

Modélisation et programmation par Objets 1

HLIN406

Marianne Huchard, Clémentine Nebut

12 janvier 2017

Ces notes de cours sont 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.

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).

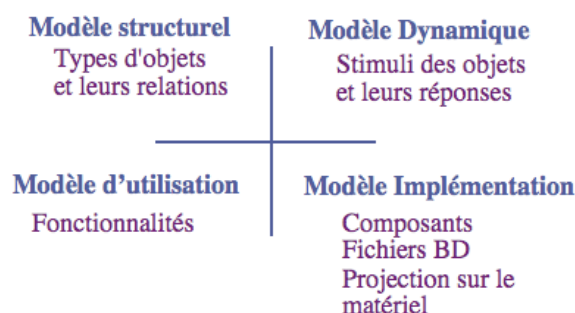


FIGURE 1.1 – Vue générale des modèles UML

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 paquetages : les éléments de base du modèle statique

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 structurales 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.

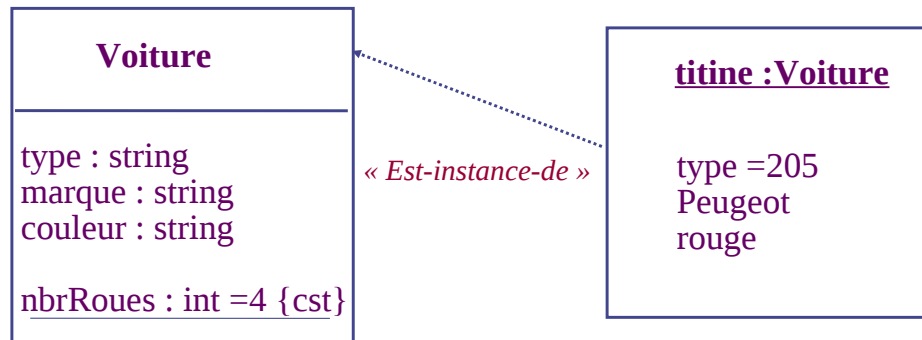


FIGURE 2.1 – Classe (à gauche) et objet/instance (à droite)

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* ∈ {+, −, #, ~}, 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).

