

Modélisation et programmation par Objets 1

HLIN406

Marianne Huchard, Clémentine Nebut

15 janvier 2019

Ces notes de cours sont (perpétuellement) en cours de réalisation. Elles sont donc susceptibles de contenir des erreurs ou des imprécisions, ou d'être incomplètes. Elles ne dispensent en aucun cas d'une présence en cours, TD ou TP.

Table des matières

1	Introduction	4
1.1	Pourquoi vous parler de conception par objets?	4
1.2	Modélisation des systèmes informatique	4
1.2.1	Notion de modélisation	4
1.2.2	UML, un langage de modélisation	5
1.3	Concrétisation en Java	6
2	Classes et instances	7
2.1	Les classes et instances en UML	7
2.2	Les paquetages en UML	10
2.3	Classes, instances et paquetages en Java	11
2.3.1	Types de base en Java	11
2.3.2	Écriture des classes	11
2.3.3	Création des instances	12
2.3.4	Accès aux attributs	12
3	Opérations et méthodes	13
3.1	Classes, opérations et méthodes	13
3.2	Opérations en UML	13
3.3	Méthodes en Java	16
3.3.1	Déclaration de méthodes	16
3.3.2	Exécution d'un premier programme	17
3.3.3	Les accesseurs	17
3.3.4	Quelques instructions de base	18
3.3.5	Structures de contrôle	21
4	Spécialisation/généralisation et héritage	24
4.1	Généralisation - Spécialisation	24
4.1.1	Classer les objets	24
4.1.2	Discriminants et contraintes	24
4.2	Hiérarchie des classes et héritage dans Java	27
4.3	Redéfinition de méthodes - Surcharge - Masquage	29
4.4	Les constructeurs	30
4.5	Protections	31
4.6	Classes et méthodes abstraites	32
4.7	Le polymorphisme	32
4.7.1	Transformation de type ou Casting	32
4.7.2	Polymorphisme des opérations	33

5	Associations et collections	37
5.1	Associations et liens	37
5.2	Associations et attributs	39
5.3	Associations particulières	39
5.4	Traduction des associations en Java	42
5.5	Tableaux et collections	43
5.5.1	Les tableaux	43
5.5.2	Les collections Java	44
5.5.3	Les Vecteurs	44
5.5.4	Vector vs ArrayList	45
5.5.5	Les dictionnaires associatifs	45
5.5.6	Exemple d'utilisation d'un dictionnaire associatif	46
6	Les interfaces	49
6.1	Interfaces en Java	49
6.1.1	Définition d'une interface	49
6.1.2	Spécialisation d'une interface	50
6.1.3	Implémentation d'une interface	51
6.1.4	Code utilisant les interfaces	53
6.2	Représentation en UML	54
6.3	La place des interfaces dans l'API de Java	55
7	Le modèle d'utilisation en UML	57
8	Objets et récursivité	60
8.1	Introduction à la récursivité	60
8.2	La récursivité en modélisation et en programmation	61
8.2.1	Méthodes statiques récursives	61
8.2.2	Méthodes d'instance récursives	62
8.2.3	Méthodes d'instance récursive sur des structures récursives	63
8.2.4	Exploitation de l'héritage pour la définition des structures récursives	65
8.3	Conclusion	67

Chapitre 1

Introduction

1.1 Pourquoi vous parler de conception par objets ?

Les approches par objets sont un succès dans l'histoire de l'informatique (30 dernières années) :

- elles sont fondées sur quelques idées simples qui consistent à décrire un système avec des représentations informatiques proches des entités du problème et de sa solution : si on parle d'un système bancaire on décrira des objets *Comptes bancaires* dans le langage informatique ; cela facilite la communication entre les intervenants d'un projet ;
- elles ont des avantages reconnus en termes de :
 - facilité du codage initial,
 - stabilité du logiciel construit car les objets manipulés sont plus stables que les fonctionnalités attendues,
 - aisance à réutiliser les artefacts existants et ...
 - à maintenir le logiciel, le corriger, le faire évoluer ;
- elles ont connu un fort développement dans les langages de conception, de programmation, les bases de données, les interfaces graphiques, les systèmes d'exploitation, etc.

Objectifs Ce cours présente les concepts essentiels de l'approche objet en s'appuyant sur un langage de modélisation (UML) et un langage de programmation (Java). Le langage de programmation permettra de rendre plus concrets les concepts étudiés.

1.2 Modélisation des systèmes informatique

1.2.1 Notion de modélisation

La modélisation est la première activité d'un informaticien face à un système à mettre en place.

Modéliser consiste à produire une représentation simplifiée du monde réel pour :

- accumuler et organiser des connaissances,
- décrire un problème,
- trouver et exprimer une solution,
- raisonner, calculer.

Il s'agit en particulier de résoudre le hiatus entre :

- d'un côté le monde réel, complexe, en constante évolution, décrit de manière informelle et souvent ambiguë par les experts d'un domaine
- de l'autre le monde informatique, où les langages sont codifiés de manière stricte et disposent d'une sémantique unique.

La modélisation est une tâche rendue difficile par différents aspects :

- les spécifications parfois imprécises, incomplètes, ou incohérentes,
- taille et complexité des systèmes importantes et croissantes,
- évolution des besoins des utilisateurs,
- évolution de l'environnement technique (matériel et logiciel),
- des équipes à gérer plus grandes, avec des spécialisations techniques, nécessaires du fait de la taille des logiciels à construire, mais le travail est plus délicat à structurer.

Pour faire face à ces difficultés, les méthodes d'analyse et de conception proposent des guides structurant :

- organisation du travail en différentes phases (analyse, conception, codage, etc.) ou en niveaux d'abstraction (conceptuel, logique, physique),
- concepts fondateurs : par exemple les concepts de fonction, signal, état, objet, classe, etc.,
- représentations semi-formelles, documents, diagrammes, etc.

Dans cette approche le langage de modélisation est un formalisme de représentation qui facilite la communication, l'organisation et la vérification.

1.2.2 UML, un langage de modélisation

UML (Unified Modeling Language) est un langage de modélisation graphique véhiculant en particulier

- les concepts des approches par objets : classe, instance, classification, etc.
- intégrant d'autres aspects : associations, fonctionnalités, événements, états, séquences, etc.

UML est né en 1995 de la fusion de plusieurs méthodes à objets incluant OOSE (Jacobson), OOD (Booch), OMT (Rumbaugh).

UML propose d'étudier et de décrire un système informatique selon quatre points de vue principaux qui correspondent à quatre modèles (voir figure 1.1).

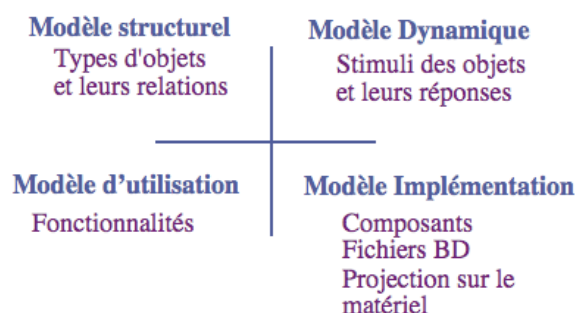


FIGURE 1.1 – Vue générale des modèles UML

Chaque modèle est une représentation abstraite d'une réalité, il fournit une image simplifiée du monde réel selon un point de vue. Il permet :

- de comprendre et visualiser (en réduisant la complexité),

- de communiquer (à partir d'un langage commun à travers un nombre restreint de concepts),
- de mémoriser les choix effectués,
- de valider (contrôle de la cohérence, simuler, tester).

Dans chaque modèle, on écrit un certain nombre de diagrammes qui décrivent chacun certains aspects particuliers (voir Figure 1.2).

Diagrammes (représentations graphiques de modèles)

Diagrammes de classes d'instances	Diagrammes de collaboration de séquences d'états, d'activités
Diagrammes de cas d'utilisation	Diagrammes de déploiement de composants

FIGURE 1.2 – Vue générale des diagrammes UML

L'un des atouts majeurs d'UML est que le langage sert dans la plupart des étapes de construction d'un logiciel, son rôle s'arrête juste pour la phase de codage (implémentation dans un langage de programmation).

1.3 Concrétisation en Java

Comme nous l'avons évoqué au début de l'introduction, nous utiliserons le langage Java pour projeter les constructions faites lors de la modélisation, donnant ainsi un aspect plus concret à ce cours.

Java est un langage de programmation à objets relativement récent (1991) et qui fait la synthèse de quelques-uns des langages existant à l'époque de sa création.

- Il emprunte une grande partie de sa syntaxe à C++ ;
- il recherchait à l'origine une plus grande simplicité que C++ ;
- il permet de s'abstraire des problèmes de gestion de la mémoire ;
- il n'est pas « tout objet » et n'a pas les capacités de réflexivité des langages à objets les plus avancés, mais en a cependant plus que C++ ;
- il fonctionne à l'aide de deux programmes, un compilateur et un interprète, et ce qui a fait en partie son succès est la possibilité d'avoir cet interprète dans tous les navigateurs internet.

Chapitre 2

Classes et instances

2.1 Les classes et instances en UML

Le modèle statique (ou structurel) se compose de deux types de diagrammes.

- Les diagrammes d’objets ou d’instances décrivent les objets du domaine modélisé et les éléments de la solution informatique (par exemple des personnes, des comptes bancaires), ainsi que des liens entre ces objets (par exemple le fait qu’une personne possède un compte bancaire) ;
- Les diagrammes de classes sont une abstraction des diagrammes d’objets : ils contiennent des classes qui regroupent des objets ayant des caractéristiques communes et des relations entre ces classes. De manière duale, les diagrammes d’instances doivent être conformes aux diagrammes de classes.

Voyons de plus près ces deux types de diagrammes.

Lors de l’analyse, notre esprit raisonne à la fois :

- par identification d’objets de base (Estelle, la voiture d’Estelle),
- par utilisation de ces objets comme des prototypes (la voiture d’Estelle vue comme une voiture caractéristique, à laquelle ressemblent les autres voitures, moyennant quelques modifications),
- par regroupement des objets partageant des propriétés structurelles et comportementales en classes.

Le deuxième mode de pensée a été exploré par une branche des langages à objets appelée les langages à prototypes qui sont moins connus que les langages dits à classes auxquels nous nous intéressons dans ce cours.

Dans le présent contexte des langages à classes, on dira souvent qu’une classe est un concept du domaine sur lequel porte le logiciel (voiture ou compte bancaire) ou du domaine du logiciel (par exemple un type de données abstrait tel que la pile). Une classe peut se voir selon trois points de vue :

- un aspect *extensionnel* : l’ensemble des objets (ou instances) représentés par la classe,
- un aspect *intensionnel* : la description commune à tous les objets de la classe, incluant les données (partie statique ou attributs) et les opérations (partie dynamique),
- un aspect *génération* : la classe sert à engendrer les objets.

À gauche de la figure 2.1, nous présentons une classe en notation graphique UML. Elle ne contient que des propriétés structurelles qui s’appellent des **attributs**. Ce sont des données décrivant l’objet (ici, le type, la marque et la couleur, toutes de type chaîne de caractères) et qui forment son **état**. En notation UML les diagrammes de classes montrent

donc essentiellement l'aspect intensionnel des classes.

À droite de cette même figure 2.1, nous voyons un objet (ou instance) tel qu'en contiennent les diagrammes d'instances. Il s'agit ici d'une instance de la classe **Voiture**, qui se trouve décrite par une valuation des attributs. En l'absence d'ambiguïté, les noms des attributs peuvent être omis.

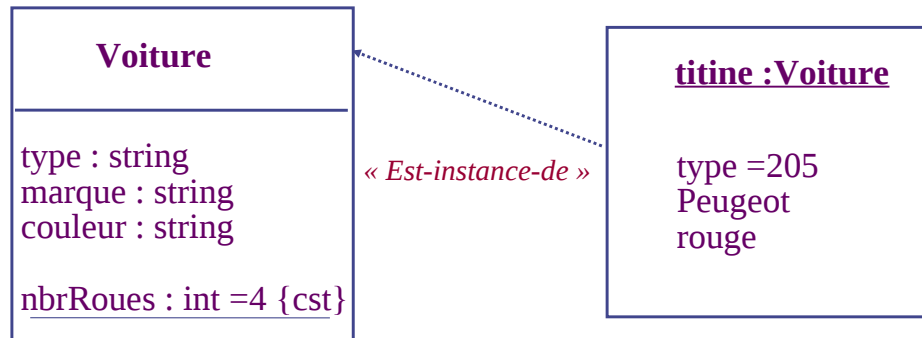


FIGURE 2.1 – Classe (à gauche) et objet/instance (à droite)

Les attributs peuvent être décrits par de nombreux autres éléments que le type (voir exemple figure 2.2). La syntaxe est la suivante :

[visibilité] [/]nom[:type] [[multiplicité]] [= valeurParDéfaut]

où *visibilité* $\in \{+, -, \#, \sim\}$, et *multiplicité* définit une valeur (1, 2, n, ...) ou une plage de valeurs (1..*, 1..6, ...).

- la visibilité exprime la possibilité de référencer l'attribut suivant les contextes
 - Public. + est la marque d'un attribut accessible partout (public)
 - Privé. - est la marque d'un attribut accessible uniquement par sa propre classe (privé)
 - Package. ~ est la marque d'un attribut accessible par tout le paquetage
 - Protected. # est la marque d'un attribut accessible par les sous-classes de la classe
- le nom est la seule partie obligatoire de la description
- la multiplicité décrit le nombre de valeurs que peut prendre l'attribut (à un même moment)
- le type décrit le domaine de valeurs
- la valeur initiale décrit la valeur que possède l'attribut à l'origine
- des propriétés peuvent préciser si l'attribut est constant ({**constant**}), si on peut seulement ajouter des valeurs dans le cas où il est multi-valué ({**addOnly**}), etc.

Certains attributs peuvent être descriptifs de la classe elle-même plutôt que d'une instance, leur valeur est alors partagée par toutes les instances : ce sont les attributs *de classe* (voir figure 2.3). On les distingue des autres car ils sont soulignés.

Enfin certains attributs ont la particularité que leur valeur peut être déduite de la valeur d'autres attributs ou d'autres éléments décrivant la classe. Ce sont des attributs *dérivés* (voir figure 2.4).

Les énumérations

Une énumération est un type de données dont on peut énumérer toutes les valeurs possibles. Par exemple :

