

Introduction à la programmation en Java

25 septembre 2007

Cours 1

Introduction et premiers éléments de Java

Dans ce cours, nous introduisons la programmation par objets, décrivons les éléments de Java correspondant au premier cours d'algorithmique impérative (nous n'y parlerons pas encore vraiment d'objets), puis nous ajoutons une forme très simplifiée de la notion de classe.

1.1 La programmation par objets

1.1.1 Les grands principes

Nous avons vu en algorithmique un style de programmation « impératif » qui est apparu comme une généralisation du langage machine, que l'on a essayé de rendre plus lisible en lui ajoutant des structures de contrôle (conditionnelles, répétitives) plus adaptées que le « branchement » à une étiquette. Ce style de programmation se fonde théoriquement sur le modèle de machine de Turing. Parmi les langages les plus connus, on trouve BASIC, PASCAL, C.

Un autre style de programmation tout aussi ancien se base sur la définition et l'évaluation de fonctions : il s'agit du style « fonctionnel », repris par le célèbre langage LISP, dont le modèle théorique est le lambda-calcul.

Un troisième style est plus déclaratif et se base sur la logique, il s'agit de la « programmation logique », mise en œuvre en particulier dans le langage PROLOG.

L'idée essentielle de la programmation par objet est qu'écrire un programme doit se rapprocher d'un exercice de simulation, et les premiers programmes à objets étaient justement des programmes de simulation. Dans cette simulation, on s'intéresse plus précisément aux types de données abstraits (« sur quoi travaille-t-on et comment manipule-t-on les données ») plutôt qu'aux traitements seuls (« comment fait-on les choses »). Les fonctionnalités d'un système sont en effet jugées moins stables dans le temps, et moins réutilisables que les objets sur lesquels on travaille.

Pour cela les notions suivantes ont été introduites :

- l'*objet* dénote une entité du monde que l'on cherche à représenter dans le programme ;

- on distingue pour les objets le statut de *classe* et le statut d'*instance* (tout au moins dans le modèle dit « à classes », qui est dominant) ;
- les classes sont organisées dans des hiérarchies de spécialisation/généralisation qui font écho aux classifications du domaine que l'on modélise ;
- l'aspect opérationnel se base sur « l'envoi de message » qui est une forme d'appel de fonction un peu particulière.

1.1.2 Les principaux langages à objets

La programmation par objets est née en 1967, avec le langage SIMULA.

Les langages peuvent se diviser globalement en deux groupes :

- les langages qui étendent un langage impératif ou fonctionnel existant, comme C++ (1985, qui étend C), CLOS (resp. LISP).
- les langages intégralement développés selon le paradigme objet, tels qu'EIFFEL (1988), ou Smalltalk (créé en 1972, où les structures de contrôles sont elle-mêmes des messages, et où tous les types sont des classes ou se comportent comme telles).

1.2 Le langage Java

Le langage Java est plus récent (1991). Il n'a rien inventé, mais fait une bonne synthèse des concepts des langages qui le précèdent. Il a été développé dans le but de proposer un langage :

- plus simple que C++, qui était le plus largement utilisé,
- avec de riches bibliothèques de classes (surtout pour le développement d'interfaces graphiques),
- qui permet d'écrire des programmes dans des environnements distribués (pour Internet, en particulier, et pour cela, il a été conçu avec des exigences de sécurité et de portabilité).

L'une des particularités de Java, est de fonctionner grâce à deux programmes :

- un *compilateur* analyse le code source et produit un programme en « bytecode », langage d'une machine abstraite (la machine virtuelle Java),
- un *interpréteur* traduit à la volée le bytecode dans le langage machine natif, et donc « exécute » le programme.

Java garde certaines caractéristiques d'un langage de programmation impératif (les structures de contrôle, les types « primitifs » tels que les entiers, les réels, les caractères, qui ne sont pas des classes, les tableaux qui ont un statut entre les classes et les types ordinaires). Pour faire une transition en douceur avec les cours d'algorithmique, nous commençons par vous présenter les types « primitifs ». La classe chaîne complète ces premiers types avec un léger parfum d'objet, puis nous introduisons la notion de classe.

1.3 Quelques formules magiques pour débiter en Java

1.3.1 Le programme

Nous vous présentons ci-dessous la structure d'un programme Java minimal dans lequel on pourrait effectuer des saisies et des affichages. Absorberez-le tel quel pour l'instant, nous rem-

placerons les points par des instructions, et à l'issue de cette série de cours, vous comprendrez mieux toutes ces expressions.

Ce programme source doit impérativement être dans un fichier dont le nom est `ExempleDeProgramme.java`.

```
// Déclare que ce programme appartient à un ensemble de programmes qui se nomme
// Prog.MesExemples

package Prog.MesExemples;

//Permettra d'utiliser les fonctions correspondant à lire clavier et écrire écran

import java.io.*;

// Emballage du programme

public class ExempleDeProgramme
{
    public static void main(String argv[]) throws IOException
    {
        .....;
        .....;
        .....;

    }
}
```

Notez aussi pour la suite que les instructions se terminent obligatoirement par un ; et que les caractères { et } marquent le début et la fin d'un bloc d'instructions.

Le bloc `main` est la partie principale d'un programme. L'exécution du programme commence par le début de ce bloc, et se termine quand on en sort.

1.3.2 Les commentaires

En Java, il existe trois sortes de commentaires :

- les commentaires limités à une portion de ligne, qui débutent par //
- les commentaires ayant un nombre indéterminé de lignes, le commentaire est dans ce cas inclus entre les balises /* et */
- les commentaires qui seront utilisés par un programme qui génère automatiquement la documentation (l'utilitaire **javadoc**), le commentaire est dans ce cas inclus entre les balises /** et */

1.4 Types, variables, expressions, instructions simples

1.4.1 Le type booléen (ou logique)

Nom en Java. `boolean`

Domaine. **V** s'écrit `true`, **F** s'écrit `false`

Opérations.

Algorithmique	non	egal	non_egal	et	ou
Java	!	==	!=	&& &	

Vous remarquerez que Java propose deux opérateurs pour le **et** et le **ou** vus en algorithmique. Les opérateurs `&` et `|` sont de véritables opérateurs « logiques », qui évaluent leurs opérandes gauche et droit avant d'effectuer l'opération, tandis que les opérateurs `&&` et `||` n'évaluent l'opérande droit *que si* la valeur de l'opérande gauche ne suffit pas pour conclure. En programmation, nous utiliserons généralement les opérateurs `&&` et `||` parce qu'un impératif d'efficacité s'ajoute à celui de lisibilité (nous ne recherchons cependant pas l'efficacité à tout prix).

Exemple.

Pour deux variables booléennes `a` et `b`,

- évaluer `a || b` revient à évaluer tout d'abord `a` ;
si `a` vaut `true`, `b` n'est pas évalué, si `a` vaut `false`, `b` est évalué.
- évaluer `a | b` revient à évaluer `a` et `b`, puis en déduire le résultat.

Expressions.

`true || false` (résultat booléen égal à `true`)

`(a || b) && (c || V)` (résultat booléen) si `a`, `b`, `c` sont des variables booléennes.

1.4.2 Le type entier

Il y a quatre types représentant les entiers en Java (`int`, `short`, `long`, `byte`). Ces types se différencient par la taille de l'intervalle de \mathcal{Z} qui est couvert. Nous n'en utiliserons qu'un.

Nom en Java. `int`

Domaine. $[-2^{31}, 2^{31} - 1]$, une valeur du type `int` est en effet codée sur 4 octets en mémoire (32 bits, dont l'un sert au codage du signe). Ceci devrait nous suffire : $2^{31} = 2\,147\,483\,647$.

Opérations.

Algorithmique	+	-	*	/	%	<	>	≤	≥	egal	non_egal
Java	+	-	*	/	%	<	>	<=	>=	==	!=

Les priorités sont les mêmes que celles définies en algorithmique (*, /, % sont prioritaires par rapport à + et -). En dehors de ces priorités, l'évaluation se fait de gauche à droite.

Expressions.

11/3, 11%3, 4 * x, v > 4

1.4.3 Le type réel

Le langage Java propose deux types pour les réels (float, double), nous n'en étudierons qu'un. Ils se différencient par l'intervalle de valeurs, et la précision.

Nom en Java. double

Domaine.

L'intervalle est $[-1.79...10^{308}, +1.79...10^{308}]$, avec 15 chiffres significatifs. Ils sont codés sur 8 octets.

Les réels se notent avec un point à la place de la virgule (notation anglaise) : 1.05 , +1.05 , -1.005

avec un exposant : 1.3E+2 ou 1.3E2 (pour $1.3 * 10^2 = 130.0$), 1.3E-2 (pour $1.3 * 10^{-2} = 0.013$)

Opérations.

Algorithmique	+	-	*	/	%	<	>	≤	≥	egal	non_egal
Java	+	-	*	/	%	<	>	<=	>=	==	!=

Comme en algorithmique, l'opération / n'est pas ambiguë, c'est l'opération entière seulement si les deux opérandes sont des entiers. Les opérateurs == et != sont définis à la précision près.

Expressions.

11/3.0 vaut 3.6666666666666666

1.4.4 Le type caractère

Nom en Java. char

Domaine.

Les caractères qui ont une représentation dans le système Unicode (jeu de 65536 caractères possibles, dont 35000 sont employés). Les passionnés pourront consulter : <http://www.unicode.org/>
Les caractères se notent entre apostrophes simples (ou « quote » en anglais) : 'A' 'a' '+' '1'

Quelques caractères spéciaux vous seront utiles :

<code>\t</code>	tabulation horizontale
<code>\n</code>	saut de ligne
<code>\"</code>	guillemets
<code>\'</code>	apostrophe

Opérations.

Algorithmique	<	>	≤	≥	egal	non_egal
Java	<	>	<=	>=	==	!=

Expressions.

'a' < 'z' (résultat booléen : true)

1.4.5 Le type chaîne de caractères

Java propose un type, qui est plus précisément *une classe*. Nous n'en présentons ici qu'un premier aperçu, qui sera complété dans une prochaine section.

Nom en Java. String

Domaine.

Un ensemble de suites finies (ou séquences) de caractères.

Les chaînes de caractères se notent entre guillemets (ou « double quote » en anglais) : "prune", "citron \n orange", "" (chaîne vide), "a" (chaîne formée du seul caractère 'a'), "abc\nde" (chaîne formée des caractères a b c " d e).

Opérations.

La concaténation se note + comme en algorithmique, par exemple "il"+"pleut". Les opérations de comparaison sont plus délicates à mettre en œuvre.

`t egal "citron"` se traduira en Java par `t.equals("citron")`

`"citron" egal "citron"` se traduira en Java par `"citron".equals("citron")`

`t non_egal "citron"` se traduira en Java par `!t.equals("citron")`

`t1 < t2` se traduira en Java par `t1.compareTo(t2) < 0`

`t1 > t2` se traduira en Java par `t1.compareTo(t2) > 0`

Nous éclaircirons cette écriture plus loin.

Un petit mystère bien utile.

On peut concaténer une chaîne et un objet d'un autre type, par exemple écrire "le citron numéro "+4+" est sur l'étagère", qui sera compris comme la chaîne "le citron numéro 4 est sur l'étagère". Ceci est dû au fait que Java sait transformer tout objet en chaîne de caractères. Ce point sera approfondi par la suite.

1.5 Déclaration de variables locales

Elle se fait dans le corps du programme et ressemble de très près à ce que nous écrivions dans la colonne « objets utilisés ».

```
package Prog.MesExemples;
import java.io.*;
public class illustreDéclaration
{
    public static void main(String argv[]) throws IOException
    {
        int i;
        double r;

        .....

    } // fin de la partie "main"
} // fin de la classe
```

1.6 Instructions simples

1.6.1 Affectation

Algorithmique	Java
$NV \leftarrow E$	$NV = E;$

où NV est un identificateur de variable, et E une expression de même type.