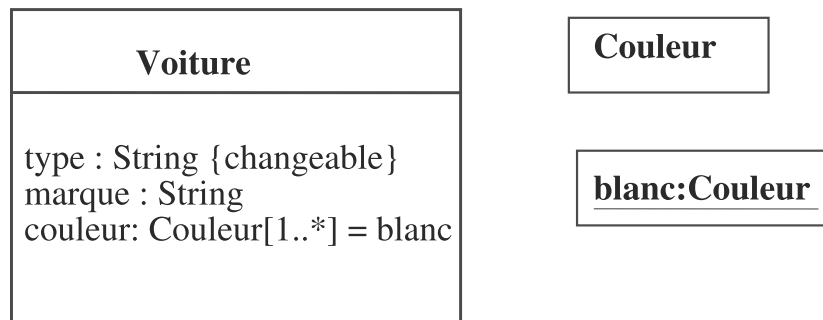


FIGURE 2.2 – Détails sur la syntaxe de description des attributs

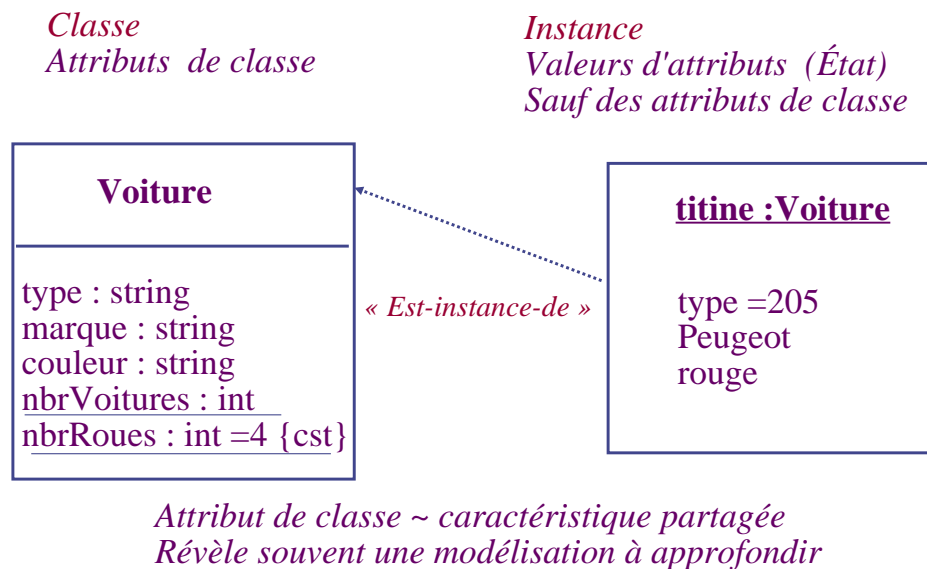


FIGURE 2.3 – Attributs de classe

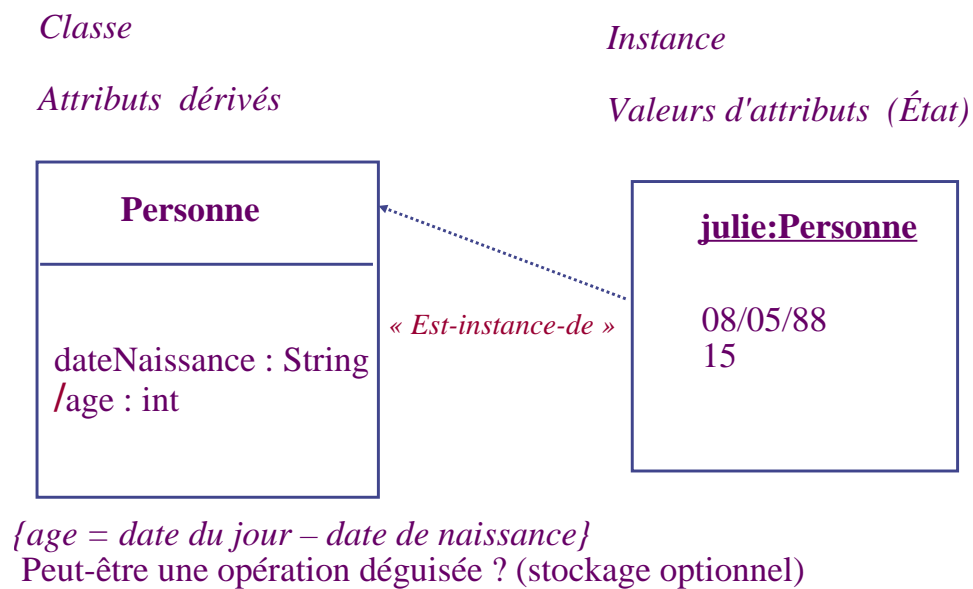


FIGURE 2.4 – Attribut dérivé

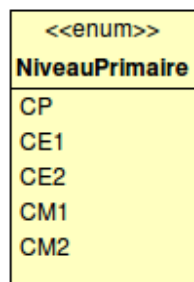


FIGURE 2.5 – Énumération en UML

Les 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
- 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).

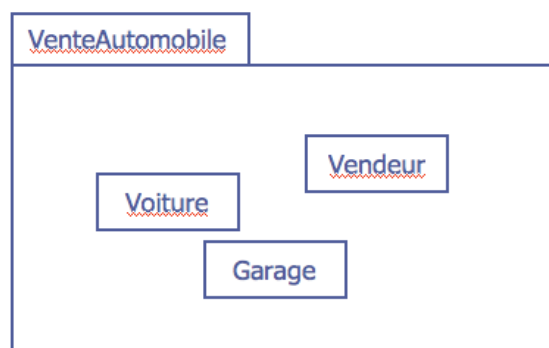


FIGURE 2.6 – Paquetages en UML

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]

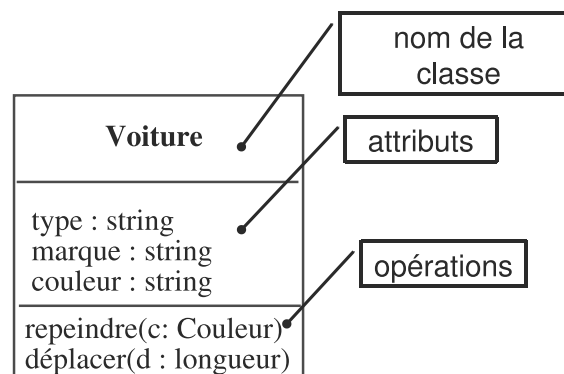


FIGURE 3.1 – Les opérations dans les classes UML

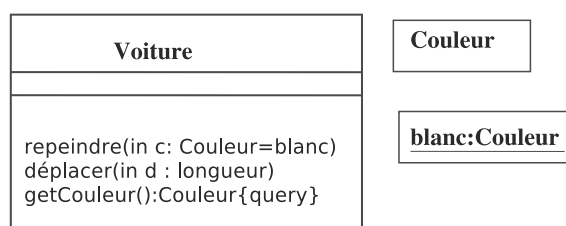


FIGURE 3.2 – Exemple d’opérations en UML

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. Dénuté +. Signifie que cette opération pourra être appelée par n’importe quel objet.
- Privée. Dénuté -. Signifie que cette opération ne pourra être appelée que par des objets instances de la même classe.
- Paquetage. Dénuté ~. Signifie que cette opération ne pourra être appelée que par des objets instances de classes du même paquetage.
- Protégée. Dénuté #. 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

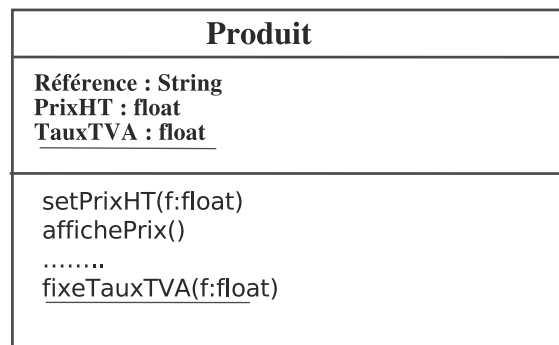


FIGURE 3.3 – Opérations de classe

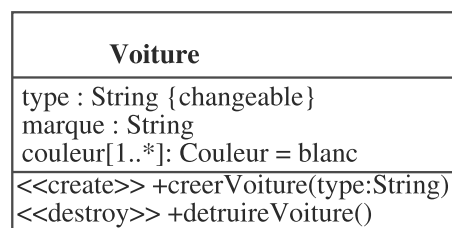


FIGURE 3.4 – Constructeurs et destructeurs en UML

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).

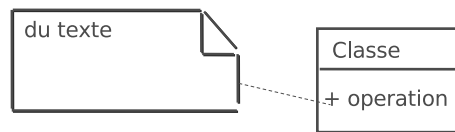


FIGURE 3.5 – Note UML

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)
```

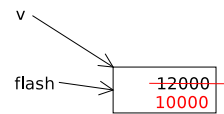
Un exemple est donné au listing 3.4. Le **get** peut permettre de faire des statistiques sur les accès à l'attribut. Le **set** peut permettre d'effectuer des vérifications sur les valeurs, ou bien encore de prendre en charge des attributs dérivés. Un attribut dérivé peut être implémenté par une méthode, 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.