- la civilité d'une personne qui a pour valeurs possibles : Mme, M, Mlle

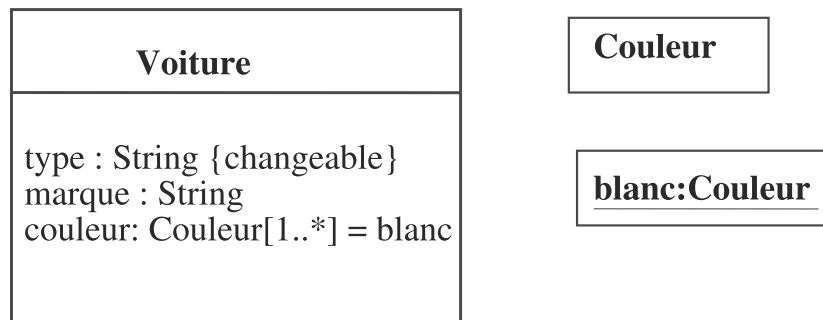


FIGURE 2.2 – Détails sur la syntaxe de description des attributs

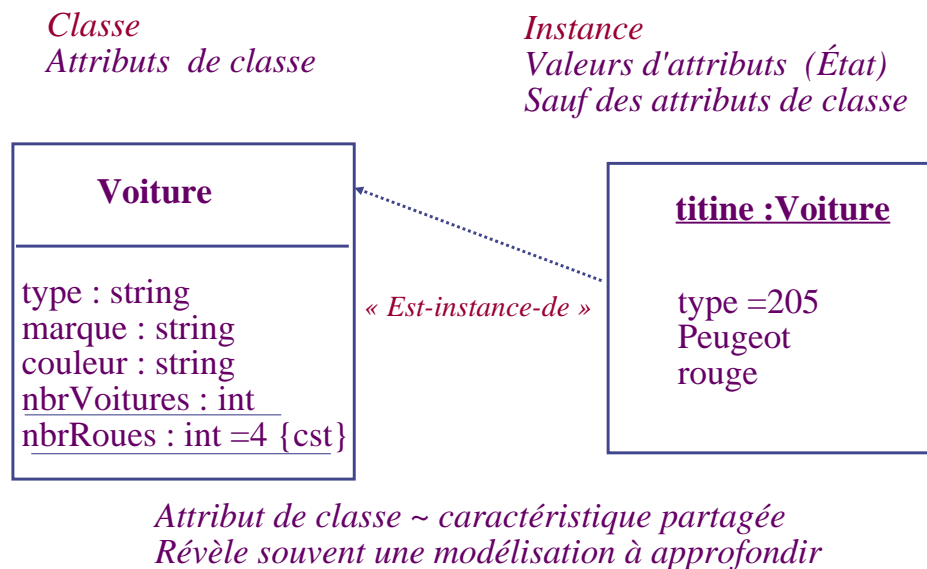


FIGURE 2.3 – Attributs de classe

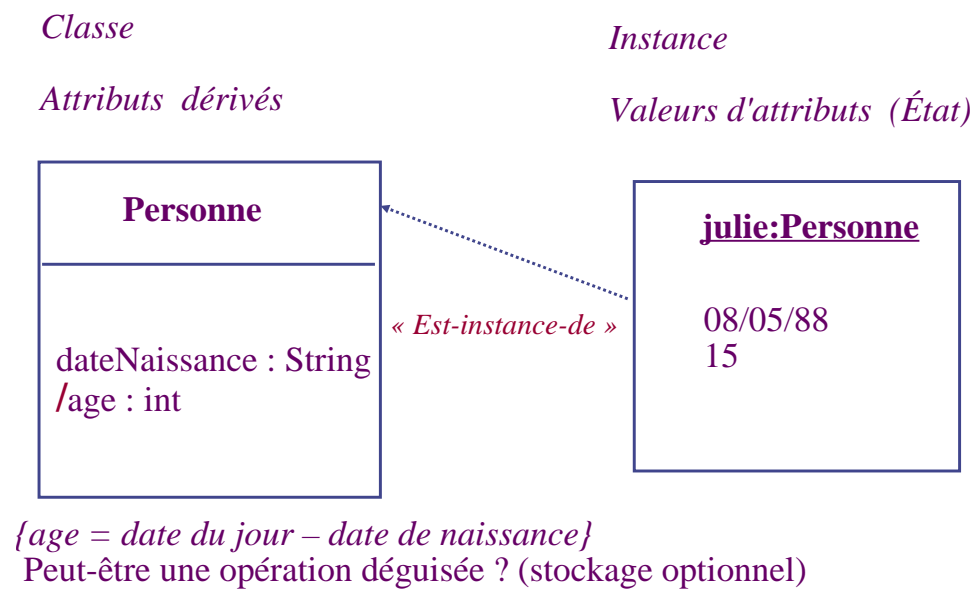


FIGURE 2.4 – Attribut dérivé

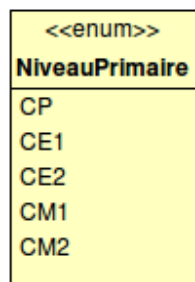


FIGURE 2.5 – Énumération en UML

- les stations de ski d'un grand domaine qui ont pour valeurs possibles : Valmorel, Combelouvière, Saint-François-Longchamp
- les niveaux à l'école primaire qui ont pour valeurs : CP, CE1, CE2, CM1, CM2

2.2 Les paquetages en UML

Un paquetage est un regroupement logique d'éléments UML, par exemple de classes. Les paquetages servent à structurer une application et sont utilisés dans certains langages, notamment Java, ce qui assure une bonne traçabilité de l'analyse à l'implémentation. Ils seront liés par des relations de dépendance dont nous reparlerons plus loin. Par exemple on regroupe dans le paquetage **VenteAutomobile** toutes les classes qui concernent ce domaine (Figure 2.6).

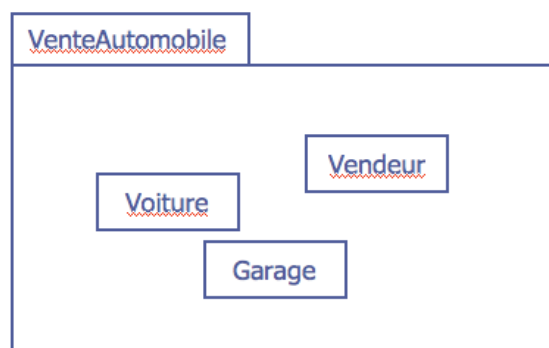


FIGURE 2.6 – Paquetages en UML

2.3 Classes, instances et paquetages en Java

À ce stade de notre cours, les classes et instances se traduisent assez directement dans le langage Java, procurant ce que l'on appelle des types construits, d'un plus haut niveau d'abstraction que les types de base (entiers, booléens, caractères), et plus proches des concepts manipulés en analyse.

2.3.1 Types de base en Java

Nous commençons cependant par évoquer ces types de base, car ils serviront en particulier à typer les attributs. Ils sont les suivants.

- **boolean**, constitué des deux valeurs **true** et **false**. Les opérateurs se notent :

Java	non	égal	différent	et alors / et	ou sinon / ou
	!	==	!=	&& &	

- **int**, entiers entre -2^{31} et $2^{31} - 1$
- **float**, **double**, ces derniers sont des réels entre $-1.79E + 308$ et $+1.79E + 308$ avec 15 chiffres significatifs.
- **char**, caractères représentés dans le système Unicode. Les constantes se notent entre apostrophes simples, par exemple 'A'.
- **String**, qui n'est pas ... un type de base, c'est en réalité une classe mais nous rangeons ce type ici du fait de son usage très courant. Les chaînes de caractères constantes se notent entre guillemets, par exemple "hello world". Les opérations seront déterminées par les méthodes de cette classe. Nous les verrons plus loin.

2.3.2 Écriture des classes

Nous donnons ici une traduction simplifiée à l'extrême de la classe *Voiture* qui serait incluse dans le paquetage **ExemplesCours1**. Notez les modificateurs **static** pour les attributs de classe, et **final** pour traduire le fait qu'un attribut est constant (plus précisément, on ne peut l'initialiser qu'une fois, mais l'initialisation peut être séparée de la déclaration : elle peut se faire par exemple dans un constructeur). Les attributs ont une valeur initiale implicite : 0 pour les nombres et null pour les références (variables désignant des objets).

```
package ExemplesCours1;
public class Voiture
```

```
{
private String type; // null
private String marque; // null
private String couleur; // null
private static int nbrVoitures; // 0
private static final int nbrRoues = 4;
}
```

2.3.3 Création des instances

L’instruction suivante permet de déclarer une variable nommée `titine`.

```
Voiture titine;
```

Puis nous pouvons la créer.

```
titine = new Voiture();
```

`titine` doit être comprise comme une variable dont la valeur est une désignation de l’objet.

2.3.4 Accès aux attributs

Pour écrire des valeurs dans les attributs d’instance, nous utilisons des instructions d’affectation.

```
titine.type = "205";
titine.marque = "Peugeot";
titine.couleur = "rouge";
```

Grâce à elles, notre instance Java a à présent les mêmes valeurs que notre instance UML.

Pour écrire des valeurs dans les attributs de classe, on préfixe le nom de l’attribut par le nom de la classe.

```
Voiture.nbrVoitures = 3;
```

Toutes ces instructions ne peuvent bien entendu être écrites que dans les contextes où les attributs sont accessibles.

Définition et utilisation d’énumérations

Le listing 2.1 montre un exemple de définition d’énumération en Java, puis une manipulation d’une valeur pour cette énumération (ici, le niveau CP).

Listing 2.1 – énumérations en Java

```
1 public enum Niveau{
2 CP, CE1, CE2, CM1, CM2;
3 }
4
5 ...
6
7 Niveau n=Niveau.CP;
8 ...
```

Chapitre 3

Opérations et méthodes

3.1 Classes, opérations et méthodes

Nous avons vu au chapitre précédent que l'on pouvait définir des classes, et leur associer des attributs. On peut ainsi définir ce qu'**est** un objet, mais pas ce qu'il fait, ou peut faire : c'est le rôle des opérations (terme UML) ou méthodes (terme Java).

Les méthodes / opérations définissent des comportements des instances de la classe. Par exemple, on a défini une classe voiture, on va maintenant voir comment exprimer ce que peut faire une voiture : klaxonner, fournir une assistance au parking, etc.

Les méthodes / opérations peuvent manipuler les attributs, ou faire appel à d'autres méthodes de la classe. Elles peuvent être paramétrées et retourner des résultats.

3.2 Opérations en UML

Les opérations sont les seuls éléments dynamiques du diagramme de classes. Elles se notent dans le compartiment inférieur des classes (voir figure 3.1).

Détail des opérations en UML (voir figure 3.2)

La syntaxe pour la déclaration des opérations est la suivante :
[visibilité] nom (liste-paramètres) [: typeRetour] [liste-propriétés]

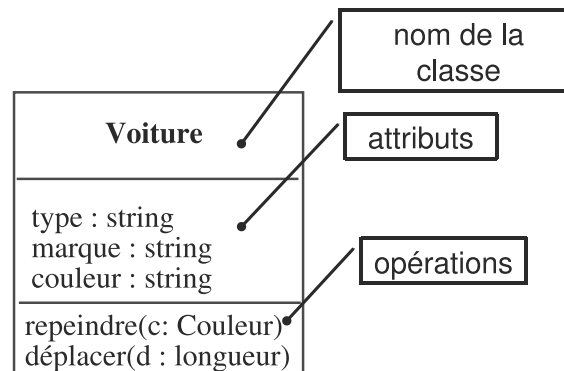


FIGURE 3.1 – Les opérations dans les classes UML

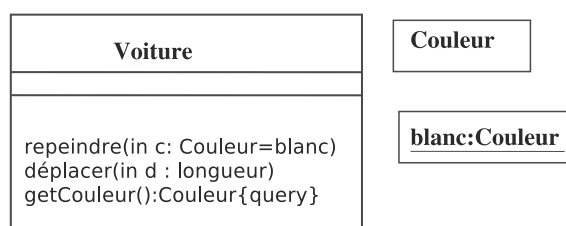


FIGURE 3.2 – Exemple d’opérations en UML

où la syntaxe de chaque paramètre est :

[direction] nom : type[[multiplicité]] [= valeurParDéfaut] [liste-propriétés]

avec *direction* $\in \{in, out, inout\}$, et *multiplicité* définit une valeur (1, 2, n, ...) ou une plage de valeurs (1..*, 1..6, ...). Une opération a un nom. On essaie en général de lui donner le nom portant le plus de sémantique possible : on évite d’appeler les opérations o1, o2, ou op, mais plutôt : klaxonner, déplacer, repeindre.

Visibilité Une opération a une visibilité.

- Publique. Dénoté +. Signifie que cette opération pourra être appelée par n’importe quel objet.
- Privée. Dénoté -. Signifie que cette opération ne pourra être appelée que par des objets instances de la même classe.
- Paquetage. Dénoté ~. Signifie que cette opération ne pourra être appelée que par des objets instances de classes du même paquetage.
- Protégée. Dénoté #. Signifie que cette opération ne pourra être appelée que par des objets instances de la même classe ou d’une de ses sous-classes (on verra plus tard ce que cela signifie exactement).

Paramètres Une opération peut avoir des paramètres. On peut spécifier le mode de passage d’un paramètre :

- in** le paramètre est une entrée de l’opération, et pas une sortie : il n’est pas modifié par l’opération. C’est le cas le plus courant. C’est aussi le cas par défaut en UML.
- out** le paramètre est une sortie de l’opération, et pas une entrée. C’est utile quand on souhaite retourner plusieurs résultats : comme il n’y a qu’un type de retour, on donne les autres résultats dans des paramètres out.
- inout** le paramètre est à la fois entrée et sortie.

Propriétés Une opération peut avoir des propriétés précisant le type d’opération, par exemple {query} spécifie que l’opération n’a pas d’effet de bord, ce n’est qu’une requête. Les propriétés sont placées entre accolades. Ces accolades signalent une valeur marquée (tagged value). Une valeur marquée a un nom, et peut contenir une valeur. Une valeur marquée peut être attachée à n’importe quel élément de modèle UML. Il existe des valeurs marquées pré-définies par UML, mais aussi définies par l’utilisateur, les valeurs marquées font donc partie des mécanismes d’extension d’UML.

Opérations de classe

Une opération de classe est une opération qui ne s’applique pas à une instance de la classe : elle peut être appelée même sans avoir instancié la classe. Une opération de classe

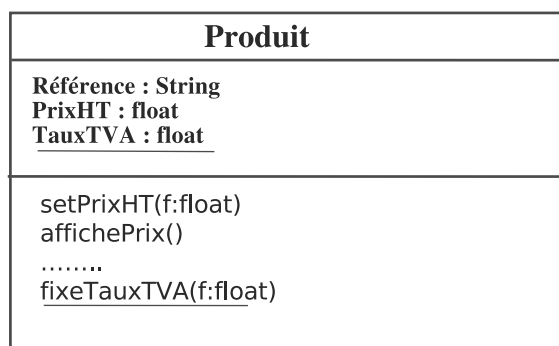


FIGURE 3.3 – Opérations de classe

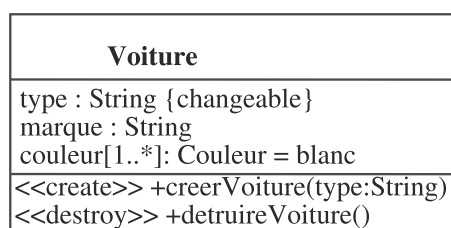


FIGURE 3.4 – Constructeurs et destructeurs en UML

ne peut accéder qu'à des attributs et opérations de classe. En UML, les opérations de classe sont soulignées (voir Figure 3.3).

Constructeurs et destructeurs

Il existe des opérations particulières qui sont en charge de la gestion de la durée de vie des objets : les constructeurs et les destructeurs. Un constructeur est une opération particulière d'une classe qui est l'opération qui permet de créer des instances de cette classe. Symétriquement, un destructeur est une opération particulière qui permet de détruire une instance de cette classe. En UML, pour préciser qu'une opération est un constructeur ou un destructeur, on place devant l'opération les stéréotypes <<create>> ou <<destroy>> (voir figure 3.4). Les stéréotypes se présentent comme des chaînes entre chevrons. Ce sont des étiquettes qui peuvent être attachées à n'importe quel élément de modèle UML, et qui donnent une sémantique particulière à l'élément de modèle.

Le corps des opérations en UML

Nous avons vu jusqu'ici comment spécifier les signatures des opérations en UML, mais pas ce que font exactement les opérations, leur comportement. En UML, il n'y a pas à proprement parler de langage d'action permettant de spécifier le comportement des opérations. On peut par contre utiliser des diagrammes dynamiques pour les spécifier (nous verrons ces diagrammes plus tard). On peut aussi documenter l'opération avec du pseudo-code, dans une note de commentaire. On peut en effet attacher à tout élément de modèle UML une note contenant du texte (voir figure 3.5).

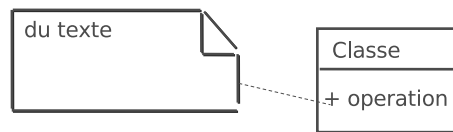


FIGURE 3.5 – Note UML

3.3 Méthodes en Java

Nous allons voir comment écrire des méthodes en Java.

Listing 3.1 – Classe Voiture en java

```
1 package ExemplesCours2;
2 public class Voiture
3 {
4     private String type;
5     private String marque;
6     private String couleur;
7     private static int nbrVoitures;
8     private static final int nbrRoues = 4;
9
10    public Voiture(String leType, String laMarque, String couleur){
11        type=leType;
12        marque=laMarque;
13        this.couleur=couleur;
14    }
15
16    public static int getNbrRoues(){
17        return nbrRoues;
18    }
19
20    public String getMarque(){
21        return marque;
22    }
23
24    private void setMarque(String m){
25        marque=m;
26    }
27
28    public void repeindre(String c){
29        couleur=c;
30    }
31 }
```

3.3.1 Déclaration de méthodes

Le listing 3.1 illustre quelques déclarations de méthodes. On notera :

- la déclaration de constructeur : en Java, le constructeur d’une classe doit avoir le même nom que la classe, et il n’y a pas de type de retour.
- il n’y a pas vraiment de destructeur en Java. Il existe une méthode particulière nommée `finalize` qui est appelée quand le ramasse-miettes détruit l’objet car il n’est plus référencé.

- la déclaration de méthode de classe, avec le mot clef **static**. Pour appeler une méthode de classe, on préfixe le nom de l'opération par le nom de la classe ou par une instance de la classe si on en a une (on préférera le premier procédé).
- l'utilisation des mots clefs **private** et **public** pour définir la visibilité des méthodes
- il n'y a pas en Java la distinction entre paramètre in, out, ou inout.

On peut définir dans une même classe plusieurs méthodes portant le même nom, à condition que leur signature soient différentes. On peut en effet écrire une méthode `int add(int a, int b)` et une méthode `float add(float a, float b)` dans une même classe. Cette possibilité s'appelle la surcharge.

3.3.2 Exécution d'un premier programme

Le listing 3.2 donne un exemple de programme utilisant la classe **Voiture**.