Le programme ci-dessous illustre cette instruction d'affectation, on remarque que l'on peut faire conjointement une déclaration de variable et son initialisation.

```
package Prog.MesExemples;
import java.io.*;
public class illustreAffectation
{
    public static void main(String argv[]) throws IOException
    {
```

```

    int i;
    i = 2;
    i = i + 4;

    double r = i / 4;

} // fin de la partie "main"
} // fin de la classe

```

1.6.2 Saisie de valeur

Nous vous rappelons la forme algorithmique de cette instruction.

```

lire clavier NV

où NV est un identificateur de variable

```

Vous ne nous croirez certainement pas si nous vous disons que cela se traduit ainsi en Java, dans le cas de la lecture d'un entier (et c'est encore différent s'il s'agit d'un réel)

```

// déclaration d'un objet représentant le clavier
BufferedReader clavier = new BufferedReader(new InputStreamReader(System.in));
// déclaration de l'entier que l'on veut saisir
int i;
// lire clavier i
i = Integer.parseInt(clavier.readLine());

```

Pour vous épargner de telles écritures, nous vous proposons une classe « maison » qui va simplifier les choses. Elle vous sera fournie lors des travaux pratiques. Son utilisation est décrite par le programme Java suivant qui saisit des valeurs de divers types.

```

package Prog.MesExemples;
import Prog.MesEntreesSorties.ConsoleStandard;
import java.io.*;

public class TestConsole
{
    public static void main(String[] a) throws java.io.IOException
    {
        // création d'un objet représentant le clavier
        ConsoleStandard clavier = new ConsoleStandard();
        // exemples de saisies
        int i; i = clavier.lireEntier();           // pour << lire clavier i >>
        String s; s = clavier.lireChaine();       // pour << lire clavier s >>
    }
}

```

```

    double d; d = clavier.lireDouble();    // pour << lire clavier d >>
    char c; c = clavier.lireCaractere();   // pour << lire clavier c >>
    boolean b; b = clavier.lireBooleen(); // pour << lire clavier b >>
}
}

```

Il n'y aura pas de forme simplifiée de saisie équivalente à celle que nous avons vue en algorithmique.

1.6.3 Affichage de valeur

La forme algorithmique de cette instruction était la suivante.

```

écrire écran E

où E est une expression

```

En Java cela se traduit plus simplement que dans le cas de la saisie.

```
System.out.print(E);
```

Grâce à ce que nous savons sur les chaînes de caractères, la forme simplifiée `écrire écran E1,E2, ... EN;` se traduira par :

```
System.out.print(""+E1+E2+...+EN);
```

Enfin, sachez qu'il existe une variante de cette instruction, qui passe à la ligne après avoir écrit son paramètre :

```
System.out.println(4);
```

Ce qui équivaut à écrire :

```
System.out.print(4+'\n');
```

Pour les réels, vous aurez de petits soucis d'affichage, par exemple, `System.out.print(10.0/3)` ; affichera `3.333333333333335`, ce qui est un peu long. Le petit programme suivant vous montre comment réduire cet affichage à 2 décimales :

```

package Prog.MesExemples;
import java.io.*;
import java.text.*;

public class saisie
{

```

```

public static void main(String argv[]) throws IOException
{
    DecimalFormat df = new DecimalFormat("0.##");
    System.out.println(df.format(10.0/3));
} // fin de la partie "main"
} // fin de la classe

```

1.7 Un algorithme simple

Nous vous proposons ci-dessous deux versions d'un algorithme simple.

1.7.1 Somme et produit

Algorithme *SommeEtProduit*

Données : *deux réels lus au clavier*

Résultats : *affiche la somme et le produit des deux réels*

Objets utilisés	Instructions
réels x, y /* reçoivent les valeurs lues */	écrire écran "entrez deux réels" lire clavier x lire clavier y écrire écran "somme = ", $x+y$," produit = ", $x*y$

Dans une première version, deux variables locales, `somme` et `produit` sont introduites pour effectuer les calculs intermédiaires.

```

package Prog.MesExemples;
import Prog.MesEntreesSorties.ConsoleStandard;
import java.io.*;

public class SommeEtProduit
{
    public static void main(String[] a) throws java.io.IOException
    {
        ConsoleStandard clavier = new ConsoleStandard();

        double x, y;
        System.out.println("entrez deux réels ");
        x = clavier.lireDouble();
        y = clavier.lireDouble();
        double somme = x+y;
        double produit = x*y;
    }
}

```

```

        System.out.println("somme = "+somme+" produit = "+produit);
    }
}

```

Voici l'effet sur l'écran de ce programme (4 et 1.2 correspondent évidemment à l'écho des saisies de x et y.

```

entrez deux réels
4
1.2
somme = 5.2 produit = 4.8

```

Nous vous proposons également un programme qui n'utilise pas ces deux variables intermédiaires.

```

package Prog.MesExemples;
import Prog.MesEntreesSorties.ConsoleStandard;
import java.io.*;

public class SommeEtProduit
{
    public static void main(String[] a) throws java.io.IOException
    {
        ConsoleStandard clavier = new ConsoleStandard();

        double x, y;
        System.out.println("entrez deux réels ");
        x = clavier.lireDouble();
        y = clavier.lireDouble();
        System.out.println("somme = +(x+y)+" produit = +(x*y));
    }
}

```

Vous pourrez noter dans ces programmes les points suivants :

- une déclaration conjointe de deux variables de même type : `double x, y;`
- dans le deuxième programme, les parenthèses qui encadrent les expressions `x+y` et `x*y` : la première est obligatoire (sinon les deux réels sont affichés plutôt que leur somme), la deuxième ne l'est pas (mais elle rend le programme plus lisible).

1.8 Un petit pas vers les classes

Du point de vue de la conception, une classe est le descripteur d'un ensemble d'objets qui ont des propriétés (attributs) et un comportement semblables, et le même type de relations avec les objets des autres classes. Dans un premier temps, nous laisserons de côté l'aspect

comportemental pour nous focaliser sur l'aspect statique (attributs et relations). Nous allons partir d'une classe décrite en UML, étudier sa traduction en Java, créer des instances et les manipuler.

1.8.1 Une classe *Personne*

La figure 1.1 montre le diagramme UML d'une classe décrivant le concept de *Personne* et deux instances. Nous commençons par supposer que la classe ne dispose que d'attributs.

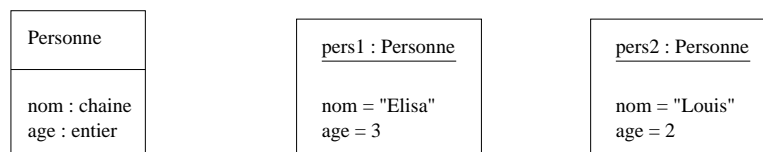


FIG. 1.1 – Une classe *Personne* et deux instances.

La partie statique de cette classe se traduit syntaxiquement par la classe Java qui suit. Vous devez considérer cette partie de code comme une *description*, et non pas comme des instructions à exécuter. Attention, cette écriture Java de la classe est ultra-simplifiée ... D'un point de vue pratique, cette déclaration de classe doit être dans un fichier dont le nom est `Personne.java`.

```
package Prog.LaVraieVie;
public class Personne
{
    private String nom;
    private int age;
}
```

1.8.2 Classes et types

Du point de vue du langage de programmation, lorsqu'on définit une classe, on définit aussi un type au sens défini dans le cours d'algorithmique. Son domaine, c'est-à-dire l'ensemble de ses valeurs, est l'ensemble des instances, appelé aussi *extension* de la classe. Le vecteur formé des valeurs des attributs d'une instance est ce que nous appellerons l'*état de l'instance*. Nous verrons dans le cours prochain la manière de décrire les opérations autorisées sur les instances. L'ensemble des attributs et des opérations forme l'*intension* de la classe.

1.8.3 Utilisons la classe *Personne*

1.8.3.1 Créer des instances

Nous présentons l'expression de création d'instance dans sa forme la plus simple. Elle correspond à l'appel d'une fonction particulière que nous détaillerons plus tard.

```
new C()
```

où C est une classe

Une manière de l'utiliser sera de déclarer tout d'abord une variable de type C, puis d'affecter l'expression ci-dessus à cette variable. Par exemple :

```
C instanceDeC;  
instanceDeC = new C();
```

1.8.3.2 Accès aux attributs

L'accès à un attribut se fait par l'intermédiaire d'accesseurs.

```
i.geta()
```

où i est une instance de la classe C qui dispose d'un attribut a

1.8.3.3 Application à la classe Personne

```
// la classe Personne et le ProgrammePersonne appartiennent au même paquetage  
package Prog.LaVraieVie;
```

```
// le programme utilise les fonctionnalités d'entrée/sortie  
import Prog.MesEntreesSorties.ConsoleStandard;  
import java.io.*;
```

```
public class ProgrammePersonne  
{  
    public static void main(String argv[]) throws IOException  
    {
```

```
        ConsoleStandard clavier = new ConsoleStandard();
```

```
        // déclaration d'une variable de type Personne
```

```
        Personne p1;
```

```
        // création de l'instance pers1
```

```
        p1 = new Personne();
```

```
        // initialisation des attributs de l'instance pers1
```

```

p1.setNom( "Elisa");
p1.setAge(3);

Personne p4 = p1; // deux variables de même valeur (pers1)

// création et manipulation de l'instance pers2
Personne p2 = new Personne();
p2.setNom("Louis");
p2.setAge(30);

// création et manipulation d'une instance avec saisie des attributs
Personne p3 = new Personne();
System.out.println("Donnez le nom de la personne");
p3.setNom(clavier.lireChaine());
System.out.println("Donnez l'age de la personne");
p3.setAge(clavier.lireEntier());

// affichage du nom des trois personnes créées
System.out.println("Voici les noms des personnes créées : "
    +p1.getNom()+", "
    +p2.getNom()+", et "
    +p3.getNom+". "
    );

// modification de l'age d'une personne
p3.setAge(31);

// affichage de la moyenne d'age des trois personnes créées
System.out.println("Voici la moyenne d'age des personnes créées : "
    +((p1.getAge()+p2.getAge()+p3.getAge())/3.0)+"."
    );

} // fin de la partie "main"
} // fin de la classe

```

1.8.3.4 Représentation schématique des instances

De la même manière que nous faisons des traces pour les algorithmes impératifs, nous aurons souvent recours à une représentation imagée des variables dont le type est une classe. La valeur de ces variables est une instance, qui est dénotée par un identifiant (une désignation, qui correspond aussi à ce que l'on appelle l' O.I.D. ou identifiant d'objet, et en pratique, il s'agit souvent d'une adresse en mémoire) que nous pourrons d'ailleurs voir en Java (mais cette désignation ne sera pas aussi explicite que les identifiants `pers1` et `pers2`). Nous représentons l'état de l'instance (les valeurs de ses attributs) par une boîte qui contient autant de cases que

d'attributs. Les noms des attributs sont placés sur un côté, et leurs valeurs dans les cases. La désignation de l'instance dont on représente l'état figure sous la boîte, entre parenthèses. La figure 1.2 montre les représentations des variables `p1`, `p2` et `p4` de notre exemple. A titre de comparaison, nous avons fait aussi figurer la représentation de deux variables entières `x` et `y` supposées initialisées avec la même valeur 2 (dénotée par ce même symbole) pour vous montrer qu'il s'agit du même mécanisme.

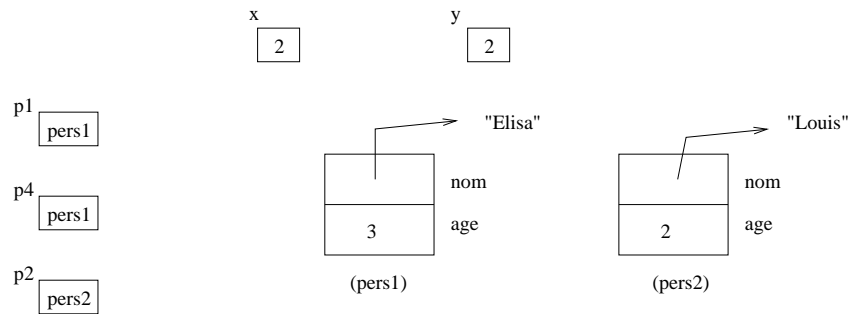


FIG. 1.2 – Représentation schématique d'instances de *Personne* et de leur état.

