copie de s1
```

Méthodes :

- `boolean s1.equals(String s2)` : indique si s1 est équivalent (même suite de caractères) que s2.
- `int s1.length()` : retourne la longueur de la chaîne.
- `char s.charAt(int i)` : retourne le caractère situé à l'index i.

Autre : cf. le manuel de références.

Conversion des chaînes vers et à partir d'autres objets.

- Objet vers chaîne : utilisation de la méthode toString

```
Truc a = new Truc() ;
String s = a.toString() ; // mais toString doit être défini pour Truc
```

- Entité primitive vers chaîne : utilisation de la méthode valueOf.

```
Ex :
    String.valueOf(4) // ou
    new String(4)     // ou
    Integer.toString(4) //
```

- Chaîne vers objet (nécessite sa définition)
- Chaîne vers entité primitive : utilisation des « wrappers »

```
Ex :
    Integer.valueOf("45");
```

6.1.3 Classe : java.lang.StringBuffer

Implémente des chaînes de caractères dont le contenu peut être modifié.

6.1.3.1 Constructeurs

```
StringBuffer() ;                    // Constructs an empty string buffer.
public StringBuffer(int length)     // Constructs a string buffer with
                                    // the specified initial length.
public StringBuffer(String str)     // Constructs a string buffer with
                                    // the specified initial value.
```

6.1.3.2 Méthodes

```
StringBuffer s.append(Object o)     // ajoute la représentation de o
                                    //au StringBuffer s. Retourne s.
StringBuffer insert(int i,Object o) // insère une chaîne à l'index i
```

Exemple d'utilisation d'un StringBuffer : inversion d'une chaîne de caractères :

```
public static String reverseIt(String source) {
    int i, len = source.length() ;
    StringBuffer dest = new StringBuffer(len) ;
    for (i = (len - 1) ; i >= 0 ; i--) {
    dest.append(source.charAt(i)) ;
                    }
    return dest.toString() ;
}
```