Listing 3.2 – Utilisation de la classe Voiture en java

```
1 package ExemplesCours2;
2 public class essaiVoiture{
3     public static void main(String[] arg){
4         Voiture v=new Voiture("C3", "Citroen", "rouge");
5         int nb=v.getNbrRoues();
6         System.out.println("Ma_"+v.getMarque()+"_a_"+nb+"_roues");
7     }
8 }
```

On notera :

- la méthode bizarre appelée **main** : c'est le point d'entrée de notre programme, c'est-à-dire que c'est elle qui est appelée quand on fait :
 `> java ExemplesCours2.essaiVoiture`
Cette méthode est statique : on n'a pas besoin de créer d'instance de la classe **essaiVoiture** pour utiliser la méthode **main**. Le paramètre correspond à ce qui est donné comme arguments en ligne de commande, ils sont stockés sous forme d'un tableau de chaînes. Nous verrons les tableaux ultérieurement.
- la concaténation de chaînes pour l'affichage, et la traduction automatique d'entiers en chaînes.

3.3.3 Les accesseurs

Les accesseurs sont des méthodes qui permettent d'accéder aux attributs, en lecture et en écriture. En Java, par convention ils sont notés **getAtt** et **setAtt** pour un attribut **att**. Leur signature est la suivante pour un attribut **att** de type **T** :

```
1 T getAtt()
2 void setAtt(T valeur)
```

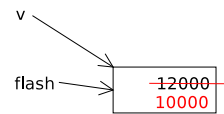
Un exemple est donné au listing 3.4. Le **get** peut permettre de faire des statistiques sur les accès à l'attribut. Le **set** peut permettre d'effectuer des vérifications sur les valeurs, ou bien encore de prendre en charge des attributs dérivés. Un attribut dérivé peut être implémenté par une méthode, ou un attribut mis à jour quand cela est nécessaire. Ainsi le **set** d'un attribut peut permettre d'aller mettre à jour un attribut dérivé qui en dépend.

```

public class Voiture{
    public int prix;
    public Voiture(int p){
        prix=p;
    }
}

public class Expertise{
    public void expertiser(v:Voiture){
        v.prix=10000;
    }
    ...
    Voiture flash=new Voiture(12000);
    ...
    expertiser(flash);
}

```



Pendant l'exécution de la méthode `expertiser`, `v` et `flash` désignent le même objet.

FIGURE 3.6 – Passage de paramètres par référence

3.3.4 Quelques instructions de base

Affectation

L'affectation se note `=` (voir par exemple ligne 11 du listing 3.1).

Déclaration de variables locales

Dans le corps d'une méthode, on peut déclarer des variables locales. Par exemple :

```

int i;
int j=0;
Voiture v;

```

Pour les variables locales, on ne précise pas de visibilité : la portée de ces variables s'arrête à la fin de la méthode. On peut tout de suite initialiser les variables locales déclarées. Les variables locales n'ont pas de valeur initiale implicite.

Création de nouvelles instances

Pour créer de nouvelles instances d'une classe, on utilise le constructeur de la classe. On doit aussi utiliser le mot clef `new`.

```

Voiture v;
v=new Voiture("C4", "Citroen", "bleu");

```

Le passage de paramètres

Dans le corps d'une méthode, les paramètres sont comme des variables locales. Tout se passe comme si on avait des variables locales déclarées au début de la méthode, et qu'au début de méthode on affectait les valeurs des paramètres effectifs à ces variables locales. Les paramètres de type simple (types de base en Java comme `int` et `boolean`, dont le nom commence par une minuscule) sont passés par valeur. Tous les autres paramètres sont passés par référence (voir Figure 3.6)¹. Nous allons illustrer le passage de paramètres sur un petit exemple, donné listings 3.3 et 3.4.

1. On dit parfois que la référence est passée par valeur

Listing 3.3 – Echange.java

```
1 package ExemplesCours2;
2 public class Echange{
3     public void fauxEchange(int a, int b){
4         System.out.println("a="+a+" b="+b);
5         int vi=a;
6         a=b;
7         b=vi;
8         System.out.println("a="+a+" b="+b);
9     }
10
11     public void pseudoEchange(MonInt a, MonInt b){
12         System.out.println("a.getEntier="+a.getEntier()+" b.getEntier="+b
13             .getEntier());
14         int vi=a.getEntier();
15         a.setEntier(b.getEntier());
16         b.setEntier(vi);
17         System.out.println("a.getEntier="+a.getEntier()+" b.getEntier="+b
18             .getEntier());
19     }
20
21     public static void main(String[] args){
22         int x=2, y=3;
23         System.out.println("x="+x+" y="+y);
24         Echange echange=new Echange();
25         echange.fauxEchange(x,y);
26         System.out.println("x="+x+" y="+y);
27
28         MonInt xx=new MonInt(2);
29         MonInt yy=new MonInt(3);
30         System.out.println("xx.getEntier="+xx.getEntier()+" yy.getEntier="+
31             yy.getEntier());
32         echange.pseudoEchange(xx,yy);
33         System.out.println("xx.getEntier="+xx.getEntier()+" yy.getEntier="+
34             yy.getEntier());
35     }
36 }
```

Listing 3.4 – MonInt.java

```
1 package ExemplesCours2;
2 public class MonInt{
3
4     private int entier;
5
6     public MonInt(int e){
7         entier=e;
8     }
9
10    public int getEntier(){
11        return entier;
12    }
13
14    public void setEntier(int e){
```

```
15     entier=e;
16   }
17 }
```

Désignation de l'instance courante

En Java, on désigne l'instance courante par le mot-clef **this**. On a besoin de cette désignation par exemple quand il y a conflit de noms (comme par exemple au listing 3.1) ou quand on veut passer l'instance courante en paramètre d'une méthode.

L'instruction return

L'instruction **return** permet de retourner un résultat (voir l'exemple du listing 3.1). Un **return** provoque une sortie immédiate de la méthode : on ne doit donc jamais mettre de code juste sous un **return**, il ne serait pas exécuté. On ne peut utiliser un **return** que dans une méthode pour laquelle on a déclaré un type de retour, et bien sûr le type de l'objet retourné doit être cohérent avec le type de retour déclaré.

Les commentaires

Il existe plusieurs formats pour les commentaires :

```
// ceci est un commentaire (s'arrête à la fin de la ligne)
/* ceci est un autre commentaire
   qui s'arrête quand on rencontre le marqueur de fin que voilà */
/** ceci est un commentaire particulier, utilisé par l'utilitaire javadoc **/
```

Affichage

On peut afficher des données sur la console grâce à une bibliothèque java.

```
System.out.println("affichage puis passage à la ligne");
System.out.print("affichage sans ");
System.out.print("passer à la ligne");
```

La méthode toString()

Toutes les classes disposent implicitement d'une méthode **String toString()** qui retourne une chaîne de caractères dont le rôle est de représenter une instance ou son état sous une forme lisible et affichable. Si on ne définit pas de méthode **toString** dans une classe, la méthode par défaut est appelée, elle retourne une désignation de l'instance. Il est conseillé de définir une méthode **toString** pour chaque classe. Nous verrons plus tard quel mécanisme se cache derrière cette méthode par défaut ... La méthode **toString** est illustrée au listing 3.5.

Listing 3.5 – Personne.java

```
1 package ExemplesCours2;
2 public class Personne{
3     private String nom;
4     private int numSecu;
5 }
```

```

6  public String toString() {
7      String result=nom+" "+age;
8      return result;
9  }
10 }
```

3.3.5 Structures de contrôle

Conditionnelles

Conditionnelle simple Syntaxe générale :

Listing 3.6 – Conditionnelle en Java

```

1  if (expression booléenne) {
2      bloc1
3  }
4  else {
5      bloc2
6  }
```

- La condition doit être évaluable en true ou false et elle est obligatoirement entourée de parenthèses.
- Les points-virgules sont obligatoires après chaque instruction et interdits après }.
- Si un bloc ne comporte qu'une seule instruction, on peut omettre les accolades qui l'entourent.
- Les conditionnelles peuvent s'imbriquer.

Listing 3.7 – Conditionnelle en Java

```

1  int a =3;
2  int b =4;
3  System.out.print("Le_plus_petit_entre_"+a+"_et_"+b+"_est:_");
4  if (b < a ) {
5      System.out.println(b);
6  }
7  else { System.out.println(a);
8  }
```

L'opérateur conditionnel () ? ... : ... Le : se lit *sinon*.

```

1  System.out.println( (b < a) ? b : a );
2  int c = (b < a) ? a-b : b-a ;
```

L'instruction de choix multiples Syntaxe générale :

```

1  switch (expr entiere ou caractere ou enumeration ) {
2      case i :
3      case j :
4          [bloc d'instructions]
5      .....break;
```

```
6  case_k:
7  ...
8  default:
9  ...
10 }
```

- L’instruction **default** est facultative ; elle est à placer à la fin. Elle permet de traiter toutes les autres valeurs de l’expression n’apparaissant pas dans les cas précédents.
 - Le **break** est obligatoire pour ne pas traiter les autres cas.
-

```
1  int mois, nbJours;
2  switch (mois) {
3      case 1:
4      case 3:
5      case 5:
6      case 7:
7      case 8:
8      case 10:
9      case 12:
10     nbJours = 31;
11     break;
12     case 4:
13     case 6:
14     case 9:
15     case 11:
16     nbJours = 30;
17     break;
18     case 2:
19         if ( ((annee % 4 == 0) && !(annee % 100 == 0)) || (annee % 400 == 0)
20             )
21             nbJours = 29;
22         else
23             nbJours = 28;
24         break;
25     default nbJours=0;
26 }
```

Boucles

while Syntaxe :

```
1  while (expression) {
2      bloc
3  }
```

```
1  int max = 100, i = 0, somme = 0;
2  while (i <= max) {
3      somme += i;      // somme = somme + i
4      i++;
5  }
```

do while Syntaxe :

```
1      do
2          { bloc }
3      while (expression)
```

```
1 int max = 100, i = 0, somme = 0 ;
2 do {
3     somme += i ;
4     i ++;
5 }
6 while ( i <= max );
```

for Syntaxe :

```
1 for ( expression1 ; expression2 ; expression3 ){
2     bloc
3 }
```

- utilisée pour répéter N fois un même bloc d'instructions
- **expression1** : initialisation. Précise en général la valeur initiale de la variable de contrôle (ou compteur)
- **expression2** : la condition à satisfaire pour rester dans la boucle
- **expression3** : une action à réaliser à la fin de chaque boucle. En général, on actualise le compteur.

```
1 int somme = 0, max = 100;
2 for (int i =0 ; i <= max ; i++ ) {
3     somme += i ;
4 }
```

Instructions de rupture

- Pas de **goto** en Java ;
- instruction **break** : on quitte le bloc courant et on passe à la suite ;
- instruction **continue** : on saute les instructions du bloc situé à la suite et on passe à l'itération suivante.

Chapitre 4

Spécialisation/généralisation et héritage

4.1 Généralisation - Spécialisation

Nous abordons dans ce chapitre, un des atouts majeurs de la programmation objet et qui a pour point de départ un « double » mécanisme d'inférence intellectuelle : la généralisation et la spécialisation, deux mécanismes relevant d'une démarche plus générale qui consiste à « classifier » les concepts manipulés.

4.1.1 Classer les objets

La généralisation est un mécanisme qui consiste à réunir des objets possédant des caractéristiques communes dans une nouvelle classe plus générale appelée **super-classe**.

Prenons un exemple décrit par la figure 4.1. Nous disposons dans un diagramme de classes initial d'une classe **Voiture** et d'une classe **Bateau**. Une analyse de ces classes montre que leurs objets partagent des attributs et des opérations : on abstrait une super-classe **Véhicule** qui regroupe les éléments communs aux deux sous-classes **Voiture** et **Bateau**. Les deux sous-classes héritent les caractéristiques communes définies dans leur super-classe **Véhicule** et elles déclarent en plus les caractéristiques qui les distinguent (ou bien elles redéfinissent selon leur propre point de vue une ou plusieurs caractéristiques communes).

Le mécanisme dual, la spécialisation est décrit à partir de l'exemple de la figure 4.2.

Ici la spécialisation consiste à différencier parmi les bateaux (la distinction s'effectuant selon leur type), les sous-classes **Bateau_à_moteur** et **Bateau_à_voile**. La spécialisation peut faire apparaître de nouvelles caractéristiques dans les sous-classes.

Il faut retenir que du point de vue de la modélisation, une classe C_{mere} généralise une autre classe C_{fille} si l'ensemble des objets de C_{fille} est inclus dans l'ensemble des objets de C_{mere} .

Du point de vue des objets (instances des classes), toute instance d'une sous-classe peut jouer le rôle (peut remplacer) d'une instance d'une des super-classes de sa hiérarchie de spécialisation-généralisation.

4.1.2 Discriminants et contraintes

Les relations de spécialisation/généralisation peuvent être décrites avec plus de précision par deux sortes de description UML : les contraintes et les discriminants.

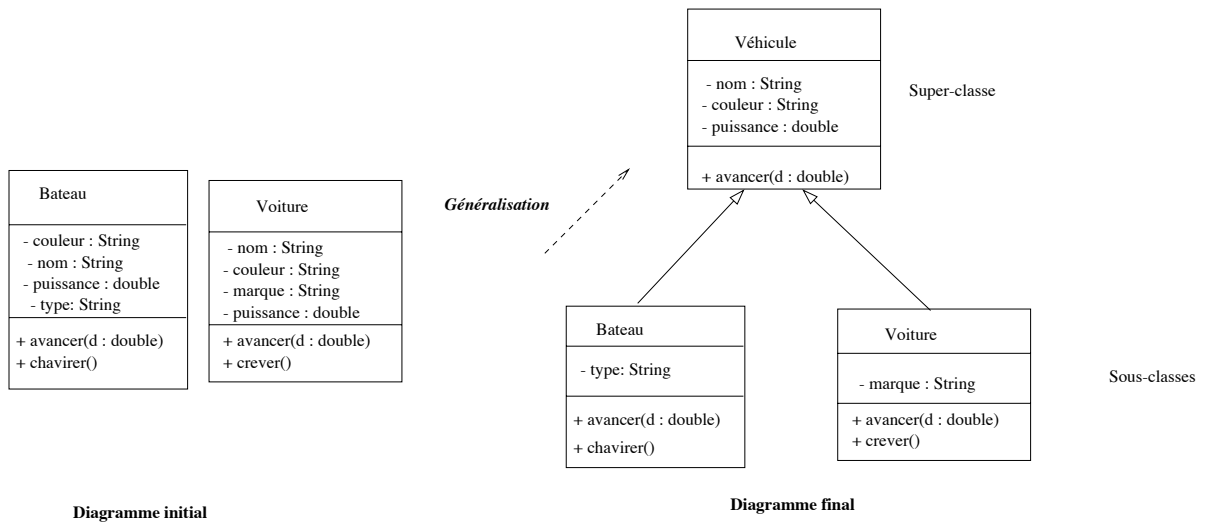


FIGURE 4.1 – Une généralisation

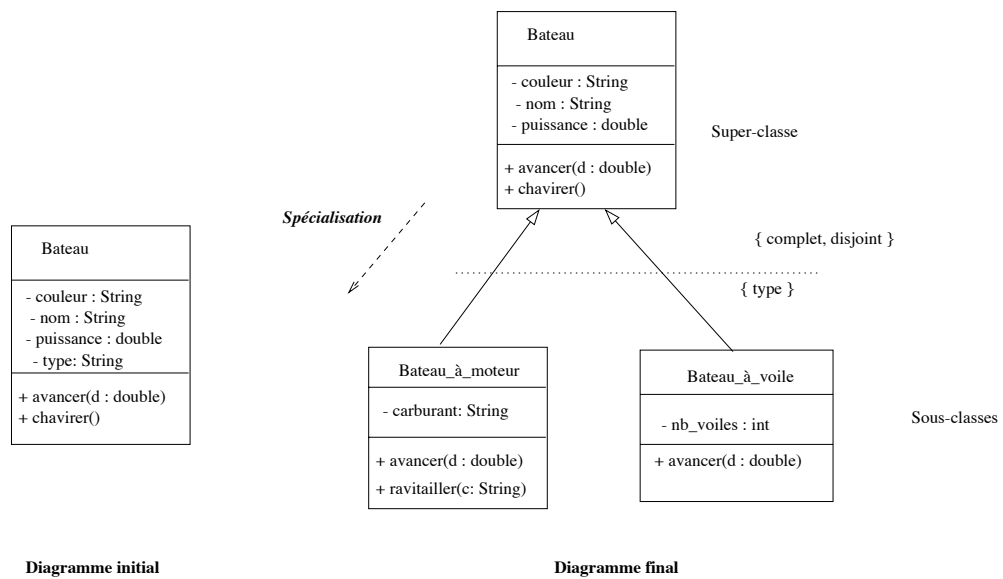


FIGURE 4.2 – Une spécialisation

Discriminant Les discriminants sont tout simplement les critères utilisés pour classer les objets dans des sous-classes. Ils étiquettent ainsi les relations de spécialisation et doivent correspondre à une classe du modèle. Dans l'exemple de la figure 4.3, deux critères différents de classification sont utilisés : **TypeContrat** partage les employés en salariés et vacataires, tandis que **RetraiteComplémentaire** partage les employés en cotisants et non cotisants. Notez qu'UML autorise qu'un objet soit classé à la fois comme cotisant et comme vacataire : on parle alors de multi-instanciation.