Une autre représentation usuelle consiste à remplacer les désignations d'instances par des flèches (voir figure 1.3).

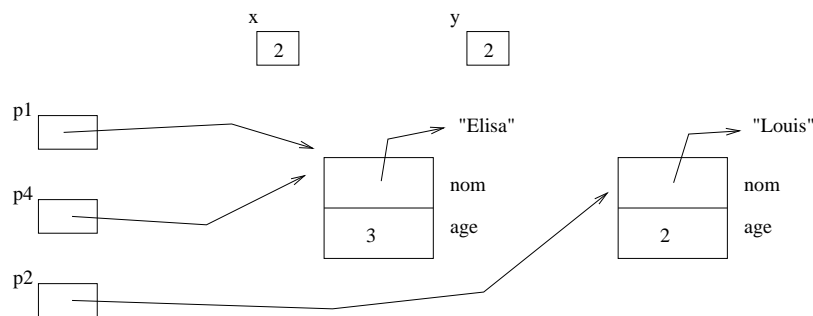


FIG. 1.3 – Désignations et flèches.

1.8.4 Représentation des associations

Dans le langage Java, il n'y a pas de type « associations » comme il y a des types « classes ». Différentes manières sont utilisées pour traduire les associations : un ou plusieurs attributs, ou bien une classe qui représente l'association. A ce point du cours, nous vous montrons comment une association simple peut se traduire à l'aide d'attributs. Nous partons d'un diagramme de classes complétant le précédent, ajoutant le concept *Voiture*, et une association représentant la possession d'une voiture par une personne. On suppose ici qu'une personne dispose au plus d'une voiture et qu'une voiture appartient à au plus une personne (voir figure 1.4).

Voici les classes Java correspondantes (à placer dans deux fichiers différents). Vous remarquerez qu'à chaque extrémité d'association correspond un attribut dans la classe incidente. Lorsque

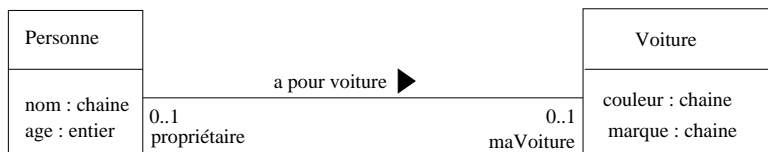


FIG. 1.4 – Personnes et voitures - diagramme de classes.

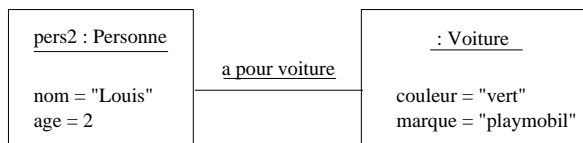


FIG. 1.5 – Personnes et voitures - diagramme d'instances.

l'association n'est navigable que dans un sens, la classe d'arrivée ne contiendra généralement pas d'attribut correspondant à l'association. Il est conseillé d'utiliser les noms de rôle de l'association comme noms d'attributs. Tant que cet attribut n'est pas initialisé, il a une valeur indéterminée. La figure 1.6 propose un schéma de conception qui montre le résultat de ce choix d'implémentation. Il ne reste plus qu'à écrire le code correspondant.

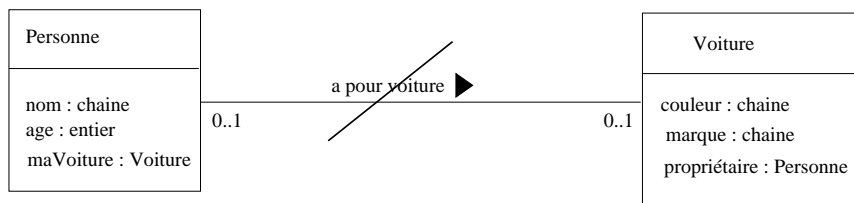


FIG. 1.6 – Schéma de conception pour « a pour voiture ».

```

package Prog.LaVraieVie;
public class Personne
{
    private String nom;
    private int age;
    private Voiture maVoiture;
}

package Prog.LaVraieVie;
public class Voiture
{
    private String couleur;
    private String marque;
    private Personne proprietaire;
}

```

Exercice. Ecrivez un programme permettant de créer les éléments du diagramme d'instances de la figure 1.5.

1.8.5 Initialisations

Avant de quitter les classes, sachez que plutôt que de laisser une variable dont le type est une classe avec une valeur indéterminée, on peut l'initialiser avec la valeur `null`.

On peut également donner des valeurs « par défaut » aux attributs d'une classe. Ceci est réalisé en affectant ces valeurs aux attributs dans la déclaration de la classe, comme le montre la version de la classe `Personne` présentée ci-dessous.

```
package Prog.LaVraieVie;
public class Personne
{
    private String nom = "inconnue";
    private int age = -1;
    private Voiture maVoiture = null;
}
```

Lorsqu'une instance est créée, en l'absence d'une autre initialisation, ses attributs sont initialisés avec les valeurs par défaut que vous donnez. Si vous ne le faites pas, Java utilise ses propres valeurs par défaut qui dépendent du type, par exemple 0 pour les nombres, `null` pour les désignations d'instances (y compris les chaînes de caractères), `false` pour les booléens.

Attention, une variable locale ne dispose pas en Java de valeur par défaut (c'est comme en algorithmique), ce n'est vrai que pour les attributs !

1.9 Organisation par paquetages

Java est un langage qui prône l'ordre dans les répertoires ! Il incite en particulier à ranger ses classes dans des paquetages (ensembles de classes), dont la structure correspond à la structure de vos répertoires. Mais surtout un paquetage réunit un ensemble de classes qui ont un même propos : par exemple on réunit les classes illustrant la partie algorithmique de ce cours dans le paquetage `MesExemples`, et les classes illustrant la partie objet dans le paquetage `LaVraieVie`. Tous les paquetages définis ici (hors ceux de Java) sont inclus dans un ensemble plus large que nous avons appelé `Prog`, et ce nom sert de ce fait de préfixe aux noms de paquetage. La notion de paquetage est plus large : UML en propose ainsi une notation spéciale. La figure 1.7 vous présente les paquetages (à gauche) et les répertoires (à droite) correspondants aux classes de ce cours. Les paquetages sont reliés par une dépendance `«import»`, dépendance standard pour UML et qui traduit le fait que les noms des éléments publics du paquetage cible sont directement utilisables par le paquetage source.

Les noms de paquetage servent aussi au compilateur et à l'interpréteur Java à retrouver les classes.

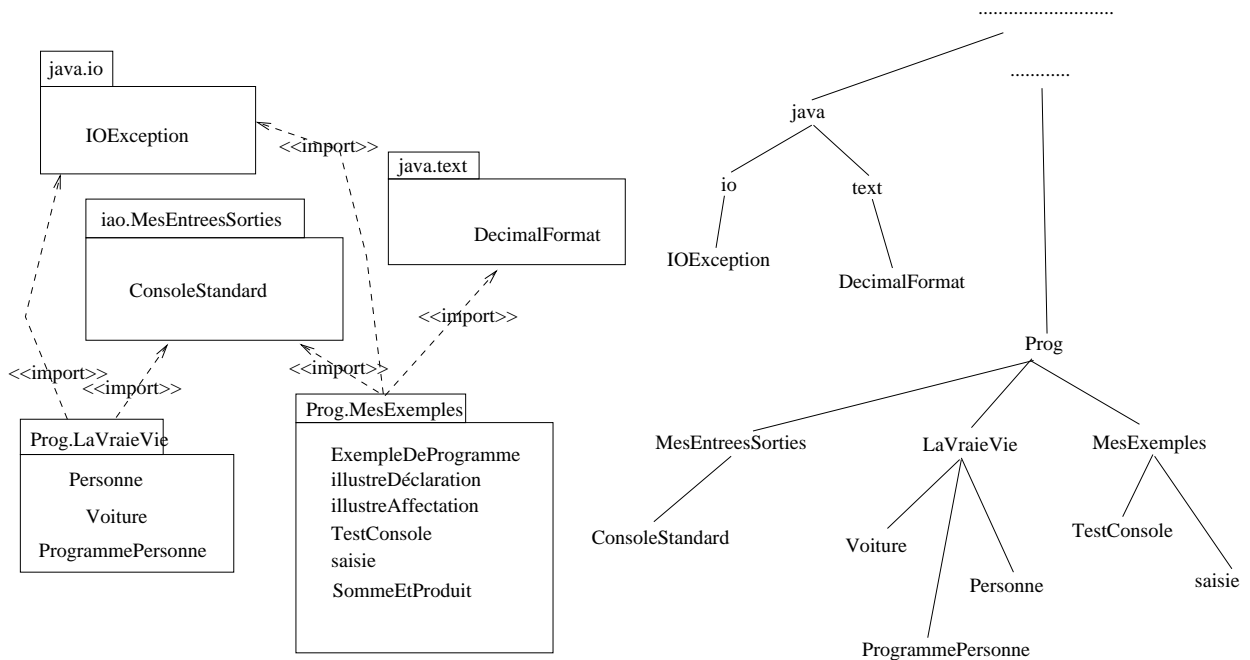


FIG. 1.7 – Paquetages.

Cours 2

Instructions conditionnelles - Méthodes

2.1 Instructions conditionnelles

2.1.1 Conditionnelle à une possibilité

Syntaxe algorithmique.

```
si  $C$  alors  
     $A_1; A_2; \dots; A_n$   
fsi
```

où C est une condition, $A_1; A_2; \dots; A_n$ sont des instructions.

Syntaxe Java.

```
if  $C$   
    {  $A_1; A_2; \dots; A_n$ ; }
```

```
if  $C$   
    A;
```

Algorithme *EcrireOrdre1*

Données : *deux entiers lus au clavier*

Résultats : *affichage à l'écran du plus petit puis du plus grand de ces nombres*

Principe : Les deux nombres sont placés dans deux variables *a* et *b*. Si nécessaire les contenus de *a* et *b* sont échangés de sorte que *a* contienne le plus petit, et *b* le plus grand.

Objets utilisés	Instructions
entiers <i>a, b</i> les entiers lus	écrire écran "entrez deux entiers" lire clavier <i>a, b</i>
entier <i>vi</i> sert pour l'échange	si $a > b$ alors /* échange des valeurs */ $vi \leftarrow a$ $a \leftarrow b$ $b \leftarrow vi$ fsi écrire écran <i>a, ' ', b, '\n'</i>

```
package Prog.MesExemples;
import Prog.MesEntreesSorties.ConsoleStandard;
import java.io.*;

public class EcrireOrdre1
{
    public static void main(String argv[]) throws IOException
    {
        // variables locales
        int a,b,vi;
        ConsoleStandard clavier = new ConsoleStandard();

        System.out.println("Entrez deux entiers");
        a=clavier.lireEntier();
        b=clavier.lireEntier();

        if (a>b)
        {
            vi=a;
            a=b;
            b=vi;
        }
        System.out.println(a+" "+b);
    }
}
```

```
    } // fin de la partie "main"  
}
```

2.1.2 Conditionnelle à deux possibilités (alternative)

Syntaxe Algorithmique.

```
si  $C$  alors  
   $A_1; A_2; \dots; A_n$   
sinon  
   $B_1; B_2; \dots; B_n$   
fsi
```

où C est une condition,
 $A_1; A_2; \dots; A_n$, $B_1; B_2; \dots; B_n$ sont des instructions

Syntaxe Java.

```
if  $C$   
  {  $A_1; A_2; \dots; A_n$ ; }  
else  
  {  $B_1; B_2; \dots; B_n$ ; }
```

```
if  $C$   
  A;  
else  
  B;
```

Algorithme *EcrireOrdre2*

Données : *deux entiers lus au clavier*

Résultats : *affichage à l'écran du plus petit puis du plus grand de ces nombres*

Principe : Les deux nombres sont placés dans deux variables *a* et *b*. L'ordre entre *a* et *b* détermine l'ordre d'affichage.