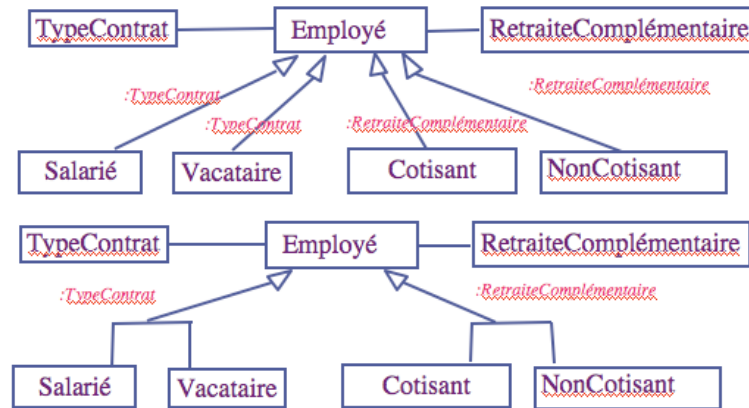


FIGURE 4.3 – Discriminants

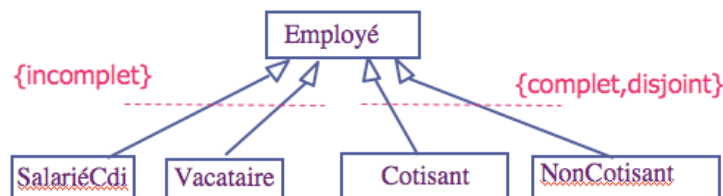


FIGURE 4.4 – Contraintes

Contraintes Les contraintes décrivent la relation entre un ensemble de sous-classes et leur super-classe en considérant le point de vue des extensions (ensemble d'instances des classes).

Il en existe quatre, dont trois sont illustrées figure 4.4 et une figure 4.5 :

- **incomplete (incomplet)** : l'union des extensions des sous-classes est strictement incluse dans l'extension de la super-classe ; par exemple il existe des employés qui ne sont ni salariés, ni vacataires ;
- **complete (complet)** : l'union des extensions des sous-classes est égale à l'extension de la super-classe ; par exemple tout employé est cotisant ou non cotisant ;
- **disjoint** : les extensions des sous-classes sont d'intersection vide ; par exemple aucun employé n'est cotisant et non cotisant ;
- **overlapping (chevauchement)** : les extensions des sous-classes se rencontrent, par exemple si on avait spécialisé une classe **Vehicule** en **VehiculeTerrestre** et **VehiculeAquatique**, certains véhicules se trouveraient dans les extensions des deux sous-classes.

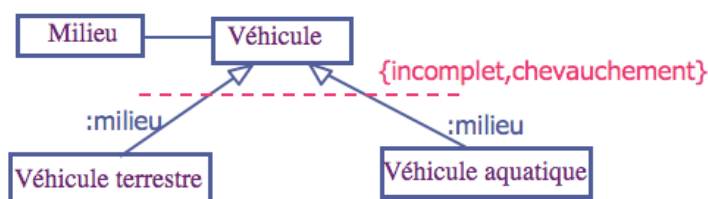


FIGURE 4.5 – Contraintes

4.2 Hiérarchie des classes et héritage dans Java

Pour les langages à objets, le programmeur définit des hiérarchies de classes provenant d'une conception dans laquelle il a utilisé le principe de généralisation-spécialisation. On dit le plus souvent que la sous-classe hérite de la super-classe, ou qu'elle étend la super-classe ou encore qu'elle dérive de la super-classe.

Un des avantages des langages à objets est que le code est réutilisable. Il est commode de construire de gros projets informatiques en étendant des classes déjà testées. Le code produit devrait être plus lisible et plus robuste car on peut contrôler plus facilement son évolution, au fur et à mesure de l'avancement du projet. En effet, grâce à la factorisation introduite par la spécialisation-généralisation, on peut modifier une ou plusieurs classes sans avoir à les réécrire complètement (par exemple en modifiant le code de leur super-classe).

Du point de vue des objets (instances de classes), pour Java, une instance d'une sous-classe possède la partie structurelle définie dans la superclasse de sa classe génitrice plus la partie structurelle définie dans celle-ci. Au niveau du comportement, les objets d'une sous-classe héritent du comportement défini dans la superclasse (ou la hiérarchie de super-classes) avec quelques possibilité de variations, comme nous le verrons plus tard. Au niveau de l'exécution, les langages à objets utilisent un mécanisme d'héritage (ou résolution de messages) qui consiste à résoudre dynamiquement l'envoi de message sur les objets (instances), c'est-à-dire à trouver et exécuter le code le plus spécifique correspondant au message.

En Java,

- toutes les classes dérivent de la classe `Object`, qui est la racine de toute hiérarchie de classes.
- une classe ne peut avoir qu'une seule super-classe. On parle d'*héritage simple*. Il existe des langages à objets, tels que C++ ou Eiffel qui autorisent l'héritage multiple.
- le mot clef permettant de définir la généralisation-spécialisation entre les classes est `extends`.

Pour la suite du chapitre nous prenons une hiérarchie de classes représentée dans le diagramme 4.6.

Nous commençons par définir la structure de la hiérarchie : vous voyez comment les entêtes des classes incluent le mot clef `extends` pour traduire en Java la relation d'héritage.

```

package Prog.MesExemples;

public class Personne{
.....} // fin de la classe Personne

.....

```

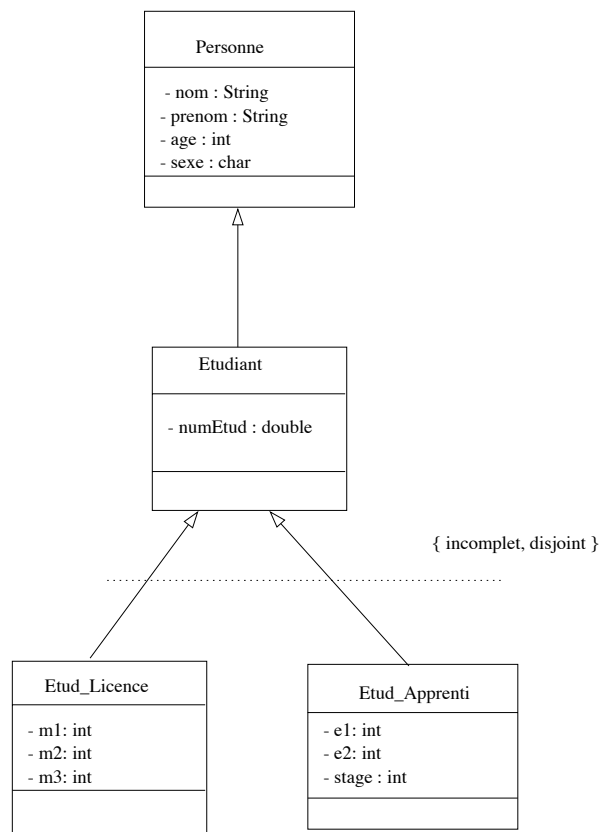


FIGURE 4.6 – Une hiérarchie de classes.

```

public class Etudiant extends Personne {

    .....

}    // fin de la classe Etudiant

.....
public class Etud_Licence extends Etudiant {

    .....

}    // fin de la classe Etud_Licence

.....
public class Etud_Apprenti extends Etudiant {

    .....
    
```

```
}    // fin de la classe Etud_Apprenti
```

Tous les étudiants sont des personnes, mais on peut les spécialiser en étudiants de licence ou étudiants apprentis. Les attributs définis dans la classe **Personne** sont nom, prénom, age, sexe. Bien sûr, on définit pour cette classe les constructeurs, les accesseurs, et une opération ou méthode **incrémenterAge()** (cette méthode vieillit d'un an).

La classe **Etudiant** possède l'attribut supplémentaire **numEtud** qui représente le numéro de l'étudiant. La classe **Etudiant** définit des opérations de calculs de moyenne, d'admission, de mention.

Les étudiants de licence (classe **Etud_Licence**) suivent 3 modules (M1, M2, M3) de coefficient 1. Ils sont admis si leur moyenne générale est supérieure ou égale à 10 et ils conservent les modules obtenus s'ils ne sont pas admis.

Les étudiants apprentis (classe **Etud_Apprenti**) suivent 2 modules (E1, E2) de coefficient 1 et un stage de coefficient 2. Ils sont admis si leur moyenne est supérieure ou égale à 10 et si leur note de stage est supérieure à 8.

Une instance étudiant apprenti (ou une instance d'étudiant licence) a la possibilité d'utiliser des méthodes définies dans **Personne** :

```
//exemple de code pouvant figurer dans la méthode main d'une classe application
Etud_Apprenti a = new Etud_Apprenti ("Einstein", "Albert", 22, 'M', 123, 8, 8, 10);
// L'age de cet apprenti est 22 ans
a.incrémenterAge();
System.out.println(a.getName()+"était âgé de " + a.getAge()+" ans");
```

A l'exécution, on obtient :

```
Einstein Albert était âgé de 23 ans
```

4.3 Redéfinition de méthodes - Surcharge - Masquage

Une opération (ou une méthode) est déclarée dans une classe, possède un nom, un type de retour et une liste de paramètres, on parle de signature pour désigner l'ensemble des informations (nom de classe + nom de l'opération + type de retour + liste des paramètres). A priori dans une hiérarchie de classes, chaque classe ayant un nom différent, deux opérations de même signature sont interdites. Mais le programmeur peut :

- au sein d'une même classe donner le même nom à une opération si la liste des paramètres est différente,
- dans des classes distinctes, donner le même nom et la même liste de paramètres. Lorsqu'une méthode d'une sous-classe a même nom et même liste de paramètres qu'une méthode d'une super-classe, on dit que la méthode de la sous-classe *redéfinit* la méthode de la super-classe.

On parle alors de *surcharge*. Lorsque le long d'un chemin de spécialisation-généralisation on rencontre des opérations de même nom et même liste de paramètres on parle de *masquage*. Donc lorsque l'on code une sous-classe, on a la possibilité de masquer la définition d'une (ou de plusieurs) méthode(s) de la super-classe, simplement en redéfinissant une méthode qui a le même nom et les mêmes paramètres que celle de la super-classe.

Remarque : La méthode de la super-classe sera encore accessible mais en la préfixant par le mot-clé **super**, pseudo-variable (car définie par le système).

```
// Classe Etudiant
.....
public class Etudiant extends Personne{
.....
public boolean admis()
    {return (moyenne() >= 10;)}
}

// Classe Etud_Apprenti

public class Etud_Apprenti extends Etudiant {

.....
public boolean admis()
{
    return (super.admis() && getnoteSt() > 8);
    // on fait appel ici à l'opération admis() définie dans Etudiant
    // on aurait pu aussi définir admis dans Etud_Apprenti par
    // return (moyenne() >= 10 && getnoteSt() > 8;)
}
}
```

Remarque : La forme de l'opération `admis` qui utilise `super` est plus stable et plus réutilisable, en effet si les conditions d'admission générale sur tous les étudiants changent, le code de l'opération `admis` dans la classe `Etudiant` sera modifié, entraînant la modification automatique de toutes les opérations qui font appel à elle via `super.admis()`.

4.4 Les constructeurs

Dès qu'un constructeur a été défini dans une classe de la hiérarchie de classes, il est nécessaire de définir des constructeurs lui correspondant dans toutes les sous-classes. D'autre part, il est commode et intéressant de faire appel aux constructeurs des super-classes que l'on invoque par le mot clé `super` suivi de ses arguments entre parenthèses. L'invocation du constructeur de la super-classe doit se faire obligatoirement à la première ligne.

```
// la classe Personne

public Class Personne{
.....
// constructeur par défaut
public Personne() {
    nom = "";
    prenom = "";
    age = 0;
    sexe = 'F';}
// autre constructeur
public Personne(String n, String p, int a, char s) {
    nom = n;
    prenom = p;
```

```
age = a;
sexe = s;}
// la classe Etudiant
public Class Etudiant extends Personne{
.....
// constructeur par défaut
public Etudiant()
{
    // super() sera automatiquement réalisé
    numEtu=-1;
}

// autre constructeur
public Etudiant(String n, String p, int a, char s, double n)
{
    super(n, p, a, s); // appel au 2ième constructeur
                        // de la super-classe Personne;
    numEtu=n;
}

// la classe Etu_Apprenti
public Class Etu_Apprenti extends Etudiant{
// constructeur par défaut
public Etu_Apprenti ()
{
    // Par défaut, super() est appelée.
    E1=-1; E2=-1; stage=-1;
}

// autre constructeur
public Etu_Apprenti (String n, String p, int a, char s, double n, int e1, int e2, int st)
{
    super(n, p, a, s, n); // appel au 2ième constructeur de Etudiant
    E1 = e1; E2 = e2; stage = st;
}
```

Remarque : L'utilisation de **super** dans les constructeurs permet de mieux gérer l'évolution du code, une modification d'un constructeur d'une super-classe provoquant automatiquement la modification des constructeurs des sous-classes faisant appel à lui par **super**.

4.5 Protections

Nous avons déjà introduit les directives de protection, nous pouvons donc compléter ici leur description (quoique le mécanisme d'accès reste encore bien obscur) :

- **public** : la méthode (ou l'attribut) est accessible par tous et de n'importe où.
- **protected** : la méthode (ou l'attribut) n'est accessible que par les classes du même package et par les sous-classes (même si elles se trouvent dans un autre package).

Remarque : pour le contrôle d'accès **protected**, il semblait logique de n'autoriser l'accès qu'à la classe concernée et à ses sous classes (comme en C++).

- **private** : la méthode (ou l'attribut) est accessible uniquement par les méthodes de la classe. *Remarque* : Cependant les instances d'une même classe ont accès aux méthodes et attributs privés des autres instances de cette classe.

Lorsque rien n'est précisé, l'accès est autorisé depuis toutes les classes du même package.

4.6 Classes et méthodes abstraites

Dans une hiérarchie de classes, plus une classe occupe un rang élevé, plus elle est générale donc plus elle est abstraite. On peut donc envisager de l'abstraire complètement en lui ôtant d'une part le rôle de génitrice (elle ne sera pas autorisée à créer des instances) et en lui permettant d'autre part de factoriser structures et comportements (sans savoir exactement comment les faire) uniquement pour rendre service à sa sous-hiérarchie.

Une méthode *abstraite* est une méthode déclarée avec le mot clef **abstract** et ne possède pas de corps (pas de définition de code). Par exemple, le calcul de la moyenne d'un étudiant ne peut pas être décrit au niveau de la classe **Etudiant**.

Néanmoins, on sait que les étudiants de licence et les étudiants apprentis doivent être capable de calculer leur moyenne. La méthode **moyenne** doit être déclarée abstraite au niveau de la classe **Etudiant**. Par contre, il faudra obligatoirement définir le corps de la méthode **moyenne** dans les sous-classes de la classe **Etudiant**.

Si une classe contient au moins une méthode abstraite, alors il faut déclarer cette classe abstraite, mais on peut aussi définir des classes abstraites (parce qu'on ne veut pas qu'elles engendrent des objets) sans aucune méthode abstraite.

```
.....
// la classe est déclarée abstraite
public abstract class Etudiant extends Personne{

.....
// la méthode est déclarée abstraite
public abstract double moyenne();
}
```

Remarques : Dans le formalisme UML les classes et méthodes abstraites ont leur nom en italique.

Une classe abstraite peut être sous-classe d'une classe concrète (ici **Etudiant** est une sous-classe abstraite de **Personne** qui est concrète).

L'intérêt de définir une méthode abstraite est double : il permet au développeur de ne pas oublier de redéfinir une méthode qui a été définie **abstract** au niveau d'une des super-classes ; il permet de mettre en œuvre le polymorphisme.