Objets utilisés	Instructions
entier <i>a, b</i> les entiers lus	écrire écran "entrez deux entiers" lire clavier <i>a, b</i> si $a > b$ alors écrire écran <i>b, ' ', a, '\n'</i> sinon écrire écran <i>a, ' ', b, '\n'</i> fsi

```
package Prog.MesExemples;
import Prog.MesEntreesSorties.ConsoleStandard;
import java.io.*;

public class EcrireOrdre2
{
    public static void main(String argv[]) throws IOException
    {
        // variables locales
        int a,b;
        ConsoleStandard clavier = new ConsoleStandard();

        System.out.println("Entrez deux entiers");
        a=clavier.lireEntier();
        b=clavier.lireEntier();

        if (a>b)
            System.out.println(b+" "+a);
        else
            System.out.println(a+" "+b);

    } // fin de la partie "main"
} // fin de la classe
```

2.1.3 Conditionnelle à possibilités multiples

Syntaxe algorithmique.

```
cas où  $E$  vaut  
   $v_1 : A_{11}; A_{12}; \dots; A_{1n_1}$   
   $v_2 : A_{21}; A_{22}; \dots; A_{2n_2}$   
  .....  
   $v_m : A_{m1}; A_{m2}; \dots; A_{mn_m}$   
  autrement :  $B_1; B_2; \dots; B_n$   
f cas
```

où E est une variable entière ou caractère,
 $v_1; \dots; v_m$ sont des constantes distinctes du même type que
 E
 $A_{11}; \dots; A_{mn_m}; B_1; \dots; B_n$ sont des instructions

Syntaxe Java.

```
switch( $E$ )  
{  
  case  $v_1 : A_{11}; A_{12}; \dots; A_{1n_1};$  break;  
  case  $v_2 : A_{21}; A_{22}; \dots; A_{2n_2};$  break;  
  .....  
  case  $v_m : A_{m1}; A_{m2}; \dots; A_{mn_m};$  break;  
  default :  $B_1; B_2; \dots; B_n;$   
}
```

où E est une variable entière (mais pas long) ou caractère,

La clause `default` : est facultative.

L'exemple suivant montre comment traiter la forme abrégée dans laquelle une même suite d'instructions peut être exécutée pour différentes valeurs de E .

Algorithme *ReconnaitVoyelles*

Données : *un caractère lu au clavier*

Résultats : *affiche un message différenciant voyelles minuscules et voyelles majuscules des autres caractères*

Objets utilisés	Instructions
caractère <i>c</i> lu au clavier	écrire écran "Entrez un caractère" lire clavier <i>c</i> cas où <i>c</i> vaut 'a','e','i','o','u','y' : écrire écran "voyelle minuscule" 'A','E','I','O','U','Y' : écrire écran "voyelle majuscule" autrement : écrire écran "ce n'est pas une voyelle!" fcas

```
package Prog.MesExemples;
import Prog.MesEntreesSorties.ConsoleStandard;
import java.io.*;
public class ReconnaitVoyelles
{
    public static void main(String argv[]) throws IOException
    {
        char c;
        ConsoleStandard clavier = new ConsoleStandard();
        System.out.println("Entrez un caractère");
        c = clavier.lireCaractere();
        switch(c)
        {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
            case 'y':
                System.out.println("voyelle minuscule"); break;
            case 'A':
            case 'E':
            case 'I':
            case 'O':
            case 'U':
```

```

        case 'Y':
            System.out.println("voyelle majuscule"); break;
        default:
            System.out.println("ce n'est pas une voyelle !");
    }
} // fin de la partie "main"
} // fin de la classe

```

2.2 Complétons les classes par des méthodes

2.2.1 Généralités sur la notion de méthode

Nous avons vu au chapitre précédent la partie structurelle des classes (comment décrire leurs attributs). Nous nous intéressons maintenant à la manière d'écrire les opérations des classes. Les opérations s'implémentent sous la forme de « méthodes », qui sont des sortes de fonctions telles que nous les avons vues en algorithmique, avec trois aspects essentiels.

- Une méthode s'applique à une instance, et possède des paramètres et un type de retour là où la fonction ne dispose que de paramètres et d'un type de retour. Comme elle s'applique à une instance, elle peut travailler (manipuler) les attributs de l'instance sous réserve des mécanismes de protection dont nous parlons plus loin.
- Des classes différentes peuvent proposer des méthodes portant le même nom (ex : `affiche()`). La sélection de la méthode à appeler dépend donc du contexte, en l'occurrence du type de l'objet sur lequel on l'applique.
- Dans une classe, plusieurs méthodes peuvent porter le même nom, à condition que leurs liste de paramètres soit différente (en nombre ou en types). Dans ce cas, les paramètres réels permettent de déterminer la méthode à appeler.

L'exemple ci-dessous met en parallèle les deux notions de fonction et de méthode sur des exemples dans une notation algorithmique. On suppose l'existence d'un type `Personne`, classe qui dispose d'un attribut `age`.

Comparaison des appels pour <code>pers1</code> qui désigne une valeur de <code>Personne</code>	
<i>Fonction</i> pers1 est un paramètre	<i>Méthode</i> on applique l'opération à l'instance
<code>estMajeure(pers1)</code>	<code>pers1.estMajeure()</code>

```
booléen fonction estMajeure(Personne p)
  paramètres E :   Personne p
  valeur renvoyée : booléen, vrai ssi p a plus de 18 ans
```

```
retourner (p.getAge() > 18)
```

```
booléen méthode estMajeure()
  paramètres E :
  Objet receveur : une instance de la classe Personne
  valeur renvoyée : booléen, vrai ssi le receveur a plus de 18 ans
```

```
retourner (getAge() > 18)
```

On voit que la différence essentielle porte sur le fait que la version méthode, s'appliquant à une instance,

- n'attend pas cette instance comme paramètre,
- peut manipuler directement les attributs de l'instance, sans avoir besoin de les préfixer par une désignation d'instance. Cependant, si on souhaite expliciter l'objet receveur, on peut le faire, grâce à une variable spéciale qui désigne, pendant l'exécution de la méthode, l'instance sur laquelle on a appliqué la méthode. En algorithmique, nous noterons cette variable spéciale **receveur**.

Nous pouvons ainsi réécrire la méthode précédente comme suit.

```
booléen méthode estMajeure()
  paramètres E :
  Objet receveur : une instance de la classe Personne
  valeur renvoyée : booléen, vrai ssi le receveur a plus de 18 ans
```

```
retourner (receveur.getAge() > 18)
```

2.2.1.1 Définition d'une méthode en Java

Une méthode se définit à l'intérieur de la classe. Elle dispose d'une entête et d'un corps.

Attention! En Java, il n'y a qu'un seul mode de passage des paramètres.

Tous les paramètres sont passés par valeur.

On ne peut donc modifier aucun paramètre à l'intérieur d'une méthode.

S'il s'agit d'une instance, cela signifie que l'on ne peut pas modifier son identité, sa désignation, **mais cela n'empêche pas que l'on peut changer son état.**

La syntaxe la plus simple est la suivante (nous ajouterons plus tard des informations).

```
typeRetour  nomMéthode(typeParam1  identParam1,  ...  typeParamn
identParamn)
{
.....
}
```

où *typeRetour* est un type, qui peut être `void` (pour vide),

nomMéthode est un identificateur (par convention commence par une minuscule),

*typeParam*_{*i*} est un type,

*identParam*_{*i*} est l'identificateur d'un paramètre formel, qui ne doit pas être modifié dans le corps de la méthode.

Pour écrire le corps de la méthode :

- les variables utilisables sont les paramètres, les variables locales s'il y en a, et les attributs de l'objet receveur,
- *receveur* se dit `this`
- *retourner* s'écrit `return` en Java

Nous complétons la classe `Personne` pour vous montrer quelques exemples de méthodes, avec ou sans type de retour, avec ou sans variable locale, avec ou sans paramètre. Dans la dernière méthode, nous illustrons l'utilisation de la variable spéciale `this`.

```
package Prog.LaVraieVie;
public class Personne
{
// attributs

    private String nom;
    private int age;
    private Voiture maVoiture;

// accesseurs
    public int getAge()
```

```

    {
        return age;
    }

// méthodes

public void saisirAge() throws IOException
{
    System.out.println("Veuillez entrer l'age de (d') "+nom);
    ConsoleStandard clavier = new ConsoleStandard();
    age = clavier.lireEntier();
}

public int anneeNaissance()
{
    int anneeCourante = Calendar.getInstance().get(Calendar.YEAR);
    return (anneeCourante - age);
}

public boolean estMajeure()
{
    return Age > 18;
}

boolean aMemeAge(Personne autrePers)
{
    return age == autrePers.getAge();

    // variante
    // return this.age == autrePers.getAge();
}
}

```

2.2.1.2 Appel d'une méthode

La syntaxe ressemble fortement à celle de l'accès à un attribut.

```
i.m(E1, ... En)
```

où *i* est une instance de la classe *C* qui dispose d'une méthode *m*,
E1, ... *En* sont des expressions, ce sont les paramètres réels,
m dispose de paramètres formels compatibles en nombre, type et ordre avec les paramètres réels.

Le programme ci-dessous illustre l'appel de méthode.

```
package Prog.LaVraieVie;
import Prog.MesEntreesSorties.ConsoleStandard;
import java.io.*;
public class ProgrammePersonne
{
    public static void main(String argv[]) throws IOException
    {
        Personne pers1 = new Personne();
        pers1.setNom("Elisa");
        pers1.saisirAge();
        System.out.println("annee de naissance de (d') "+pers1.getNom()
            +" = "+pers1.anneeNaissance());
        System.out.println(pers1.getNom()+" est-elle majeure ? "+pers1.estMajeure());

        Personne pers2 = new Personne();
        pers2.setNom("Eloise");
        pers2.saisirAge();
        System.out.println(pers1.getNom()+" et "+pers2.getNom()+" ont le même âge ? "+
            pers1.aMemeAge(pers2));
    }
}
```

2.2.1.3 Troublantes variables

Comme nous l'avons dit, les paramètres sont passés par valeur, donc vous ne devez pas modifier la valeur des paramètres ... Mais c'est sous votre responsabilité!
A titre d'exercice, analysez la méthode et le programme suivant.

```
public class Personne
{
    .....

    public void FauxEchange(int a, int b)
    {
```

```

        System.out.println("a = "+a+" b = "+b);
        int vi = a;
        a = b;
        b = vi;
        System.out.println("a = "+a+" b = "+b);
    }
}

// dans le main de ProgrammePersonne .....
int x=2, y=3;
System.out.println("x = "+x+" y = "+y);
pers2.FauxEchange(x,y);
System.out.println("x = "+x+" y = "+y);

```

Le résultat de l'exécution figure ci-dessous.

```

x = 2 y = 3
a = 2 b = 3
a = 3 b = 2
x = 2 y = 3

```

2.2.2 Encapsulation

Dans ce paragraphe, nous étudions les mécanismes que Java propose pour réaliser « l'encapsulation » des objets, qui consiste à interdire l'accès ou la connaissance de leur implémentation. Avec une déclaration de classe telle que :

```

package Prog.LaVraieVie;
public class Personne
{
    String nom;
    int age;
    Voiture maVoiture;

    .....
}

```

Les attributs sont accessibles par tout autre programme du même paquetage. On peut ainsi connaître leur valeur, la manière dont elle est codée, et la modifier sans aucun contrôle. Ce peut être ennuyeux dans de nombreux cas.

Prenons l'exemple d'une classe `Date`. Une date peut être codée de diverses manières : six entiers (année, mois, jour, heure, minute, seconde), un tableau de six entiers, un seul entier donnant le nombre de secondes correspondant à cette date depuis une date origine des temps, une chaîne de caractères contenant année, mois, ... seconde séparées par des caractères spéciaux, etc. L'utilisateur de la classe n'a aucun intérêt à connaître cette implémentation (ce codage), ce qui l'intéresse, c'est essentiellement d'utiliser les objets de la classe pour créer une date de diverses manières, et la connaître sous différentes formes.

- Si l'on est amené à changer le codage de la classe `Date` il est préférable de modifier aussi peu que possible les programmes qui l'utilisent. C'est justement l'un des intérêts de l'approche objet de promouvoir le développement séparé des différentes classes qui composent un logiciel, et une manière de le faire est d'éviter de s'appuyer sur l'implémentation (le codage, les attributs).
- Par ailleurs, une date ne se crée pas n'importe comment, tout entier ne peut être une valeur correcte pour un mois. On ne peut pas facilement contrôler ce genre de problème lorsqu'on effectue une affectation telle que `d.mois = 43`, où `d` serait une instance de `Date` et `mois` un de ses attributs.
- Enfin, on peut désirer connaître le nombre de fois où on accède à une date, ce qui ne peut pas être fait dans une expression d'accès telle que `d.jour`, où `d` serait une instance de `Date` et `jour` un de ses attributs.

Tout ceci, plus d'autres problèmes que vous découvrirez avec l'héritage, fait qu'il est important de cacher l'implémentation, en particulier les choix de codage des attributs.

Pour masquer les attributs (et ce sera applicable aussi aux opérations), on utilise des directives de protection. Ce sont des mots-clés de Java, tels que `public` et `private`, que l'on place devant les attributs ou les opérations, pour signifier qu'ils sont accessibles par toute autre classe (dans le cas de `public`), ou seulement par le code de la classe qui les déclare (dans le cas de `private`). Par exemple, pour l'extrait de classe suivant, l'attribut `nom` est `public`, il peut être utilisé dans les opérations de `Personne`, dans un autre programme `ProgrammePersonne` du même paquetage, ainsi que dans un programme `TestPersonne` d'un autre paquetage.

L'attribut `age` est privé, il ne peut être utilisé que dans les opérations de `Personne`.

L'attribut `maVoiture`, pour lequel on n'a pas écrit de directive de protection est visible depuis tout le paquetage, il peut être utilisé dans les opérations de `Personne`, mais aussi de toute autre classe (programme) du paquetage.

```
package Prog.LaVraieVie;
public class Personne
{
    public String nom;
    private int age;
        Voiture maVoiture;

    void uneOperationDePersonne()
    {
        Personne p = new Personne();
        System.out.println(p.nom + p.age + p.maVoiture);
    }
}
```

Le programme ci-dessous est dans le même paquetage que la classe `Personne`.

```
package Prog.LaVraieVie;
import Prog.MesEntreesSorties.ConsoleStandard;
```

```

import java.io.*;

public class ProgrammePersonne
{
    public static void main(String argv[]) throws IOException
    {
        Personne p = new Personne();
        System.out.println(p.nom + p.maVoiture);
        p.uneOperationDePersonne();
    } // fin de la partie "main"
} // fin de la classe

```

Celui-ci est par contre dans un paquetage différent.

```

package Prog.MesExemples;
import Prog.LaVraieVie.*;
import java.io.*;

public class TestPersonne
{
    public static void main(String argv[]) throws IOException
    {
        Personne p = new Personne();
        System.out.println(p.nom);
    }
}

```

La figure 2.1 schématise ces différents aspects : les propriétés barrées d'une croix sont inaccessibles.

Pour clore ce paragraphe, une règle que nous nous imposerons dans le cadre de ce cours : les attributs seront toujours privés, et les méthodes généralement publiques (mais nous verrons des variantes sur ce deuxième point). Pour accéder aux attributs, on utilisera donc des méthodes particulières, que l'on appelle communément des « accesseurs », qui sont décrites plus loin.

Les classes peuvent être affectées de deux niveaux de protection : **public**, en utilisant le mot clef, et **package**, lorsqu'on ne précise rien. Une contrainte à noter : on ne peut placer qu'une seule classe publique dans un fichier.

2.3 Méthodes spéciales

2.3.1 Constructeurs

Nous avons fait quelques initialisations des objets soit lors de la déclaration des attributs, soit dans le corps d'un programme. Il y a une bonne manière de regrouper ces initialisations, qui consiste à définir une méthode spéciale que l'on appelle un *constructeur*.

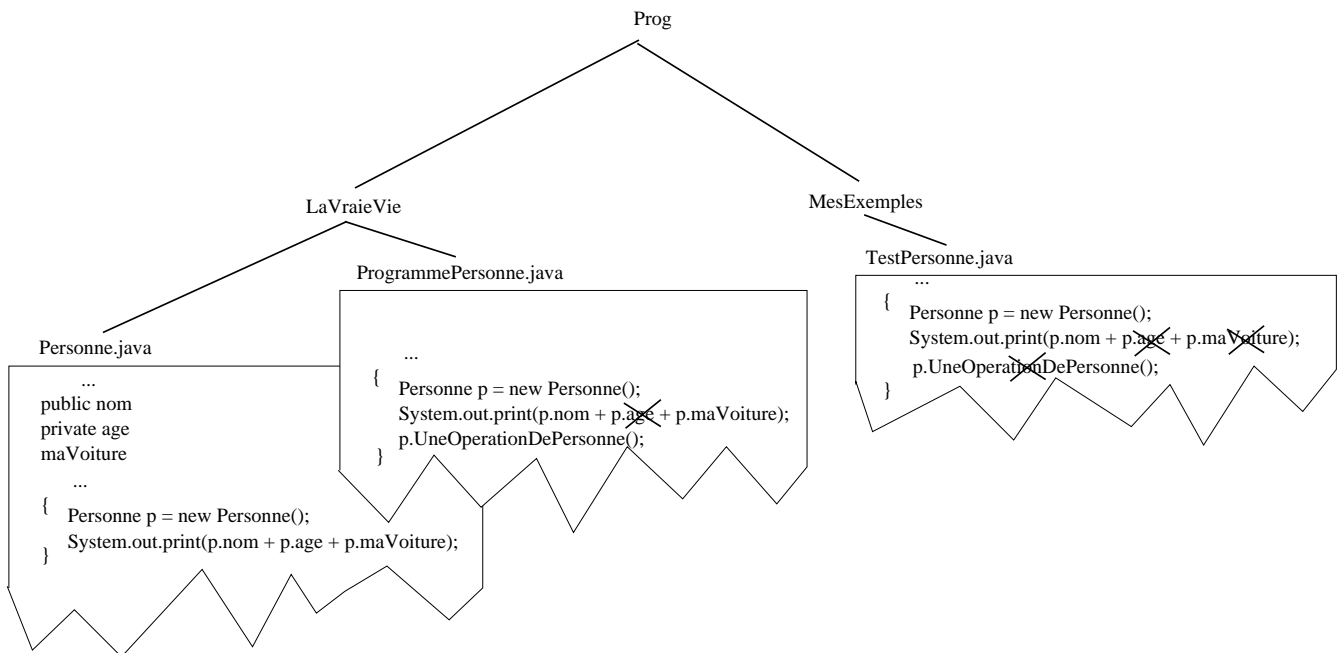


FIG. 2.1 – Protection.

Le constructeur **porte le même nom que la classe**. Il n'a pas de type de retour, par contre il admet une liste de paramètres identique à celle d'une méthode normale.

Une règle à retenir est que, dans le cas général, il faut toujours prévoir un constructeur sans paramètres. Il permet de définir ce qu'est un objet « par défaut » bien construit.

On peut mettre autant de constructeurs que l'on veut. Par exemple, pour la classe **Personne**, on pourrait proposer les constructeurs ci-dessous.

```

public class Personne
{
    private String nom;
    private int age;
    private Voiture maVoiture;

    public Personne()
    {nom = "inconnu"; age = -1;}

    public Personne(String n)
    {nom = n; age = -1;}

    public Personne(String n, int a)
    {nom = n; age = a;}

    public Personne(String n, Voiture v)

```

```

        {nom = n; age = -1; maVoiture = v;}
    .....
}

```

Le constructeur est appelé lors de l'évaluation d'une expression de la forme suivante (qui généralise celle que nous avons vue dans le cours précédent).

```

new C(E1, ... En)

```

où C est une classe, E1, ... En sont des expressions.

Lorsqu'une instance est créée par une expression `new`, Java recherche et applique à l'instance le constructeur de la classe C dont les paramètres formels correspondent en nombre, ordre et types aux paramètres réels E1, ... En.

Par exemple, on peut maintenant créer et initialiser les instances de la classe `Personne` de différentes manières. Dans le programme suivant,

- lorsque `pers1` sera créée, le constructeur sans paramètres lui sera appliqué,
- lorsque `pers2` sera créée, le constructeur qui a un paramètre lui sera appliqué,
- lorsque `pers3` sera créée, le constructeur qui a deux paramètres, et dont le deuxième paramètre est un entier, lui sera appliqué.

```

package Prog.MesExemples;
import Prog.MesEntreesSorties.ConsoleStandard;
import java.io.*;

public class ProgrammePersonne
{
    public static void main(String argv[]) throws IOException
    {
        Personne pers1 = new Personne();

        Personne pers2 = new Personne("Louis");

        Personne pers3 = new Personne("Eloise",2);
    }
}

```

2.3.2 Accesseurs

Les « accesseurs » sont des méthodes qui permettent d'accéder aux attributs pour connaître leur valeur ou les modifier. En Java, ils sont réglés par des conventions de nommage.

La signature d'un accesseur permettant de connaître la valeur d'un attribut `attrX` de type `T` est la suivante :

```
T getAttX()
```

La signature d'un accesseur permettant d'affecter une valeur à un attribut `attrX` de type `T` est la suivante :

```
void setAttX(T valeurDeT)
```

Par exemple, pour l'attribut `nom` de la classe `Personne`, on introduirait les deux accesseurs suivants.

```
public class Personne
{
    private String nom;

    .....

    public String getNom()
    {return nom;}

    public void setNom(String n)
    {nom = n;}

    .....
}
```

Tous les attributs ne sont pas forcément munis d'accesseurs (ils ne doivent pas être connus des utilisateurs de la classe), certains peuvent n'avoir que l'un des deux types d'accesseurs (par exemple, on peut connaître la valeur d'un attribut, mais jamais la modifier directement). Par ailleurs cette notion d'accesseur est relativement ambiguë, puisque d'une certaine manière l'utiliser de manière stricte dévoile ce qui est attribut et ce qui ne l'est pas. Par exemple, pour la classe `Personne`, qu'est-ce qui différencie pour un utilisateur de la classe les méthodes `getAge` et `annéeNaissance`? Rien, seule l'implémentation change : `getAge` retourne la valeur d'un attribut (`age`), tandis que l'année de naissance est calculée, mais la classe pourrait très bien être implémentée à l'aide d'un attribut `annéeNaissance`, et dans ce cas l'âge serait calculé. En conclusion, il ne faut surtout pas faire d'extrémisme dans leur définition! Par contre, il est important de les utiliser au maximum, y compris dans la définition des autres méthodes de la classe (ainsi, même si l'implémentation change, ces méthodes devraient toujours exister).

2.3.3 La méthode toString

Autre convention de Java, toutes les classes possèdent 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.

En particulier, cette méthode est appelée (eh oui !) lorsque l'on affiche au moyen de `System.out.print`. Si vous ne la définissez pas dans votre classe, elle existe par défaut et retourne une désignation de l'instance.