4.7 Le polymorphisme

4.7.1 Transformation de type ou Casting

Une référence sur un objet d'une sous-classe peut toujours être implicitement convertie en une référence sur un objet de la super-classe.

```
Personne p;
Etud_Licence e=new Etud_Licence();
p=e;
```

```
// tout étudiant de licence est un étudiant et a fortiori une personne
```

L'opération inverse (cast-down) est possible de manière explicite (mais avec précaution et uniquement en cas de nécessité absolue).

```
Etud_Licence e;
Personne p=new Etud_Licence();
e = (Etud_Licence) p ;
// p peut désigner des personnes (type statique ou type de la variable)
// p désigne ici en realite un etudiant de licence
// (type dynamique ou type défini par le new)
```

Cette opération de transformation de type explicite peut être utilisée indépendamment des hiérarchies de classes sur les types primitifs.

```
double x = 9.9743;
int xEntier = (int) x; // alors xEntier=9
```

Pour vérifier qu'une instance appartient bien à une classe précise d'une hiérarchie, on peut utiliser l'opérateur `instanceof`. Par exemple, si on veut tester si l'instance `e` a pour classe ou pour superclasse la classe `Etud_Apprenti`, on écrit :

```
if (e instanceof Etud_Apprenti) .....
```

4.7.2 Polymorphisme des opérations

Le fait que l'on puisse définir dans plusieurs classes des méthodes de même nom revient donc à désigner plusieurs formes de traitement derrière ces méthodes. Le code de la méthode qui sera réellement exécuté n'est donc pas figé, un appel de message autorisé à la compilation donnera des résultats différents à l'exécution car le langage retrouvera selon l'objet et la classe à laquelle il doit son existence le code à exécuter (on parle de *liaison dynamique*). La capacité pour un message de correspondre à plusieurs formes de traitements est appelé **polymorphisme**

Quelques exemples pour fixer les idées.

```
//une autre hiérarchie de classes
```

```
public abstract class Felin {
.....
    public abstract String pousseSonCri();
.....
}

public class Lion extends Felin {
.....
    public String pousseSonCri() {return "rouaaaaah";}
    public String toString() {return "lion";}
.....
}
```

```
public class Chat extends Felin {
    .....
    public String pousseSonCri() {return "miaou";}
    public String toString() {return "chat";}
    .....
}

public class AppliCriDeLaBete {
    public static void main(String args[]) {
        Felin tab[] = new Felin[2];
        tab[0] = new Lion();
        tab[1] = new Chat();
        for( int i=0; i<2; i++)
            System.out.println("Le cri du "+tab[i]+" est : "+ tab[i].pousseSonCri());
    }
}
```

Remarquer ici, que le main utilise un tableau de Felin, que le casting implicite est utilisé, et que tab[i] étant un Felin pour le compilateur et la méthode pousseSonCri étant définie dans Felin (sous forme abstraite), il n'y a pas d'erreur de compilation et à l'exécution, on obtient :

```
Le cri du lion est : rouaaaaah
Le cri du chat est : miaou
```

Nous montrons à présent un exemple de polymorphisme dans le cas d'une classe représentant les promotions d'étudiants.

```
.....
public class Promotion {
    private ArrayList<Etudiant> listeEtudiants = new ArrayList<Etudiant>();
    .....
    public double moyenneGenerale() {
        double somme = 0;
        for (int i=0;i<listeEtudiants.size();i++)
            {Etudiant etud = listeEtudiants.get(i));
        // Le polymorphisme a lieu ici: le calcul de la méthode moyenne() sera
        // différent si etud est une instance de Etud_Licence ou de Etud_Apprenti.
            somme = somme + etud.moyenne();
        }
        if (listeEtudiants.size()>0)
            return (somme / listeEtudiants.size());
        else return 0;
    }

    public String nomDesEtudiants() {
        String listeNom = "";
```

```
// Le polymorphisme a lieu ici: la méthode toString() est appelée sur tous les
// elements de la ArrayList listeEtudiants.

        for (int i=0;i<listeEtudiants.size();i++)
            listeNom = listeNom +listeEtudiants.get(i).toString() + " \n";

        return listeNom;
    }
} // fin classe Promotion

public class AppliPromo {
    public static void main(String args[]) {
        Promotion promo = new Promotion (2000);
        Etud_Licence e1 = new Etud_Licence("Cesar", "Julio", 26, 'M', 127, 14, 12, 10);
        promo.inscrit (e1);
        Etud_Licence e2 = new Etud_Licence("Curie", "Marie", 22, 'F', 124, 8, 17, 20);
        promo.inscrit (e2);
        Etud_Apprenti a1 = new Etud_Apprenti ("Bol", "Gemoï", 22, 'M', 624, 8, 8, 10);
        promo.inscrit (a1);
        Etud_apprenti a2 = new Etud_Apprenti ("Einstein", "Albert", 22, 'M', 123, 13, 17, 17);
        promo.inscrit (a2);

        System.out.println("La moyenne de la promotion est : " + promo.moyenneGenerale() );
        promo.incrementeNoteE1DesApprentis();
        System.out.println("La nouvelle moyenne de la promotion est : "
                            + promo.moyenneGenerale() );
        System.out.println("Les étudiants de la promotion sont : "
                            + promo.nomDesEtudiants() );
    }
}
```

A l'exécution, on obtient, en supposant que les méthodes `toString` des classes représentant les étudiants insèrent le type des étudiants (licence ou apprenti) :

```
La moyenne de la promotion est : 13
La nouvelle moyenne de la promotion est : 13.5
Les étudiants de la promotion sont :
Cesar Julio (Etud_Licence)
Curie Marie (Etud_Licence)
Bol Gemoï (Etud_Apprenti)
Einstein Albert (Etud_Apprenti)
```

En général, il est préférable de ne pas avoir à recourir à la transformation de type explicite. Dans notre exemple `Etudiant`, il n'est pas nécessaire de caster les étudiants d'une liste d'étudiants en `Etud_Apprenti` ou `Etud_Licence` pour calculer leur `moyenne()`.

Pour faire court, le polymorphisme fonctionne grâce à la liaison dynamique qui cherche à l'exécution la méthode la plus spécifique pour résoudre un envoi de message. Cette méthode plus spécifique se trouve soit dans la classe de l'instance (classe déterminée par le type

dynamique suivant le `new`), soit dans une super-classe, en remontant à partir de la classe de l'instance et en utilisant la première méthode trouvée répondant au message.

Chapitre 5

Associations et collections

En UML, une classe représente un ensemble d'objets (instances). Une *association* représente un ensemble de liens entre des objets (instances) de classes spécifiées. Nous illustrerons tout d'abord uniquement le cas des associations binaires qui établissent des liens entre des objets de deux classes spécifiées, car ce sont les plus généralement utilisées.

5.1 Associations et liens

Une association définit une relation entre 2 ou plusieurs classes, cette relation décrivant des connexions structurelles entre leurs instances. De fait, une association représente un ensemble de liens entre des objets des classes impliquées dans l'association. Par exemple, si on suppose disposer des classes *Ville* et *Pays*, on peut définir l'association *a pour capitale* (ou dans l'autre sens : *est la capitale de*) (voir figure 5.1).

Cette association est ici dite binaire car elle fait intervenir 2 classes : *Pays* et *Ville*. Elle définit des couples d'instances pays/ville, ces couples sont appelés des liens. Par exemple on voit à la figure 5.1 que par cette association, il existe un lien existant entre le pays de nom France et la ville de nom Paris.

Graphiquement, une association binaire se matérialise par un trait entre les deux classes impliquées, en y faisant apparaître tout ou partie des informations suivantes :

- le nom de l'association,
- le nom des rôles aux extrémités de l'association,
- la multiplicité des extrémités,
- la navigabilité.

Le nom de l'association est souvent augmenté d'une pointe de flèche ou d'un triangle qui précise dans quel sens l'association doit être lue (dans la Figure 5.1, on lit : "France a

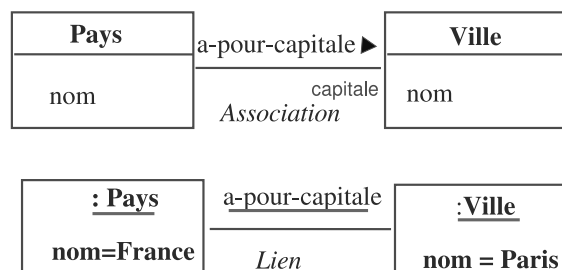


FIGURE 5.1 – Associations et liens



FIGURE 5.2 – Association binaire

pour capitale Paris”, et non pas “Paris a pour capitale France”.

Les noms des rôles sont placés aux extrémités de l’association. Un rôle placé à l’extrémité A d’une association entre A et B désigne le rôle que jouent les instances de la classe A dans l’association. Paris joue le rôle de capitale dans l’association *a_pour_capitale* entre Pays et Ville. Pour la figure 5.1, une personne joue le rôle de passager, le bus joue le rôle de véhicule.

La multiplicité (ou cardinalité) dénote le nombre d’instances impliquées de part et d’autre. Un bus transporte plusieurs passagers et un passager est transporté par un seul bus (à un instant *t*). La multiplicité peut être *i..j* (de *i* à *j*), *** (plusieurs), *1..** (au moins un), *i* (exactement *i*).

La navigabilité indique s’il est possible de traverser une association. On représente graphiquement la navigabilité par une flèche du côté de la terminaison navigable et on empêche la navigabilité par une croix du côté de la terminaison non navigable (mais ce n’est pas obligatoire). Dans la figure 5.2, le bus peut traverser l’association pour avoir connaissance de ses passagers, mais les passagers n’ont pas connaissance du bus qui les transporte. On n’a pas forcément le sens de lecture du nom de l’association dans le même sens que la navigabilité de l’association. Une association peut être navigable dans les 2 sens.

On peut noter qu’il peut exister plusieurs associations binaires différentes entre deux mêmes classes. Par exemple on peut imaginer l’association *conduit* entre la classe Bus et la classe Personne. Cette fois-ci, la personne impliquée dans l’association joue le rôle de conducteur, le bus joue le rôle de véhicule conduit, et les cardinalités sont différentes.

Autre exemple

La figure 5.3 présente deux associations entre les classes **Personne** et **Maison** (dans un diagramme de classes) et des liens correspondants (dans un diagramme d’instances).

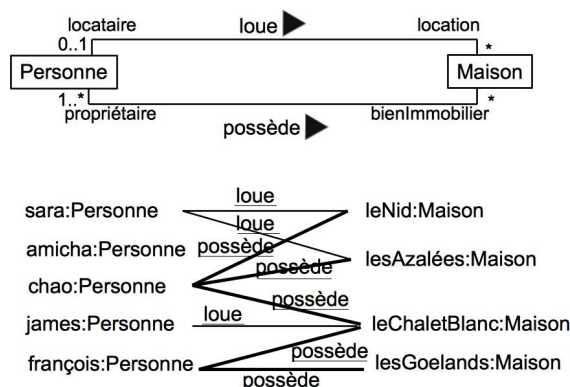


FIGURE 5.3 – Associations entre des personnes et des maisons, navigables dans les deux sens

A noter :

- le nom de l'association (loue ou possède) est suivi par un triangle noir qui indique comment le lire.
- les noms de rôles (locataire, location, propriétaire, bienImmobilier) indiquent le rôle que jouent les objets de l'extrémité d'association dans l'association. Par exemple, dans l'association **loue**, une personne joue le rôle de **locataire** alors que dans l'association **possède**, une personne joue le rôle de **propriétaire**.
- les multiplicités indiquent que :
 - une personne peut louer aucune ou plusieurs maisons
 - une maison peut être louée par au plus une personne (mais peut-être par aucune)
 - une personne peut posséder aucune ou plusieurs maisons
 - une maison a un ou plusieurs propriétaires

Interprétation :

- *Personne* et *Maison* sont des ensembles d'objets
- $\text{loue} \subseteq \text{Personne} \times \text{Maison}$
 - $\text{loue} = \{(sara, \text{leNid}), (sara, \text{lesAzales}), (james, \text{leChaletBlanc})\}$
- $\text{possede} \subseteq \text{Personne} \times \text{Maison}$
 - $\text{possede} = \{(chao, \text{leNid}), (chao, \text{lesAzales}), (chao, \text{leChaletBlanc}), (francois, \text{leChaletBlanc}), (francois, \text{lesGoeland})\}$

5.2 Associations et attributs

Il y a évidemment un lien entre associations et attributs. On pourrait se demander pourquoi introduire la notion d'association, puisqu'on pourrait toujours utiliser celle d'attribut. Il y a plusieurs raisons à cela :

- tout d'abord, pour une question de lisibilité des diagrammes : on voit bien mieux les liens entre classe avec une association qu'avec des attributs ;
- ensuite, avec une association, on définit 2 liens dépendants : les 2 bouts de l'association. Deux attributs de part et d'autre ne sont pas équivalents à une association ;
- on peut définir des associations complexes, comme on le verra par la suite.

On utilisera la convention suivante : les attributs sont uniquement de type simple (entiers, flottants, booléens, ...). Conséquence logique : on ne mettra jamais un attribut de type complexe, on préférera alors une association.

5.3 Associations particulières

Association réflexive

La figure 5.4 présente une association réflexive entre des personnes et leur mère.

Interprétation :

- *Personne* est un ensemble d'objets
- $a\text{PourMere} \subseteq \text{Personne} \times \text{Personne}$

Agrégation et composition Un *losange* placé sur une extrémité d'association indique une relation de type *ensemble-élément*, *contient*, *se compose de*. Ce symbole permet d'éviter de nommer l'association explicitement ce qui allège le diagramme de classes. On peut

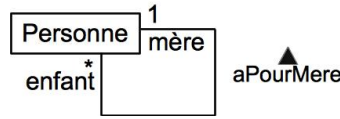


FIGURE 5.4 – Associations entre des personnes et leur mère

remettre le losange dans les diagrammes d'instances (contrairement aux multiplicités qui n'y sont pas sauf dans un cas très particulier que nous ne développerons pas ici).

On associe souvent une sémantique¹ différente au losange suivant s'il est :

- d'intérieur blanc : c'est alors une association d'*agrégation*, un élément peut être partagé par plusieurs ensembles. Sur la figure 5.5, une promotion se compose d'étudiants, qui peuvent appartenir à plusieurs promotions (par une double inscription par exemple) et dont l'existence n'est pas liée à la promotion.
- d'intérieur noir : c'est alors une association de *composition*, un élément ne peut pas être partagé par plusieurs ensembles (la multiplicité contre le losange est 1 ou 0..1). Très souvent, lorsque l'on détruit l'ensemble, alors on détruit aussi ses éléments. Sur la figure 5.5, une maison se compose de pièces, non partagées entre deux maisons et dont l'existence est liée à la maison.

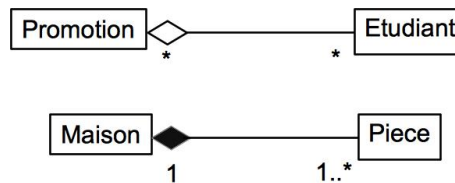


FIGURE 5.5 – Associations entre des personnes et leur mère

Association dérivée

Une association n-aire est une association qui est dérivée à partir d'autres associations, au sens où ses liens peuvent être calculés (construits) à partir des liens de ces autres associations. Dans l'exemple de la figure 5.5, les deux liens de l'association **aPourParent** pour une personne donnée sont déduits de l'existence d'un lien pour l'association **aPourPere** et d'un lien pour l'association **aPourMere**.

Association n-aire

Les associations n-aires connectent plusieurs classes. Le symbole losange évidé est à nouveau utilisé, mais ici il ne se lit pas comme une agrégation.

Dans l'exemple de la figure 5.7, une projection relie un film, un cinéma, une salle et un horaire :

- $projection \subseteq Film \times Cinema \times Salle \times Horaire$
- On place la multiplicité d'une extrémité en fixant un objet de chacune des autres extrémités
- pour un film, une salle et un horaire, il y a un seul cinéma,

1. cette sémantique peut varier, nous donnons celle qui est la plus généralement utilisée

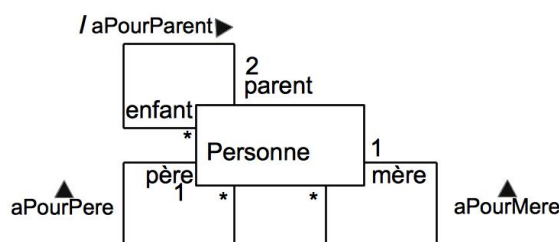
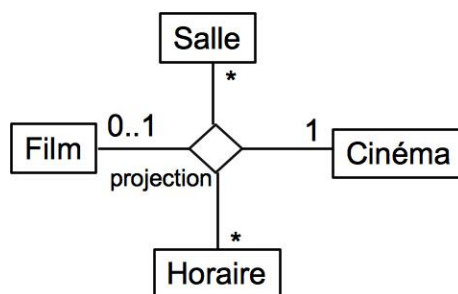


FIGURE 5.6 – Association dérivée

- pour un film, une salle et un cinéma, il y a aucun ou plusieurs horaires,
- pour un film, un horaire et un cinéma, il peut y avoir aucune ou plusieurs salles (si on admet que le film peut être projeté en parallèle,
- pour un cinéma, un horaire et une salle, il y a au plus un film projeté.