Il est conseillé d'en écrire une pour chaque classe que vous concevez. Par exemple, pour la classe `Personne` nous pourrions proposer de retourner une chaîne contenant le nom d'une personne suivi de son âge.

```
public class Personne
{
    private String nom;
    private int age;
    .....
    public String toString()
    {return nom+' '+age;}
    .....
}
```

En appliquant la règle qui consiste à utiliser le plus possible les accesseurs, on pourrait aussi écrire la méthode ainsi :

```
public class Personne
{
    private String nom;
    private int age;
    .....
    public String toString()
    {return getNom()+' '+getAge();}
    .....
}
```

2.4 La classe Personne complète

Voici la classe intégrant tout ce que nous avons appris jusqu'ici.

```
package Prog.LaVraieVie;
import java.io.*;
import java.util.*;
import Prog.MesEntreesSorties.ConsoleStandard;
```

```

public class Personne
{
    // ----- attributs -----

    private String nom;
    private int age;
    private Voiture maVoiture;

    // ----- constructeurs -----

    public Personne()
    {nom = "inconnu"; age = -1;}

    public Personne(String n)
    {nom = n; age = -1;}

    public Personne(String n, int a)
    {nom = n; age = a;}

    public Personne(String n, Voiture v)
    {nom = n; age = -1; maVoiture = v;}

    // ----- accesseurs -----

    public String getNom()
    {return nom;}

    public void setNom(String n)
    {nom = n;}

    public int getAge()
    {return age;}

    public void setAge(int a)
    {age = a;}

    public Voiture getVoiture()
    {return maVoiture;}

    public void setVoiture(Voiture v)
    {maVoiture = v;}

    // ----- toString -----

```

```

public String toString()
{return nom+' '+age;}

// ----- autres méthodes -----

public void saisirAge() throws IOException
{
    System.out.println("Veuillez entrer l'age de (d') "+nom);
    ConsoleStandard clavier = new ConsoleStandard();
    age = clavier.lireEntier();
}

public int anneeNaissance()
{
    int anneeCourante = Calendar.getInstance().get(Calendar.YEAR);
    return (anneeCourante - age);
}

public boolean estMajeure()
{
    return age > 18;
}

public boolean aMemeAge(Personne autrePers)
{
    return age == autrePers.age;
}
}

```

Un programme illustrant son utilisation dans l'état actuel. On remarque que plus aucun attribut n'est utilisé directement.

```

package Prog.LaVraieVie;
import Prog.MesEntreesSorties.ConsoleStandard;
import java.io.*;

public class ProgrammePersonne
{
    public static void main(String argv[]) throws IOException
    {
        Personne pers1 = new Personne();
        pers1.setNom("Elisa");
        pers1.saisirAge();
    }
}

```

```

System.out.println("annee de naissance de (d') "+pers1.getNom()
                  +" = "+pers1.anneeNaissance());
System.out.println(pers1.getNom()+" est-elle majeure ? "+pers1.estMajeure());

Personne pers2 = new Personne("Louis");
pers2.saisirAge();
System.out.println(pers1.getNom()+" et "+pers2.getNom()+" ont le même âge ? "+
                  pers1.aMemeAge(pers2));

Personne pers3 = new Personne("Eloise",2);

System.out.println(pers1+" "+pers2+" "+pers3);

} // fin de la partie "main"
} // fin de la classe

```

Une exécution de ce programme pourrait être :

```

Veillez entrer l'age de (d') Elisa
2
annee de naissance de (d') Elisa = 1998
Elisa est-elle majeure ? false
Veillez entrer l'age de (d') Louis
1
Elisa et Louis ont le même âge ? false
Elisa 2 Louis 1 Eloise 2

```

Cours 3

Répétitives - Tableaux - Vecteurs

3.1 Instructions répétitives

3.1.1 Répétition contrôlée par un compteur

Syntaxe algorithmique.

```
pour ( $vc \leftarrow E; C; I$ ) faire  
   $A_1; A_2; \dots; A_n$   
fpour
```

où vc est une variable entière, appelée variable de contrôle ou compteur,
 E est une expression entière qui donne la valeur initiale de vc ,
 C est une condition portant sur la valeur de vc ,
 I est une instruction « de pas » dont le rôle est d'incrémenter ou de décrémente le compteur,
 $A_1; A_2; \dots; A_n$ sont des instructions.

Syntaxe Java.

```
for ( $vc = E; C; I$ )  
  { $A_1; A_2; \dots; A_n$ ;
```

```
for ( $vc = E; C; I$ )  
  A;
```

Algorithme *PuissanceEntière1*

Données : deux entiers $x > 0, y \geq 0$, lus au clavier

Résultats : affiche à l'écran x^y

Principe : x^0 vaut 1, x^y vaut $1 * x * x * x \dots * x$ (y occurrences de x)

Objets utilisés	Instructions
entiers x, y lus au clavier	écrire écran "Entrez un entier $x > 0$ " lire clavier x
entier res vaut x^v après l'étape v	écrire écran "Entrez un entier $y \geq 0$ " lire clavier y $res \leftarrow 1$ pour ($v \leftarrow 1; v \leq y; v \leftarrow v+1$) faire $res \leftarrow res * x$
entier v var. de contrôle	fpour écrire écran " x^y vaut ", res

```
package Prog.MesExemples;
import Prog.MesEntreesSorties.ConsoleStandard;
public class PuissanceEntiere1
{
    public static void main(String argv[]) throws java.io.IOException
    {
        int x, y, res, v;
        ConsoleStandard clavier = new ConsoleStandard();

        System.out.println("entrez un entier > 0 ");
        x = clavier.lireEntier();
        System.out.println("entrez un entier >= 0 ");
        y = clavier.lireEntier();

        res = 1;
        for (v = 1; v <= y; v = v + 1)
            res = res * x;

        System.out.println(x+"^"+y+" = "+res);
    } // fin de la partie "main"
} // fin de la classe
```

3.1.2 Répétition contrôlée par une condition (forme 1)

Syntaxe algorithmique.

```
tant que  $C$  faire  
   $A_1; A_2; \dots; A_n$   
ftant
```

où C est une condition,
 $A_1; A_2; \dots; A_n$ sont des instructions

Syntaxe Java.

```
while ( $C$ )  
  { $A_1; A_2; \dots; A_n;$ }
```

```
while ( $C$ )  
  A;
```

Algorithme *PuissanceEntière2*

Données : deux entiers $x > 0, y \geq 0$, lus au clavier

Résultats : affiche à l'écran x^y

Principe : x^0 vaut 1, x^y vaut $1 * x * x * x \dots * x$ (y occurrences de x)

Objets utilisés	Instructions	Actions non-élémentaires
entier x,y lus au clavier	écrire écran "Entrez un entier x > 0" lire clavier x	
entier <i>res</i> vaut x^i à l'étape i	écrire écran "Entrez un entier $y \geq 0$ " lire clavier y $res \leftarrow 1$ $v \leftarrow 1$ tant que ($v \leq y$) faire $res \leftarrow res * x$	
entier <i>v</i> var. de contrôle	$v \leftarrow v + 1$ ftant écrire écran " x^y vaut ", res	

```

package Prog.PolyProg2000;
import Prog.MesEntreesSorties.ConsoleStandard;
public class Repetitives
{
    public static void main(String argv[]) throws java.io.IOException
    {
        int x, y, res, v;

        System.out.println("entrez un entier > 0 ");
        x = clavier.lireEntier();
        System.out.println("entrez un entier >= 0 ");
        y = clavier.lireEntier();

        res = 1;
        v = 1;
        while (v <= y)
        {
            res = res * x;
            v = v + 1;
        }
    }
}

```

```
        System.out.println(x+"^"+y+" = "+res);  
    } // fin de la partie "main"  
} // fin de la classe
```

3.1.3 Répétition contrôlée par une condition (forme 2)

Cette forme permet d'effectuer au moins une fois la séquence d'instructions avant de tester la condition.

Syntaxe algorithmique.

<pre>faire A₁; A₂; ...; A_n tant que C</pre> <p>où C est une condition, $A_1; A_2; \dots; A_n$ sont des instructions</p>
--

Syntaxe Java.

<pre>do {A₁; A₂; ...; A_n;} while (C);</pre>	<pre>do A; while (C);</pre>
--	-------------------------------

Dans l'algorithme précédent, si nous voulons être certains d'avoir saisi un entier strictement positif pour initialiser x , nous pouvons remplacer les instructions :

```
écrire écran "Entrez un entier x>0"
lire clavier x
```

par les instructions suivantes :

<pre>entier x</pre>	<pre>faire écrire écran "Entrez un entier x > 0" lire clavier x tant que (x ≤ 0)</pre>	
---------------------	---	--

Ce qui se traduira en Java par :

```
do
{
  System.out.println("entrez un entier > 0 ");
```

```

        x = clavier.lireEntier();
    }
    while (x <= 0);

```

3.2 Tableaux

Java dispose des types tableaux, qui sont, comme en algorithmique, des collections de valeurs d'un même type, ordonnées et indexées. Les valeurs placées dans un tableau peuvent être des valeurs d'un type primitif (entier, caractère, réel, booléen) ou des instances.

Déclaration d'une variable de type tableau

Tableau de T monTableau	T[] monTableau ;
où T est un type	

Vous pouvez noter que la déclaration ne précise pas de taille : ce n'est pas nécessaire puisqu'à ce stade aucun tableau n'existe, on a seulement déclaré une désignation de tableau.

Création d'un tableau

<pre>monTableau = new T[n] ;</pre> <p>où monTableau est une variable de type T[]</p>
--

Les éléments du tableau sont numérotés de 0 à n-1. Ils sont initialisés avec une valeur par défaut qui dépend du type (comme les attributs). La taille du tableau ne peut plus être modifiée après la création. Si on a besoin de plus d'espace, il faut créer un autre tableau plus grand et recopier les valeurs du premier dans le second.

Accès à l'une des valeurs d'un tableau

<pre>monTableau[i]</pre> <p>où monTableau est une variable de type T[n], et i est un entier compris entre 0 et n. Cette expression est de type T.</p>

Longueur d'un tableau La longueur d'un tableau peut être obtenue en utilisant l'expression suivante, qui est de type entier.

```
monTableau.length
```

Un exemple de tableau d'entiers Nous illustrons les tableaux en reprenant tout d'abord l'algorithme d'inversion d'un tableau.

Algorithme InversionTableau

But : Inverser l'ordre des valeurs d'un tableau

Tableau de 6 entiers T

entier l : longueur du tableau.

entier g : indice de la première case à permuter.

entier d : indice de la dernière case à permuter.

entier i : compteur avançant tant qu'on a des cases à permuter.

$l \leftarrow \text{Taille}(T)$

$g \leftarrow 0$

$d \leftarrow l - 1$

pour ($i \leftarrow 1$; $i \leq l/2$; $i \leftarrow i + 1$)

faire

 Permute($T[g], T[d]$)

$g \leftarrow g + 1$

$d \leftarrow d - 1$

fpour

Permute(A,B) :
échange entre elles
les valeurs de A et
B.

Nous l'écrivons en Java sans faire d'appel à une fonction d'échange de valeurs qui pose dans ce langage deux problèmes particuliers (pas de passage de paramètres sortie / nécessité de définir cette fonction comme une méthode).

Nous ajoutons au début du programme une saisie des valeurs du tableau.

```
package Prog.MesExemples;
import Prog.MesEntreesSorties.*;
import java.io.*;

public class InverseTableaux
{
    public static void main(String[] arg) throws IOException
    {
        ConsoleStandard clavier = new ConsoleStandard();
        int[] T = new int[5];
        int l,g,d,i,aux;
        l = T.length;

        for (i = 0; i<l; i = i+1)
        {
            System.out.print("Entrez la valeur de T["+i+"] ");
```

```

        T[i]=clavier.lireEntier();
    }

    System.out.println("avant inversion T vaut");

    for (i = 0; i<l; i = i+1)
    {
        System.out.print("T["+i+"] = "+T[i]+" ");
    }

    g = 0;
    d = l-1;
    for (i = 1; i<= l/2; i = i+1)
    {
        aux=T[g]; T[g]=T[d]; T[d]=aux;
        g = g+1; d = d-1;
    }

    System.out.println("\n après inversion T vaut ");
    for (i = 0; i<l; i = i+1)
    {
        System.out.print("T["+i+"] = "+T[i]+" ");
    }
    System.out.println();
}
}

```

La figure 3.1 visualise le tableau (la variable qui le désigne et les valeurs qu'il peut contenir).



FIG. 3.1 – Un tableau d'entiers.

Un exemple de tableau de Voyageurs Comme deuxième exemple, nous écrivons un programme qui illustre les tableaux d'objets.

Tout d'abord voici la définition de la classe `Voyageur`.

```

package Prog.Voyage;
import Prog.MesEntreesSorties.ConsoleStandard;
public class Voyageur
{

```

```

// -----ATTRIBUTS-----

private String nom;
private int age; /* entre 0 et 120, vaut 0 par défaut */
private char sexe; /* 'M' ou 'F', vaut 'F' par défaut */

// -----CONSTRUCTEURS-----

public Voyageur()
{nom="inconnu"; age = 0; sexe = 'F';}

public Voyageur(int a, char s)
{nom="inconnu"; age = a; sexe = s;}

public Voyageur(String n, int an)
{nom = n; age = an; sexe = 'F';}

// -----ACCESSEURS-----

public void setNom(String n){nom=n;}
public String getNom(){return nom;}

public void setAge(int a)
{
    if (a>=0 && age <=120) age=a;
    else {age=0; System.out.println(""+a+"est une valeur incorrecte pour l'age");}
}

public int getAge(){return age;}

public void setSexe(char s)
{
    if (s=='M' || s=='F') sexe = s;
    else {sexe='F'; System.out.println(""+s+"est une valeur incorrecte pour le sexe");}
}

public char getSexe(){return sexe;}

// -----METHODES-----

public void saisie(ConsoleStandard cons) throws java.io.IOException
{
    System.out.println("Saisie des données d'un voyageur \nEntrer son nom : ");
    nom = cons.lireChaine();
}

```

```

        System.out.println("Entrer son age : ");
        age = cons.lireEntier();
        System.out.println("Entrer un caractère pour le sexe (M ou F) : ");
        sexe = cons.lireCaractere();
    }

    public boolean carteVermeil()
    {return ((age >= 65) || ((age >= 60) && (sexe == 'F')));}
}

```

Enfin voici un programme qui crée, remplit et utilise un tableau d'instances de la classe Voyageur.

```

package Prog.Voyage;
import Prog.MesEntreesSorties.ConsoleStandard;

public class AppliVoyageurTableau
{
    public static void main(String[] argv) throws java.io.IOException
    {
        ConsoleStandard cons = new ConsoleStandard();

        // Déclaration d'un tableau de voyageurs
        Voyageur[] tabVoyageurs;

        // Création du tableau (aucun voyageur n'existe encore)
        System.out.println("Combien de voyageurs désirez-vous créer ?");
        int nombreVoyageurs = cons.lireEntier();
        tabVoyageurs = new Voyageur[nombreVoyageurs];

        // Création et saisie des données des voyageurs
        for (int i=0; i<tabVoyageurs.length; i=i+1)
        {
            Voyageur v = new Voyageur();
            v.saisie(cons);
            tabVoyageurs[i] = v;
        }

        //Affichage des informations relatives à la carte vermeil
        for (int i=0; i<tabVoyageurs.length; i=i+1)
        {
            System.out.println(tabVoyageurs[i].getNom()+" a droit a la carte vermeil ? "

```

```

+tabVoyageurs[i].carteVermeil()+"\n");
}
} //fin MAIN
} //fin CLASSE

```

La figure 3.2 visualise le tableau (la variable qui le désigne et les valeurs qu'il peut contenir).

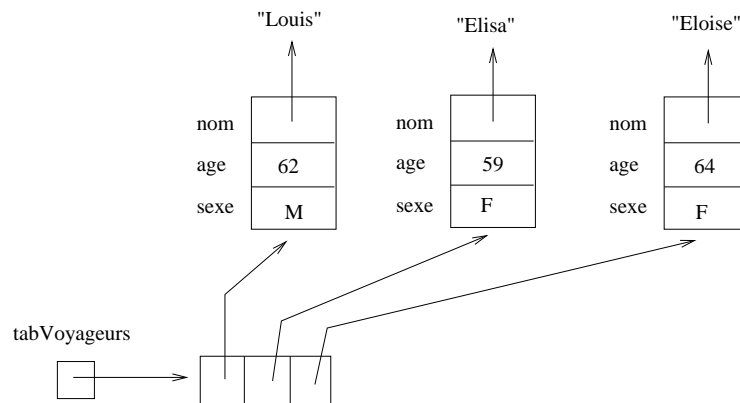


FIG. 3.2 – Un tableau de voyageurs.

Pour aller plus loin.

Nous citons trois aspects des tableaux que nous ne détaillons pas dans ce cours, reportez-vous à la documentation pour plus de précisions.

- Java permet de créer des tableaux multidimensionnels tels qu'ils vous ont été décrits en algorithmique.
- Pour copier un tableau, on doit utiliser la méthode `clone`.
- Une classe de Java (`Arrays`) propose diverses méthodes utiles sur les tableaux telles que le tri, la recherche d'éléments ou encore le garnissage conjoint de plusieurs éléments avec une valeur donnée. La classe `System`, quant à elle, contient une méthode `arraycopy` qui permet de copier une partie des valeurs d'un tableau dans un autre.

3.3 La classe Vector

Les tableaux sont utiles, mais présentent un certain nombre d'inconvénients.

- Une fois le tableau créé (avec l'opérateur `new`), la taille ne peut plus être changée. Or dans de nombreuses situations, on ne connaît pas à l'avance le nombre de cases qui seront nécessaires.
- Toutes les cases du tableau ne sont pas forcément occupées, et de ce fait, il faut gérer les cases « libres » en y stockant une valeur spéciale (ex. `null` pour un tableau d'instances), ou en plaçant toutes les valeurs au début du tableau et en mémorisant leur nombre. L'utilisation du tableau est alors plus compliquée, et les opérations d'ajout, et de retrait ne sont pas immédiates à programmer.

Pour remédier à ces difficultés, Java offre une classe (**Vector**) dont les instances sont des sortes de tableaux qui changent de taille sans que l'on n'ait besoin de s'en préoccuper, et sur lesquels on peut appliquer des opérations qui facilitent la gestion du tableau. Les éléments sont indexés à partir de 0, comme dans les tableaux.

Notez cependant les problèmes suivants.

- ▷ Un ennui majeur est que seules des instances peuvent être stockées dans les vecteurs. Pour y placer des valeurs des types primitifs, il faut les « envelopper » dans des instances de classes spéciales (**Integer**, **Float**, etc.) Nous ne développerons pas cet aspect.
- ▷ Les méthodes de la classe **Vector** prennent comme paramètres et retournent des instances dont le type annoncé est **Object**. Vous ne vous rendrez compte de rien lors des passages de paramètre, car une instance de n'importe quelle classe est aussi instance de la classe **Object**. Par contre, lorsque nous récupérons une valeur dans un vecteur, Java considère que son type est **Object**. Il faudra lui préciser, par une « transformation de type » (casting), si c'est en réalité une instance de **Voyageur**, ou de **Voiture**, (etc.) pour que l'application de méthodes spécifiques de la classe **Voyageur**, **Voiture** ou autre soit admissible.
- ▷ Puisqu'on peut mettre dans un vecteur toute instance de toute classe, lorsque vous souhaitez créer un vecteur ne contenant que des **Voyageurs**, c'est entièrement sous votre responsabilité. S'il vous prend l'envie d'y mettre un chou-fleur, ce n'est pas le compilateur qui vous en empêchera, par contre, quand vous demanderez le sexe du chou-fleur au moment de l'exécution, le programme échouera, donc attention ! Des attributs de type vecteur nous serviront en particulier à implémenter les associations de cardinalité supérieure à 1. Pour éviter que n'importe quoi puisse y être mis, nous nous garderons le plus souvent de laisser le libre accès aux vecteurs (nous éviterons même les méthodes de type **get** ou **set** sur le vecteur lui-même).

Nous allons regarder certaines des méthodes de la classe **Vector**, parmi les plus utiles. Nous les illustrons en écrivant progressivement un programme à peu près équivalent à **AppliVoyageurTableau**.

Déclaration d'un vecteur

Un vecteur est une instance de la classe tableau. Il suffit donc de déclarer une telle instance.

```
Vector vectVoyageurs;
```

Création d'un vecteur

Différents constructeurs sont disponibles pour la classe, nous utiliserons le plus simple, qui n'a pas de paramètre, et crée un vecteur vide d'éléments.

```
vectVoyageurs = new Vector();
```

Taille d'un vecteur

A tout moment après sa création, on peut consulter la taille du vecteur en lui appliquant la méthode `int size()`. Ici par exemple, on veut afficher cette taille.

```
System.out.println("taille = "+vectVoyageurs.size());
```

Ajout d'un élément

L'ajout d'un élément s'effectue à l'aide de la méthode `void addElement(Object o)` ou de la méthode `boolean add(Object o)` qui sont équivalentes (la deuxième est arrivée avec la version 1.2). Le nouvel élément est ajouté à la fin du vecteur.

```
// Création et saisie des données des voyageurs
System.out.println("Combien de voyageurs désirez-vous créer ?");
int nombreVoyageurs = cons.lireEntier();

for (int i=0; i<nombreVoyageurs; i=i+1)
{
    Voyageur v = new Voyageur();
    v.saisie(cons);
    vectVoyageurs.addElement(v);
}
```

Le schéma de la figure 3.2 est valable également, en première approximation, pour se représenter le vecteur.

Consultation (accès) d'un élément

L'accès à l'élément d'indice `i` s'effectue grâce à la méthode `Object elementAt(int i)` ou à la méthode `Object get(int i)` qui est équivalente (Java 1.2).

Lorsque l'on applique cette méthode, l'expression résultante, par exemple

```
vectVoyageurs.elementAt(i)
```

est une valeur du type `Object`. De ce fait, on ne peut appliquer à l'objet récupéré que des méthodes de la classe `Object`. L'application d'une méthode de la classe `Voyageur` telle que :

```
vectVoyageurs.elementAt(i).carteVermeil()
```

provoquera une erreur de compilation si on en reste là.

Il faut signaler au compilateur que nous souhaitons qu'il considère cet objet comme étant de la classe `Voyageur`, et pour cela, on utilise une transformation de type (encore appelée « transtypage »), dont la syntaxe est la suivante :

```
(NouveauType)valeur
```

Dans notre cas, cela s'écrira :

```
(Voyageur)(vectVoyageurs.elementAt(i))
```

Il est souvent préférable de récupérer l'objet dans une variable intermédiaire pour des raisons de lisibilité. Dans les instructions suivantes, vous en voyez une illustration.

```
//Affichage des informations relatives à la carte vermeil
for (int i=0; i<vectVoyageurs.size(); i=i+1)
{
    Voyageur voyageurCourant = (Voyageur)(vectVoyageurs.elementAt(i));
    System.out.println(voyageurCourant.getNom()
        +" a droit a la carte vermeil ? "
        +voyageurCourant.carteVermeil()+"\n");
}
```

Chaîne de caractères associée à un vecteur

On peut écrire un code spécifique pour cela, ou bien utiliser la méthode `String toString()`, de manière explicite, ou non.