FIGURE 5.7 – Association n-aire *projection*

Association qualifiée

Un extrémité d'association peut être *qualifiée*. Le *qualifieur* d'une extrémité contre une classe C est un attribut ou un ensemble d'attributs qui partitionnent les instances associées à une instance de C . La multiplicité opposée au qualifieur est déterminée par le nombre d'objets associés à une paire composée d'un objet de la classe C et d'une valeur de l'attribut (ou des attributs s'il y en a plusieurs dans le qualifieur). L'attribut appartient à l'association.

Dans l'exemple de la figure 5.8, l'association présentée précédemment entre une promotion et des étudiants est transformée en qualifiant l'extrémité de deux manières différentes, ce qui réduit les multiplicités.

- En haut, l'extrémité contre la promotion est qualifiée par un *numeroGroupe*. A une promotion et un numéro de groupe, on peut associer entre 0 et 41 étudiants.
- En bas, l'extrémité contre la promotion est qualifiée par un *numeroEtudiantPromo*. A une promotion et un numéro d'étudiant (numéro donné à l'étudiant pour chaque promotion où il peut être), on peut associer 0 ou 1 étudiant.

Classe d'association

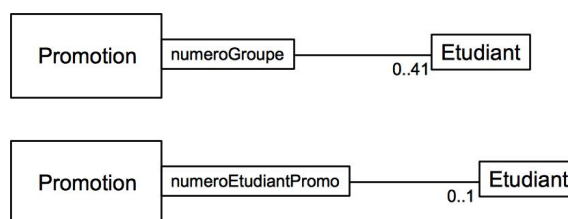


FIGURE 5.8 – Associations avec qualifieur

On peut encore plus précisément décrire les liens entre les objets par des attributs regroupés dans une *Classe d'association*. L'exemple de la figure 5.9 montre comment on peut décrire les liens entre un étudiant et sa promotion par un ensemble d'informations : son numéro de groupe (qui n'est propre ni à l'étudiant, ni à sa promotion, mais est propre à leur relation), la date d'inscription dans cette promotion, son numéro d'étudiant au sein de la promotion, etc. La classe d'association **Inscription** est attachée au trait reliant les deux classes par un trait en pointillé. Elle contient les attributs qui décrivent les liens.

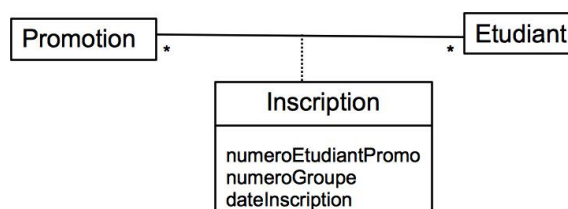


FIGURE 5.9 – Classe d'association

5.4 Traduction des associations en Java

Il n'y a pas de notion d'association en Java. On peut les traduire avec différents concepts du langage Java. Le choix est un travail qui est réalisé au moment de l'étape de conception. Vous le verrez au fur et à mesure des exercices que nous effectuerons.

Traduction par des attributs

- on associe un attribut à chaque extrémité navigable
- les noms des rôles servent à nommer les attributs (avec des variations si besoin)
- la cardinalité (multiplicité) donne des informations sur le type de l'attribut
 - multiplicité ≤ 1 type simple
 - multiplicité > 1 type collection (tableau, séquence, dictionnaire associatif, etc.)
 - s'il n'y a pas d'information supplémentaire, on peut utiliser une séquence (comme un tableau ou une `ArrayList`),
 - s'il y a un qualifieur avec un seul attribut (comme le numéro de groupe), on peut utiliser un dictionnaire associatif (comme une `HashMap` ou une `Hashtable`) où cet attribut sert de clef.
- pour une association bi-directionnelle, on veille à garder en cohérence les attributs des deux côtés
- pour une composition : on veille à respecter le fait qu'un composant ne peut pas être détenu par deux composites

Traduction par une classe

On l'utilisera principalement pour une :

- Association binaire dont on pense que sa réification est importante pour la gestion.
- Association n-aire car il est assez difficile de maintenir en cohérence plus de deux attributs.
- Association qualifiée car les attributs du qualifieur ne sont généralement pas dans les classes associées, sauf cas particulier (voir l'exemple en fin de cours).
- Classe d'association.

La classe est connectée aux classes extrémités par des attributs (un par extrémité).

5.5 Tableaux et collections

Nous avons vu que pour traduire en Java une extrémité navigable de cardinalité >1 , on fait appel à des attributs Java typés par des types capables de représenter plusieurs objets de même type. Ces types sont d'une part les tableaux comme vous les avez déjà rencontrés dans d'autres enseignements, ainsi que les collections Java (dont les Hashmap vues précédemment font partie) que nous allons rapidement présenter ici.

5.5.1 Les tableaux

En Java on sépare la déclaration d'une variable de type tableau, la construction effective d'un tableau et l'initialisation du tableau. La déclaration d'une variable de type tableau de nom `tab` dont les éléments sont de type `typ`, s'effectue par :

```
typ[] tab;
```

Lorsque l'on a déclaré un tableau en Java, il faut le créer avant de pouvoir le remplir. L'opération de construction s'effectue en utilisant un `new`, ce qui donne :

```
tab = new typ[taille];
```

Dans cette instruction, `taille` est une constante entière ou une variable de type entier dont l'évaluation doit pouvoir être effectuée à l'exécution. Une fois qu'un tableau est créé avec une certaine taille, celle-ci ne peut plus être modifiée. Cependant, la variable `tab` peut héberger par la suite un tableau de taille différente. Exemple :

```
int[] tab = new int[10];
for(int i = 0; i < 10; i++)
    tab[i] = i;
// autre solution
int[] tab2={0,1,2,3,4,5,6,7,8,9};
```

tableaux à plusieurs dimensions

```
typ[] [] tab;
tab = new typ[N][M]; // N lignes, M colonnes
int[] [] tab = {{1,2,3},{4,5,6},{7,8,9}};
```

5.5.2 Les collections Java

En Java, il existe beaucoup de types complexes définis dans la bibliothèque Java permettant de représenter des collections : ensembles, listes, ... Ces types sont dits “génériques” : ils définissent un ensemble d’éléments de type E (où E est de type non primitif), par exemple : une pile pouvant contenir des éléments de type E. Au moment de déclarer une collection, on précise si on veut une collection d’entiers (en utilisant le type `Integer` et non `int`), de voitures, de personnes, etc. On remplace le type générique E par le type souhaité. Par exemple, on peut définir une pile d’assiettes, ou une pile de feuilles, à partir de la classe `Pile` de E (notée `Pile<E>`). Il existe un certain nombre d’opérations définies pour toutes les collections : l’ajout et la suppression d’éléments, l’obtention de la taille de la collection, etc.

5.5.3 Les Vecteurs

Les vecteurs (`Vector`) sont des collections génériques qui ressemblent aux tableaux, mais dont la taille n’est pas prédéfinie. Quand on déclare un vecteur, on précise le type des objets qu’il va contenir : on déclare un vecteur de voitures, un vecteur de chaînes, un vecteur de personnes, etc. De la même façon, quand on crée le vecteur avec un `new`, le nom du type d’objets doit être précisé. La syntaxe pour déclarer et créer un vecteur `v` d’objets de type `MonType` est la suivante :

```
Vector<MonType> v; // déclaration
v=new Vector<MonType>(); // création
```

On peut bien sûr déclarer et créer un vecteur en une seule ligne :

```
Vector<MonType> v=new Vector<MonType>();
```

Pour manipuler un vecteur, on dispose de beaucoup d’opérations définies dans la classe `Vector`. Pour un vecteur d’objets de type E, on dispose entre autres des méthodes :

- `void addElement(E obj)` ou plus court `void add(E obj)`. Ajoute l’objet `obj` à la fin du vecteur, et augmente sa taille de 1.
- `E get(int index)`. Retourne l’objet placé en position `index` dans le vecteur.
- `boolean isEmpty()`. Teste si le vecteur n’a aucun élément.
- `int size()`. Retourne la taille du vecteur.
- `E remove(int index)`. Supprime l’objet en position `index` et le retourne.
- `boolean remove(Object o)`. Supprime la première occurrence de `o` rencontrée (laisse le vecteur inchangé si aucun tel objet n’est rencontré, et retourne alors faux).

Voilà un petit exemple d’utilisation des vecteurs :

```
// Création d’un ensemble capable de ne stocker que des objets String
Vector<String> prenom = new Vector<String> ();
prenom.addElement("Thomas");
prenom.addElement("Sophie");
// La méthode add interdit d’ajouter un autre type d’objet
// prenom.addElement(new Voiture()); -> interdit

//affichage des prénoms
for (int i = 0; i < prenom.size(); i++)
{
    System.out.println(prenom.get(i));
}
```

```
}  
// autre affichage des prénoms avec la boucle for existant depuis Java 1.5  
for (String prenomCourant:prenums){  
    System.out.println(prenomCourant);  
}
```

On notera l'utilisation d'une boucle `for` particulière, qui permet de parcourir une collection sans utiliser explicitement d'indice de boucle. Cette boucle `for` peut être vue comme un `for each`. Elle a pour syntaxe :

```
1 for (LeType objetIterateur : laCollection) {  
2     //Code a executer dans la boucle  
3     //Ici, on peut faire appel a objetIterateur (par exemple :  
        objetIterateur.uneMethode())  
4     // objetIterateur reference successivement tous les elements de la  
        collection laCollection, il est de type LeType  
5 }
```

Cette boucle peut être utilisée pour les tableaux, et tous les objets itérables, en particulier toutes les collections java.

5.5.4 Vector vs ArrayList

Une collection très proche des Vector est la collection ArrayList. Les différences entre les deux sont extra-fonctionnelles :

- Les Vector sont synchronisés, les ArrayList non
- Croissance différente de la structure interne (Array) : par défaut doublement pour les Vector, augmentation de 50% pour les ArrayList

On préférera donc les ArrayList si aucune concurrence n'est nécessaire dans l'accès à la liste.

5.5.5 Les dictionnaires associatifs

On traduit en général une association qualifiée par un dictionnaire associatif. Un dictionnaire associatif est une structure indexée par une clef, c'est à dire un identifiant permettant d'accéder directement à l'élément correspondant à cet identifiant dans la structure. Par exemple, on peut envisager un dictionnaire de comptes bancaires, indexés par leur numéro de compte, que l'on sait unique. Ici, la clef est le numéro de compte, et est de type entier. Le dictionnaire contient des comptes, auxquels on accède directement grâce à la clef. On utilise souvent comme dictionnaire associatif des tables de hachage. Le terme de hachage est dû au mécanisme qui est mis en œuvre pour permettre cet accès direct et rapide à partir d'une clef, mais que nous ne verrons pas ici. Pour déclarer et créer une table de hachage contenant des objets de type `MonType` indexés par des objets de type `Clef`, on écrit :

```
Hashtable<Clef, MonType> table=new Hashtable<Clef, MonType>();
```

Pour une table d'objets de type `V` et de clefs de type `K`, on a par exemple les méthodes suivantes :

- `V get(Object key)`. Retourne l'objet de clef `key` ou `null` s'il n'y en a pas.
- `V put(K key, V value)`. Ajoute l'élément `value` avec comme clef `key`. S'il existait déjà un élément de même clef, cet élément est retourné (et écrasé dans la table par `value`).

— `V remove(Object key)`. Retire l'élément de clef `key` de la table.

Voilà un petit exemple d'utilisation des tables de hachage :

```
Hashtable<String, Integer> numbers = new Hashtable<String, Integer>();
numbers.put("one", new Integer(1));
numbers.put("two", new Integer(2));
numbers.put("three", new Integer(3));

Integer n = numbers.get("two");
if (n != null) {
    System.out.println("two = " + n);
}
```

5.5.6 Exemple d'utilisation d'un dictionnaire associatif

Un dictionnaire associatif peut être réalisé en Java avec plusieurs classes dont les ensembles d'opérations sont similaires, ici nous présentons dans cet exemple la classe `HashMap<K,V>`. `K` est le type des clefs et `V` le type des valeurs.

Lorsque l'on déclare un dictionnaire associatif, deux indications de type doivent être données : le type des clefs et le type des valeurs.

Par exemple, pour mettre en place un dictionnaire associatif d'étudiants indexés par leur numéro qui est une chaîne de caractères, cela donnera :

```
HashMap<String, Etudiant> listeEtu = new HashMap<>();
```

Les opérations les plus importantes sont les suivantes :

```
// mettre des paires dans la map
V put(K key, V value)
// récupérer une valeur associée à une clef
V get(Object key)
// connaître le contenu
boolean containsKey(Object key)
boolean containsValue(Object value)
Set<Map.Entry<K,V>> entrySet()
Set<K> keySet()
Collection<V> values()
// autres opérations usuelles
boolean isEmpty()
int size()
```

Nous présentons maintenant un squelette d'implémentation de l'association qualifiée par le numéro d'étudiant entre `Promotion` et `Etudiant` (bas de la figure 5.8). Nous simplifions le problème en supposant qu'un étudiant n'appartient qu'à une promotion, donc le numéro dans cette promotion peut être un attribut de l'étudiant.

Notez les deux attributs qui se correspondent (car les deux extrémités sont navigables) : la promotion connaît ses étudiants par l'attribut `listeEtu`, tandis que l'étudiant connaît sa promotion par l'attribut de même nom. Lorsque l'on inscrit un étudiant, on met à jour l'attribut qui désigne sa promotion.

```
package promotion;
import java.util.*;
```

```
public class Promotion {
    private HashMap<String, Etudiant> listeEtu = new HashMap<>();
    private String nomPromo = "promo inconnue";

    public Promotion(String nomPromo) {this.nomPromo = nomPromo;}
    public String getNomPromo() {return nomPromo;}
    public void setNomPromo(String nomPromo) {this.nomPromo = nomPromo;}

    public void inscrire(Etudiant e)
    {listeEtu.put(e.getNumEtuPromo(), e); e.setPromotion(this);}

    public Etudiant recherche(String num)
    {return listeEtu.get(num);}

    public void affiche()
    {System.out.println(listeEtu.entrySet());}

    public String toString() {return nomPromo;}

    public static void main(String[] argv){
        Promotion etoile2016 = new Promotion("étoile 2016");
        etoile2016.inscrire(new Etudiant("jean", "0002"));
        etoile2016.inscrire(new Etudiant("samira", "0004"));
        etoile2016.affiche();
        System.out.println(etoile2016.recherche("0004"));
        System.out.println(etoile2016.recherche("0007"));
    }
}
//-----
public class Etudiant {
    private String numEtuPromo="inconnu";
    private Promotion promotion;
    private String nom="nomInconnu";
    public Etudiant(String nom, String numEtuPromo) {
        this.numEtuPromo = numEtuPromo;
        this.nom = nom;
    }
    public String getNumEtuPromo() {return numEtuPromo;}
    public void setNumEtuPromo(String numEtuPromo) {this.numEtuPromo = numEtuPromo;}
    public Promotion getPromotion() {return this.promotion;}
    public String getNom() {return nom;}
    public void setNom(String nom) {this.nom = nom;}
    public String toString() {
        return "Etudiant [numEtuPromo=" + numEtuPromo + ", Promotion="
            + this.promotion + ", nom=" + nom + "]\n";
    }
}
```

Pour finir, rappelons que lorsqu'une seule des extrémités est navigable (comme dans la figure 5.10), un seul attribut représentera l'association. Ici on supprimerait l'attribut

`promotion` de la classe `Etudiant` (et les autres points du code associés à cet attribut) et on ne conserverait que l'attribut `listeEtu` de la classe `Promotion`.

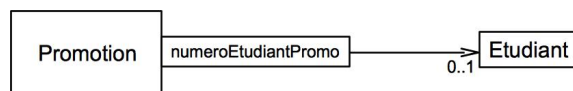


FIGURE 5.10 – Association `Promotion` vers `Etudiant` navigable dans un seul sens (`Promotion` vers `Etudiant`)

Chapitre 6

Les interfaces

6.1 Interfaces en Java

Le langage Java introduit la notion d'*interface* pour définir un *contrat* (ou cahier des charges) pour un ensemble de classes. Par contrat on entend un ensemble d'opérations, incluant des opérations d'accès à des données, que l'on s'attend à avoir dans toutes les classes respectant ce contrat.

Une interface Java regroupe plus exactement :

- Dans les versions de Java \leq Java 1.7
 - des méthodes d'instance publiques abstraites (avec les modifieurs `public abstract`)
 - des attributs de classe constant publics (avec les modifieurs `public final static`)
- A partir de Java 1.8
 - des méthodes d'instance publiques présentant des comportements par défaut (avec les modifieurs `public` et `default`)
 - des méthodes statiques
 - des types internes (non développé dans ce cours)

On remarquera qu'une interface ne contient donc jamais ni constructeur, ni le code des méthodes (en dehors des méthodes par défaut et des méthodes statiques), ni aucun attribut d'instance.

6.1.1 Définition d'une interface

Par exemple, le code ci-dessous montre une interface (très simplifiée) décrivant les quadrilatères. Un quadrilatère a 4 côtés et dispose de deux opérations permettant de connaître son périmètre et sa surface.

```
public interface Iquadrilatere {  
    public static final int nbCotes = 4;  
    public abstract double perimetre();  
    public abstract double surface();  
}
```

Les groupes de modifieurs suivants peuvent être omis :

- `public static final`
- `public abstract`
- `public`

On écrira donc le plus souvent une interface comme suit :

```
public interface Iquadrilatere {  
    int nbCotes = 4;  
    double perimetre();  
    double surface();  
}
```

6.1.2 Spécialisation d'une interface

De même que l'on peut spécialiser des classes, on peut également spécialiser des interfaces. Par exemple, l'interface `Iquadrilatere` peut être spécialisée par l'interface `Irectangle`. Le mot-clef `extends`, déjà rencontré pour indiquer qu'une classe en spécialise une autre, est utilisé ici aussi.

Dans cette interface, on précise que l'angle est toujours de 90^0 , et quatre opérations permettent de connaître ou de modifier respectivement la largeur et la hauteur.

De plus, des implémentations par défaut sont données pour les méthodes de calcul du périmètre et de la surface, ainsi qu'une description sous forme d'une chaîne de caractères. Ces méthodes peuvent être écrites à ce stade puisqu'elles appellent des méthodes dont l'existence est assurée par le contrat (les méthodes d'instance publiques prévues).

Enfin, un exemple de méthode `static` est donné. Cette méthode vérifie l'égalité de deux rectangles (donnée par celle de leurs largeurs et de leurs hauteurs respectives).

```
public interface Irectangle extends Iquadrilatere{  
    int angle = 90;  
    double getLargeur();  
    void setLargeur(double l);  
    double getHauteur();  
    void setHauteur(double h);  
  
    default double perimetre()  
    {return 2*this.getLargeur()+2*this.getHauteur();}  
  
    default double surface()  
    {return this.getLargeur()*this.getHauteur();}  
  
    default String description() // on ne peut pas utiliser le nom "toString"  
    {return "largeur =" +this.getLargeur()+" hauteur =" +this.getHauteur();}  
  
    static boolean egal(Irectangle r1, Irectangle r2)  
    {  
        return r1.getLargeur()==r2.getLargeur()  
            && r1.getHauteur()==r2.getHauteur();  
    }  
}
```

Au contraire d'une classe qui ne peut avoir qu'une super-classe directe, une interface peut avoir plusieurs super-interfaces directes. On parle de spécialisation **multiple**.

Par exemple, on peut définir une interface représentant les objets colorés puis une interface spécialisant les rectangles et les objets colorés (elle étend deux autres interfaces).

```
import java.awt.Color;
```

```
public interface IobjetCouleur {
    Color couleurDefaut = Color.white;
    Color getCouleur();
}

public interface IrectangleCouleur extends IobjetCouleur, Irectangle{
    void repeindre(Color c);
}
```

En cas d'héritage de deux méthodes par défaut dans une classe ou dans une interface, une erreur se produit, il faut alors proposer une solution (une nouvelle implémentation de cette méthode) sur le lieu de l'erreur.

6.1.3 Implémentation d'une interface

Une interface s'implémente à l'aide d'une classe. Par exemple nous proposons dans le code ci-dessous deux classes **Rectangle** qui implémentent les méthodes des interfaces qu'elles spécialisent (dont elles respectent le contrat).

RectangleAT implémente directement **Irectangle** (notez le mot-clef **implements**) et indirectement **Iquadrilatere** qui est une généralisation de **Irectangle**. Cette implémentation stocke la largeur et la hauteur dans deux attributs.

```
public class RectangleAT implements Irectangle {

    private double largeur, hauteur;

    public RectangleAT() {}

    public RectangleAT(double largeur, double hauteur) {
        this.largeur = largeur;
        this.hauteur = hauteur;
    }

    @Override
    public double getLargeur() {return this.largeur;}

    @Override
    public void setLargeur(double l) {this.largeur = l;}

    @Override
    public double getHauteur() {return this.hauteur;}

    @Override
    public void setHauteur(double h) {this.hauteur = h;}
}
```

RectangleDA implémente directement **Irectangle** (notez le mot-clef **implements**) et indirectement **Iquadrilatere** qui est une généralisation de **Irectangle**. Cette implémentation stocke la largeur et la hauteur dans un dictionnaire associatif en utilisant des paires de la forme :

- ("largeur", 4)
- ("hauteur", 8)

```
import java.util.HashMap;

public class RectangleDA implements Irectangle{

    private HashMap<String,Double> cotes = new HashMap<>();

    public RectangleDA()
    {
        cotes.put("largeur", 0.0);
        cotes.put("hauteur", 0.0);
    }

    public RectangleDA(double largeur, double hauteur)
    {
        cotes.put("largeur", largeur);
        cotes.put("hauteur", hauteur);
    }

    @Override
    public double getLargeur() {
        return cotes.get("largeur");
    }

    @Override
    public void setLargeur(double l) {
        cotes.put("largeur", l);
    }

    @Override
    public double getHauteur() {
        return cotes.get("hauteur");
    }

    @Override
    public void setHauteur(double h) {
        cotes.put("hauteur", h);
    }
}
```

Notez que :

- Une classe peut implémenter directement plusieurs interfaces.
- Si une classe n'implémente pas toutes les méthodes d'instance publiques des interfaces qu'elle implémente, c'est une classe abstraite.

Voici quelques exemples de création d'objets des classes implémentant l'interface `Irectangle`. On y voit également qu'il est impossible de créer un objet à partir d'une interface, ce qui est normal car on ne connaîtrait pas son implémentation.

```
public static void main(String[] args) {
    //Irectangle r0 = new Irectangle); // IMPOSSIBLE
    Irectangle r1 = new RectangleDA(4,5);
    Irectangle r2 = new RectangleAT(2,5);
}
```

6.1.4 Code utilisant les interfaces

Grâce à l'interface `IrectangleColore`, on est en mesure d'écrire une opération manipulant les rectangles colorés sans s'appuyer sur leurs possibles implémentations. Par exemple, dans le code ci-dessous, on décrit une classe représentant des ensembles de rectangles colorés. On peut y calculer par exemple la somme des surfaces des rectangles, utile si on veut connaître la quantité de peinture nécessaire pour repeindre les rectangles par exemple.

On commence par définir une classe implémentant les rectangles colorés.

```
public class RectangleDAcolore extends RectangleDA implements IrectangleColore {
    private Color couleur;

    public RectangleDAcolore() {
        this.couleur = IobjetColore.couleurDefaut;
    }

    public RectangleDAcolore(double largeur, double hauteur, Color couleur) {
        super(largeur, hauteur);
        this.couleur = couleur;
    }

    @Override
    public Color getCouleur() {return this.couleur;}

    @Override
    public void repeindre(Color c) {this.couleur = c;}
}
```

Puis on définit une classe accueillant un ensemble de rectangles.

```
public class StockRectangles {
    private ArrayList<IrectangleColore> listeRectangles = new ArrayList<>();

    public StockRectangles() {}

    public void ajoute(IrectangleColore rect)
    {
        if (! listeRectangles.contains(rect))
            listeRectangles.add(rect);
    }

    public double SommeSurfaces()
    {
        double somme=0;
        for (Irectangle rect : listeRectangles)
```

```
        // ou for (IrectangleColore rect : listeRectangles)
            somme+=rect.surface();
        return somme;
    }
}

....
public static void main(String[] args) {
    IrectangleColore r1 = new RectangleDAcolore(4,5,Color.pink);
    IrectangleColore r2 = new RectangleDAcolore(2,5, Color.red);
    System.out.println("égalité = "+Irectangle.egal(r1, r1));
    StockRectangles s = new StockRectangles();
    s.ajoute(r1); s.ajoute(r2);
    System.out.println("somme des surfaces = "+s.SommeSurfaces());
}
```

Pour synthétiser, les interfaces sont :

- des types plus abstraits que les classes, finalement plus réutilisables,
- une technique pour masquer l'implémentation en définissant une partie d'un type abstrait, qui favorise l'écriture de code plus général,
- des types qui améliorent la structuration par spécialisation (ou implémentation) multiple : entre interfaces, entre classes et interfaces.

6.2 Représentation en UML

La figure 6.1 présente dans la notation UML les types qui ont été construits dans le programme Java des sections précédentes. Il faut y remarquer :

- le stéréotype (mot-clef) **«interface»** qui introduit les interfaces
- les relations de spécialisation (déjà connues) entre classes ou entre interfaces (trait plein et flèche fermée)
- la relation de réalisation (d'implémentation) entre une classe et une interface qui se distingue graphiquement de la relation de spécialisation car son trait est en pointillé
- la relation d'usage, en pointillé, avec une pointe de flèche ouverte et portant l'indication **«uses»**

La figure 6.2 présente une notation alternative pour indiquer qu'une classe (ici **RectangleDA**) implémente une interface (ici **Irectangle**). La notation met en valeur le fait que la classe expose l'interface à ses classes utilisatrices (interface fournie), et qu'elle a besoin d'utiliser une classe implémentant l'interface **Map** qui offre les services des dictionnaires associatifs en Java (interface requise).

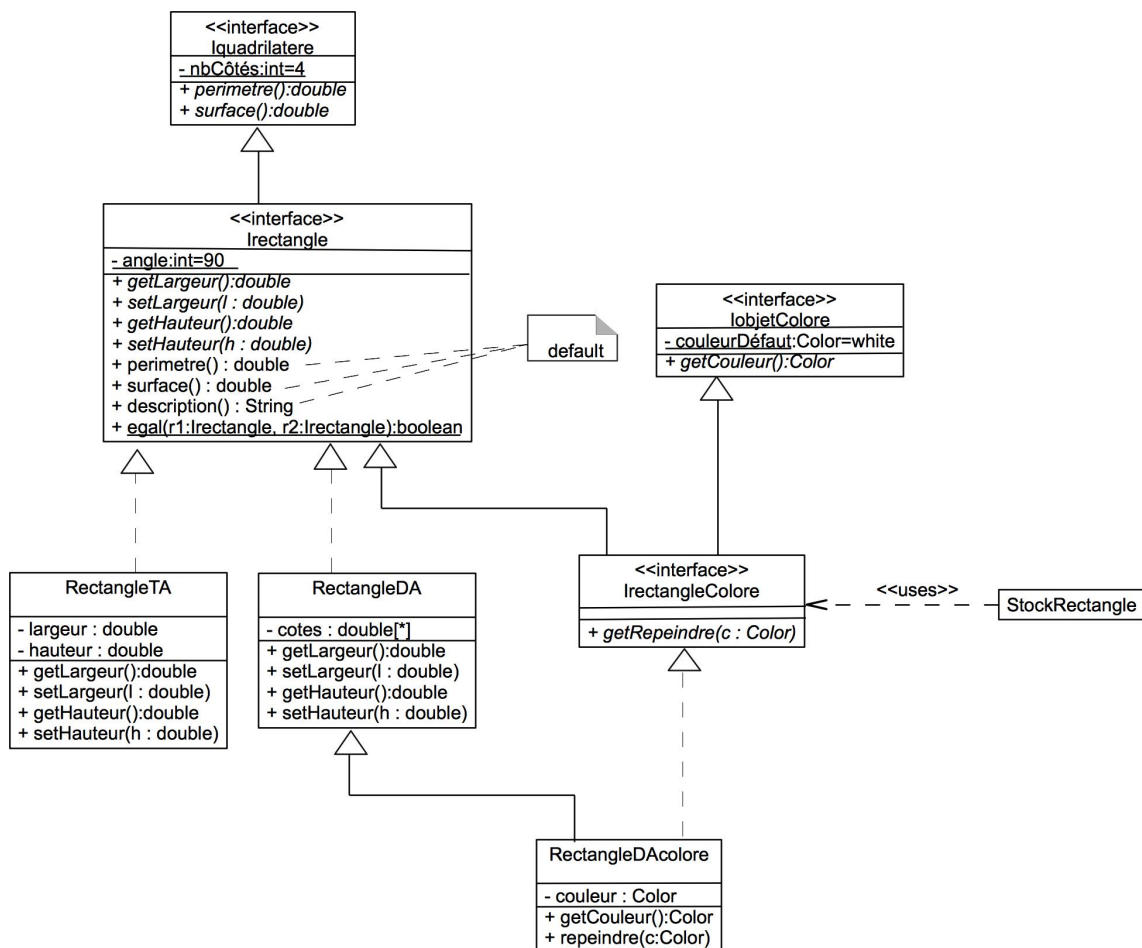


FIGURE 6.1 – Représentation des interfaces et des relations de spécialisation, de réalisation (implémentation) et d’usage

6.3 La place des interfaces dans l’API de Java

L’API de Java contient de nombreuses interfaces.

Certaines décrivent des types abstraits de données comme :

— L’interface `List`

- qui décrit les séquences d’éléments et offre des méthodes telles que :
 - insertion `add(valeur)`, récupération `get(int i)`, taille `size()`, etc.

— qui est implémentée par des classes telles que `Vector`, `ArrayList`, `LinkedList`

— L’interface `Map`

- qui décrit les dictionnaires associatifs et offre des méthodes telles que :
 - insertion `put(clef, valeur)`, récupération `get(clef)`, taille `size()`, etc.
- qui est implémentée par des classes telles que `Hashtable`, `HashMap`, `TreeMap`

— L’interface `Comparable`

- Comprenant la méthode `int compareTo(autreObjet)`
- Une séquence contenant des éléments respectant (instances d’une classe implémentant) l’interface `Comparable` pourra être triée grâce à des méthodes prédéfinies dans l’API comme `Collections.sort`.

D’autres sont vides de méthodes, on dit que ce sont des interfaces marqueurs indiquant

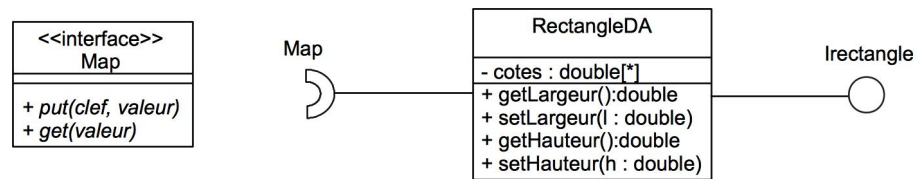


FIGURE 6.2 – La classe `RectangleDA` expose l'interface `Irectangle` et requiert l'interface `Map`

si un objet peut être soumis à un certain traitement :

- **Cloneable** : pour les objets qui seront munis d'une méthode `clone` publique par redéfinition de la méthode `protected Object clone()` de la classe `Object`.
- **Serializable** : pour les objets que l'on pourra sérialiser, c'est-à-dire placer dans des flux d'objets (et donc concrètement dans des fichiers avec un format particulier).

Chapitre 7

Le modèle d'utilisation en UML

Le modèle d'utilisation délimite le système et décrit la manière de l'utiliser ; il oriente le développement entier du système. On l'appelle aussi le modèle fonctionnel.

Plus précisément il montre :

- Les fonctionnalités externes,
- point de vue des utilisateurs,
- les interactions avec les acteurs extérieurs.

Les concepts les plus importants sont :

- la frontière qui délimite le système,
- les acteurs
 - par acteur on entend toute entité extérieure au système et interagissant avec celui-ci.
 - On trouve des acteurs humains et des acteurs machine (système extérieur communiquant avec le système étudié)
- Cas d'utilisation
 - toute manière d'utiliser le système

La figure 7.1 montre la notation graphique utilisée pour les acteurs (petits personnages ou boîtes) et les cas d'utilisation (ovales). Entre acteurs et cas d'utilisation on trace des traits représentant leurs associations.

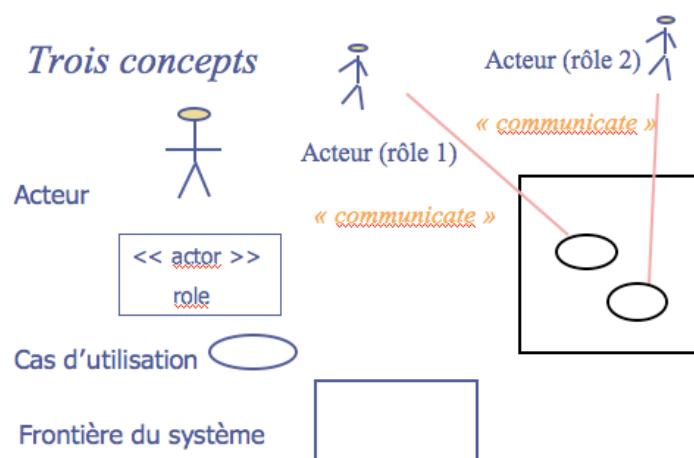


FIGURE 7.1 – Éléments des diagrammes de cas d'utilisation

La figure 7.2 donne un exemple concret, où un client peut effectuer des commandes, tandis qu'un gestionnaire de stock effectue des livraisons. Notez le rectangle qui délimite

le système : les cas d'utilisation sont à l'intérieur et les acteurs à l'extérieur. Elle montre également comment les cas d'utilisation peuvent se compléter les uns les autres.

Les cas d'utilisation peuvent être liés par des relations :

- d'utilisation **include** (le cas origine contient obligatoirement l'autre)
- de raffinement **extend** (le cas origine peut être ajouté optionnellement)
- de spécialisation (le cas origine est une forme particulière de l'autre)

Notez sur les figures la représentation graphique de ces relations.

Par exemple, le cas (la fonctionnalité) **commander** inclut (obligatoirement) le cas **décrire les produits** (Figure 7.2). le cas **Paieement CB** est une spécialisation (un cas particulier) du cas **procéder au paiement**.

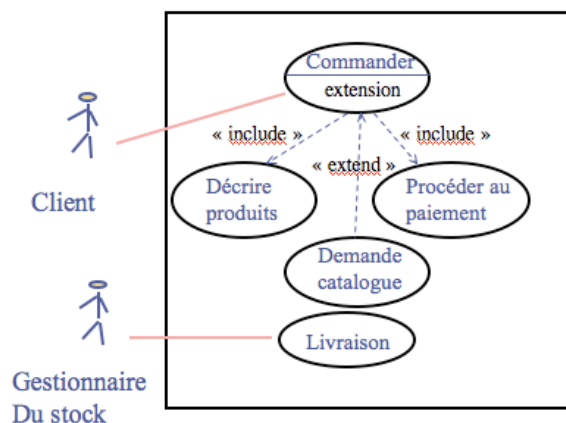


FIGURE 7.2 – Relations include et extend

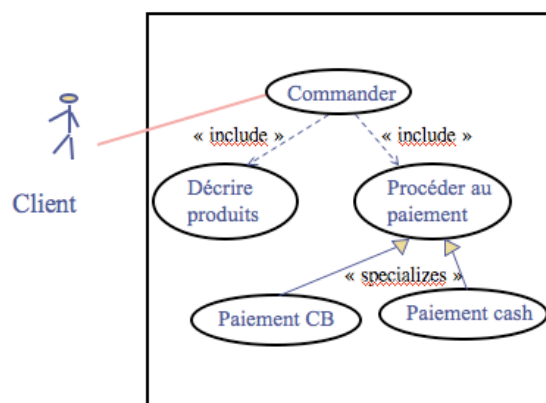


FIGURE 7.3 – Relations de spécialisation entre cas d'utilisation

Les diagrammes de cas d'utilisation s'accompagnent le plus souvent de descriptions complémentaires, textes, diagramme de séquences ou d'activités, notamment pour décrire les aspects chronologiques dans l'utilisation du système.

Une proposition courante de description complémentaire comprend :

- Sommaire d'identification
- Titre, résumé, acteurs, dates création maj, version, auteurs

- Description des enchaînements
Pré-conditions, scénario nominal, alternatives, exceptions, post-conditions
- Besoins IHM
- Contraintes « non fonctionnelles »
Temps de réponse, concurrence, ressources machine, etc.

Chapitre 8

Objets et récursivité

8.1 Introduction à la récursivité

Principe général

De manière générale, une expression récursive est une expression qui se fait référence à elle-même. En programmation, un algorithme récursif est un algorithme qui s'appelle lui-même.

Cela s'apparente :

- aux courbes fractales, aux mises en abîme dans le domaine de l'art ou du graphisme (le dessin de la boîte de la vache qui rit dont les boucles d'oreille sont des dessins de boîtes de vache qui rit).
- à certains motifs biologiques (fleur de tournesol, coquilles de nautilus)
- aux définitions de concepts telles que : "un troupeau de moutons est vide ou bien un mouton ajouté à un troupeau de moutons".
- aux définitions de fonctions mathématiques dans leurs propres termes, ou de suites mathématiques définies par récurrence.

Voici la fonction factorielle définie dans ses propres termes :

$$\begin{aligned} 0! &= 1 \\ \text{pour } n > 0, \quad n! &= n * (n - 1)! \end{aligned} \tag{8.1}$$

Et voici une suite mathématique (u_n) définie par récurrence :

$$u_0 = 1 \text{ pour } n > 0, \quad u_{n+1} = \sqrt{1 + u_n} \tag{8.2}$$

Le principe de réduction du problème La pensée récursive est en partie fondée sur un principe de réduction du problème vu sur une donnée d'une certaine taille à ce même problème vu sur une donnée de taille inférieure.

Par exemple, si on examine la définition non récursive de la fonction factorielle :

$$\begin{aligned} 0! &= 1 \\ n! &\text{ est le produit des entiers de } 1 \text{ à } n, \text{ pour } n > 0 \end{aligned} \tag{8.3}$$

Pour passer à la définition récursive on cherche à mettre en évidence :

Le cas de base

$$0! = 1 \tag{8.4}$$

Le cas général : comment $n!$ peut se penser en terme d'une factorielle d'un nombre plus petit

$$\begin{aligned} \text{pour } n > 0, \quad n! &= n * (n - 1) * (n - 2) * \dots * 1 \\ \text{pour } n > 0, \quad n! &= n * (n - 1)! \end{aligned} \tag{8.5}$$

8.2 La récursivité en modélisation et en programmation

Dans notre domaine, la récursivité va permettre de formuler de manière élégante certains algorithmes et certaines structures de données dans des cas où leur définition/expression est naturellement récursive :

- Fonction factorielle
- Liste : une liste est un élément suivi d'une sous-liste
- Arbre binaire : un arbre binaire est constitué d'un nœud/élément et de deux sous-arbres

En UML, on la trouvera fréquemment :

- sous forme de classes munies d'associations (ou d'attributs) ayant comme extrémité (comme type) cette même classe.

En Java, on la trouvera :

- pour définir des structures de données
- dans des méthodes statiques
- dans des méthodes d'instances
 - dans un cadre général
 - sur des structures de données récursives

Dans ce cours, nous n'aborderons pas différents aspects de la récursivité comme la récursivité terminale versus non terminale ou les techniques de dé-récursivation.

8.2.1 Méthodes statiques récursives

On peut écrire de manière récursive la fonction factorielle sous la forme suivante.

```
public class FonctionsRécursives{

    public static int factorielle(int n)
    {
        if (n==0) return 1;
        else return (n * factorielle(n-1));
    }

    public static void main(String[] args)
    {
        System.out.println("factorielle 0 "+factorielle(0));
        System.out.println("factorielle 4 "+factorielle(4));
    }
}
```

Un autre exemple peut être donné avec le calcul de la somme des n premiers entiers.

$$\begin{aligned} \text{Cas de base} &: \text{ si } n = 0, \text{ cela vaut } 0 \\ \text{Cas general} &: \text{ cela vaut } n + (\text{la somme des } n - 1 \text{ premiers entiers}) \end{aligned} \tag{8.6}$$

On l'écrit en Java ainsi :

```
....
public static int somme(int n)
{
    if (n==0) return 0;
    else return (n+somme(n-1));
}
public static void main(String[] args)
{
    System.out.println("somme 0 premiers entiers "+somme(0));
    System.out.println("somme 4 premiers entiers "+somme(4));
}
}
```

8.2.2 Méthodes d'instance récursives

Nous étudions à présent le cas de méthodes d'instance récursives, sur l'exemple d'une classe représentant les files d'attente de personnes. Nous commençons par deux classes schématiques (file d'attente et personne).

```
public class FileAtt {
    private ArrayList<Personne> listePersonnes
        = new ArrayList<Personne>();
    public FileAtt(){
    public void entre(Personne p)
    {
        if (! this.listePersonnes.contains(p))
            this.listePersonnes.add(p);
    }
    .....}

public class Personne { // représente un spectateur
    private String nom = "nom inconnu";
    private String prenom = "prenom inconnu";
    private int age;
    private String ticket; // titre du film
    .....}
```

Examinons une méthode itérative classique (parcourant la liste des personnes pour calculer l'âge moyen des personnes qui se trouvent dans la file d'attente).

```
public int ageMoyen()
{
    if (this.listePersonnes.isEmpty())
        return 0;
    int somme = 0;
    for (Personne p : this.listePersonnes)
        somme += p.getAge();
    return somme/this.listePersonnes.size();
}
```

Pour l'écrire de manière récursive, on raisonne ainsi :

- une méthode principale vérifie que la liste n'est pas vide, appelle une méthode auxiliaire qui est elle-même récursive et qui effectue seulement la somme des âges, puis la méthode principale divise cette somme par le nombre de personnes présentes dans la file d'attente
- la méthode auxiliaire se pense ainsi :
 - un paramètre indique le point où on en est du parcours de la liste (curseur, appelé "début" dans le programme) ; il est à 0 lors du premier appel.
 - le cas de base est : si le curseur est après la fin de la liste, on retourne 0 (il n'y a plus de personnes à examiner).
 - le cas général pour effectuer la somme des âges est : la somme vaut l'âge de la personne sous le curseur auquel on ajoute la somme des âges des personnes qui sont dans la suite de la file d'attente.

```
public int ageMoyenRecursive()
{
    if (this.listePersonnes.isEmpty())
        return 0;
    return this.sommeAgeRecAux(0)/this.listePersonnes.size();
}

public int sommeAgeRecAux(int debut)
{
    if (debut==this.listePersonnes.size())
        return 0;
    else
        return this.listePersonnes.get(debut).getAge()
            +sommeAgeRecAux(debut+1);
}
```

Sur ce type de méthode, on peut trouver la récursivité difficile à penser en à mettre en œuvre. Elle va prendre tout son intérêt lorsque la structure parcourue est elle-même récursive.

8.2.3 Méthodes d'instance récursive sur des structures récursives

On repense la file d'attente de personnes (devant un cinéma) de manière récursive. Une file d'attente est :

- soit vide (cas de base)
- soit une personne, suivie d'une file d'attente (cas général)

Voici quelques premiers éléments de cette définition et d'un programme montrant une construction de file d'attente avec ce principe. Le choix réalisé ici pour simplifier est qu'une file d'attente sera vide si son attribut `premier` ne référence pas de personne.

```
public class FileAttente {
    private Personne premier = null;
    private FileAttente suiteFile;

    public FileAttente() {}
}
```



```
public FileAttente(Personne premier, FileAttente suiteFile) {
    this.premier = premier;
    this.suiteFile = suiteFile;
}

public boolean estVide(){return premier==null;}

// .....

public static void main(String[] argv)
{
    Personne p0 = new Personne("Alice","Livre",18,"cendrillon");
    Personne p1 = new Personne("Theo","Laforet",18,"blanche-neige");
    Personne p2 = new Personne("Hector","Dulac",23,"blanche-neige");

    FileAttente f0 = new FileAttente();
    FileAttente f1 = new FileAttente(p0, f0);
    FileAttente f2 = new FileAttente(p1, f1);
    FileAttente f3 = new FileAttente(p2, f2);
}
```

La définition des méthodes devient alors naturelle sous forme récursive en s'appuyant sur le cas de base (file d'attente vide) et sur le cas général (premier élément suivi d'une file d'attente). Nous commençons par reprendre la méthode qui calcule la somme des âges.

```
public int sommeAge()
{
    if (this.estVide())
        return 0;
    else
        return this.premier.getAge() + this.suiteFile.sommeAge();
}

public double ageMoyen()
{
    if (this.estVide()) return 0;
    return this.sommeAge()/this.nombreElements();
}
```

Ce schéma général s'applique encore dans les trois méthodes suivantes qui calculent respectivement le nombre d'éléments de la liste, une chaîne de caractères représentative, ou encore une méthode qui retourne une nouvelle file d'attente résultant de l'extraction des personnes attendant pour voir un film dont le titre est passé en paramètre.

```
public int nbElements()
{
    if (this.estVide())
        return 0;
    else
        return 1 + suiteFile.nbElements();
}
```

```

}

public String toString()
{
    if (this.estVide())
        return "end";
    else
        return this.premier + "\n"+this.suiteFile;
}

public FileAttente film(String f)
{
    if (this.estVide())
        return new FileAttente();
    else
        if (this.premier.getTicket().equals(f))
            return new FileAttente(this.premier, this.suiteFile.film(f));
        else return this.suiteFile.film(f);
}

```

8.2.4 Exploitation de l'héritage pour la définition des structures récursives

Grâce à l'approche par objets, on peut également utiliser les mécanismes de spécialisation et d'héritage pour distribuer la représentation des cas de base et des cas généraux dans différentes classes. Par exemple ici on représentera par trois classes les files d'attente de personnes (voir figure 8.1).

- Une classe abstraite `FileAttente` représente le concept général.
- Une sous-classe `FileAttenteVide` représente le cas particulier des files d'attente vides. Les méthodes contiendront les cas de base particuliers aux files vides. C'est une représentation plus satisfaisante que celle qui a été utilisée dans la section précédente où on s'appuyait sur une propriété (`premier` non initialisé) pour savoir si la file était vide.
- Une sous-classe `FileAttenteNonVide` représente le cas général des files d'attente non vides. Les méthodes contiendront les cas généraux.

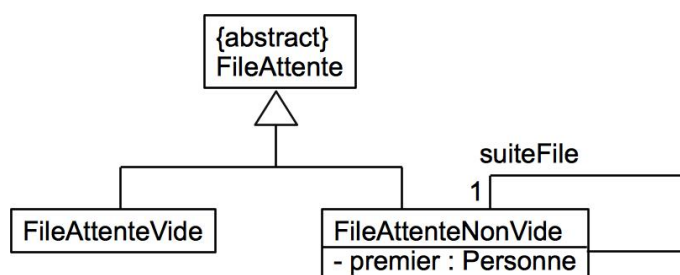


FIGURE 8.1 – Représentation des files d'attente récursives avec de l'héritage

```

public abstract class FileAttente {

```

```
public abstract boolean estVide();
public abstract int nbElements();
public abstract int sommeAge();
public abstract FileAttente film(String f);

public static void main(String[] args) {

    Personne p0 = new Personne("Alice",18,"cendrillon");
    Personne p1 = new Personne("Theo",18,"blanche-neige");
    Personne p2 = new Personne("Hector",23,"blanche-neige");
    Personne p3 = new Personne("Arthur",16,"cendrillon");
    Personne p4 = new Personne("Alex",16,"blanche-neige");
    // observer les 2 cas de création ci-dessous (file vide versus file non vide)
    FileAttente f0 = new FileAttenteVide();
    FileAttente f1 = new FileAttenteNonVide(p4, f0);
    FileAttente f2 = new FileAttenteNonVide(p3, f1);
    FileAttente f3 = new FileAttenteNonVide(p2, f2);
    FileAttente f4 = new FileAttenteNonVide(p1, f3);
    FileAttente f5 = new FileAttenteNonVide(p0, f4);

    System.out.println(f5);
    System.out.println(f5.sommeAge());
    System.out.println(f5.film("blanche-neige"));
}
}

public class FileAttenteVide extends FileAttente {

    @Override
    public boolean estVide(){return true;}

    @Override
    public int nbElements(){return 0;}

    @Override
    public int sommeAge() {return 0;}

    @Override
    public FileAttente film(String f) {
        return new FileAttenteVide();
    }

    public String toString(){
        return ".";
    }
}

public class FileAttenteNonVide extends FileAttente {
```

```
private Personne premier;
private FileAttente suiteFile;

public FileAttenteNonVide(Personne premier, FileAttente suiteFile) {
    this.premier = premier;
    this.suiteFile = suiteFile;
}

@Override
public boolean estVide(){return false;}

@Override
public int nbElements(){return 1+suiteFile.nbElements();}

@Override
public int sommeAge() {
    return this.premier.getAge()+this.suiteFile.sommeAge();
}

@Override
public FileAttente film(String f) {
    if (this.premier.getTicket().equals(f))
        return new FileAttenteNonVide(this.premier, this.suiteFile.film(f));
    else
        return this.suiteFile.film(f);
}

public String toString(){
    return this.premier.getNom() + " "+this.suiteFile.toString();
}
}
```

8.3 Conclusion

Pour profiter de la récursivité, il faut identifier les algorithmes et structures naturellement récursifs.

Enfin, il faut penser récursivement lors de la conception en décomposant le problème en :

- Ses cas particuliers, ses cas de base, sur les données de petite taille
- Ses cas généraux, qui se ramènent à des sous-problèmes identiques (à des problèmes d'ordre inférieur)

La programmation par objet a de plus cette qualité spécifique de répartir les cas de base et les cas généraux dans les différentes classes qui les représentent et qui sont organisées dans une hiérarchie d'héritage.