Dans l'exemple ci-dessous, elle est appelée de manière implicite pour effectuer l'affichage du vecteur : quand on concatène une valeur à une chaîne, en effet, la valeur est automatiquement « transformée » en chaîne. Si la valeur est une instance, ceci est réalisé automatiquement (vous n'écrivez rien vous-même) par un appel de la méthode `toString` sur l'instance.

```
//Affichage des voyageurs
System.out.println("Voyageurs contenus dans le vecteur\n"+vectVoyageurs);
```

La chaîne associée au vecteur par la méthode `toString` prédéfinie est construite en concaténant un crochet ouvrant, le résultat de l'appel de `toString` sur chaque élément du vecteur (les différents résultats sont séparés par des virgules), un crochet fermant.

Ainsi, si la classe `Voyageur` dispose de la méthode suivante :

```
public String toString()
{
    return '\n'+nom+"/"+age+"an(s)/"+sexe;
}
```

Alors un affichage du vecteur (correspondant aux valeurs de la figure 3.2) peut être :

```
[
Louis/62an(s)/M,
Elisa/59an(s)/F,
Eloise/64an(s)/F]
```

Et pour aller plus loin.

Consultez la documentation de la classe `Vector`, vous y trouverez de nombreuses autres méthodes.

Cours 4

Généralisation - Spécialisation - 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, les deux mécanismes relèvent d'une démarche plus générale qui consiste à « classifier » les concepts manipulés.

La généralisation est un mécanisme qui consiste à abstraire des caractéristiques communes à plusieurs classes en 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**, un examen de ces classes montre qu'elles 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.

Remarque : Le résultat final est toujours une hiérarchie de classes. On peut préciser un certain nombre de contraintes comme **complet**, **disjoint** qui précise si toutes les sous-classes possibles ont été définies et si les extensions des sous-classes sont des ensembles disjoints.

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

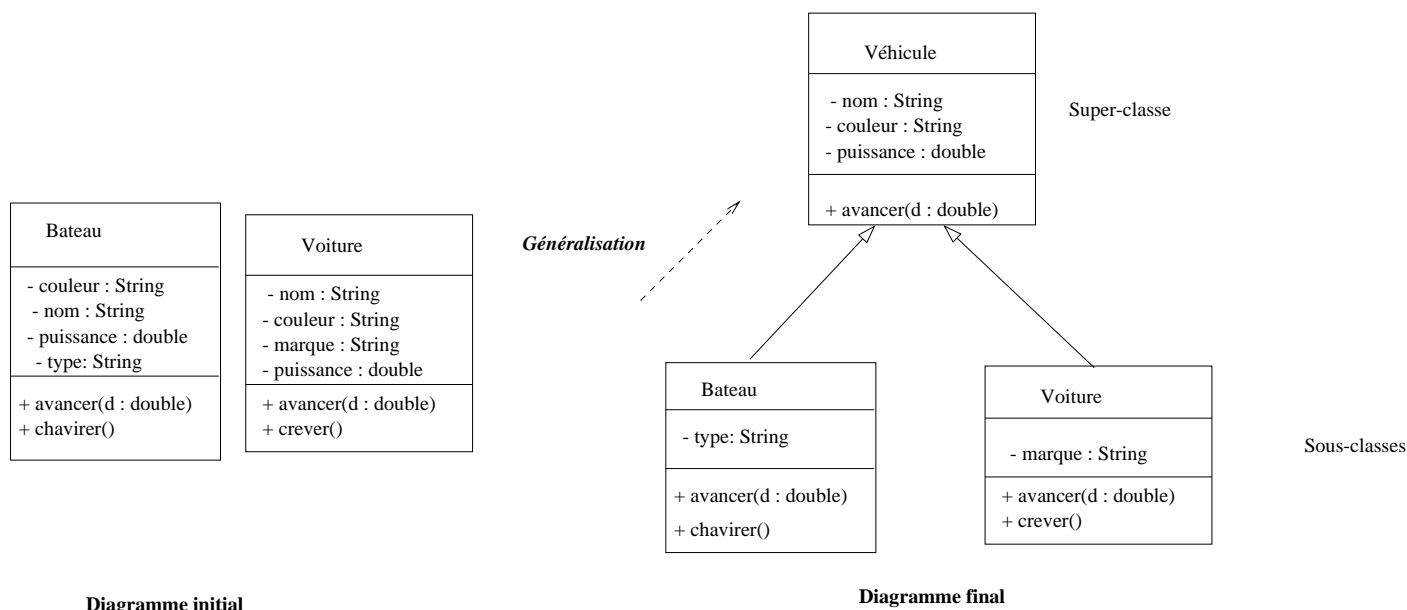


FIG. 4.1 – Une généralisation.

4.2 Hiérarchie des classes et héritage dans Java

Pour les langages orientés-objets, le programmeur définit des hiérarchies de classes provenant d'une conception dans laquelle il a utilisé généralisation-spécialisation, on dit le plus souvent la sous-classe « hérite » de la super-classe (ou qu'elle étend la super-classe ou 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 présentent un mécanisme dénommé « 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 « bon » code correspondant au message.

En Java,

- toutes les classes dérivent de la classe `Object`, qui est la racine de toute hiérarchie.
- une classe ne peut avoir qu'une seule super-classe. On parle d'héritage « simple ». Il existe des langages orienté-objets, tels C++ qui autorisent l'héritage multiple.

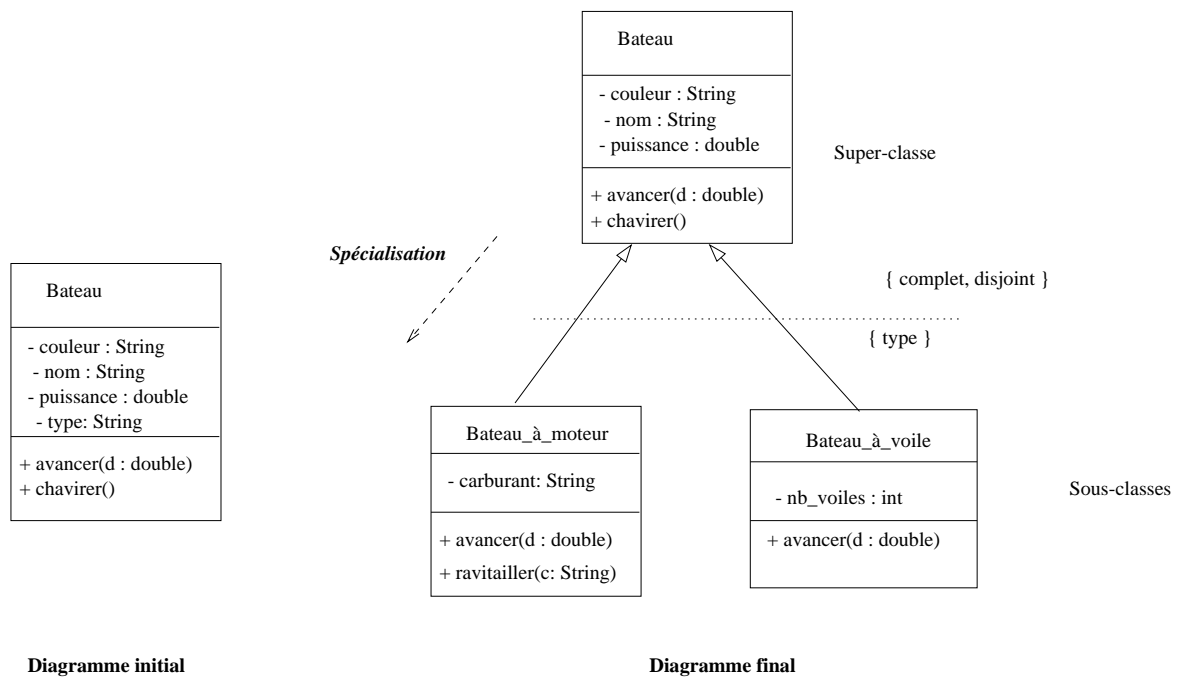


FIG. 4.2 – Une spécialisation.

– le mot clef permettant de définir la généralisation-spécialisation entre classes est **extends**. Pour la suite du chapitre nous prenons une hiérarchie de classes représentée dans le diagramme 4.3.

```

package Prog.SPEC;
import java.io.*;
public class Personne{
.....} // fin de la classe Personne

.....
public class Etudiant extends Personne {

.....

} // fin de la classe Etudiant

.....
public class Etud_Licence extends Etudiant {

```

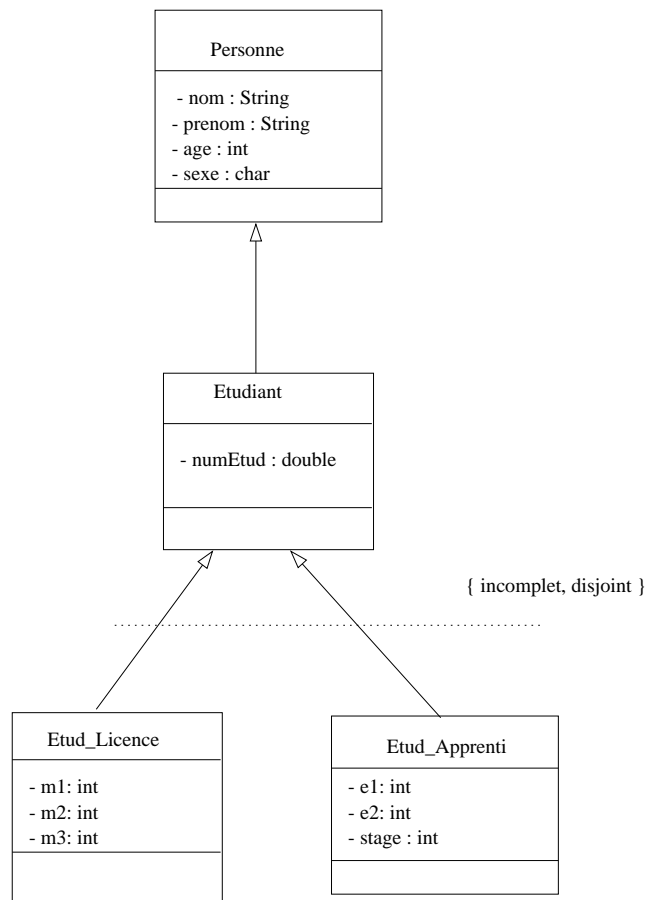


FIG. 4.3 – Une hiérarchie de classes.

```

.....

} // 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 `incrementerAge()` (cette méthode vieillit d'un an).

Les attributs définis dans la classe `Etudiant` sont `numEtud`. La classe `Etudiant` définit des opérations de calculs de moyenne, d'admission, de mention.

Pour les étudiants de licence (classe `Etud_Licence`, ils passent 3 modules (M1, M2, M3) de coefficient 1. Ils sont admis si leur moyenne générale ≥ 10 et conservent les modules obtenus s'ils ne sont pas admis.

Les étudiants apprentis (classe `Etud_Apprenti`) passent 2 modules (E1, E2) de coefficient 1 et un stage de coefficient 2. Ils sont admis si leur moyenne est ≥ 10 et si leur note de stage est > 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.incrementerAge();
System.out.println(a.getName()+"est âgé de " + a.getAge()+" ans");
```

A l'exécution, on obtient :

```
Einstein Albert est âgé de 23 ans
```

4.3 Redéfinition de méthodes - Surcharge - Masquage

Une opération (ou une méthode) est déclarée dans une classe, possède un nom et une liste de paramètres, on parle de sa « signature » (nom de classe + nom de l'opération + 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 « 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();
    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. *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. Mais SUN en a décidé autrement. De plus, par défaut (« package protected ») la méthode (ou l'attribut) est n'accessible que par les classes du même package.
- **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.

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 otant 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 cette classe.

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 abstraite sous-classe de *Personne* 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 processus de 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). La transformation de type est en particulier utilisée lorsque l'on récupère des objets dans un vecteur (la classe **Vector** étant définie comme une collection d'éléments de la classe **Object**). Dans ce contexte là, le casting permet de préciser, si nécessaire, au compilateur, la classe désirée de l'objet récupéré.

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
```

Mais, cette opération devient plus fondamentale dans le cas de l'utilisation de hiérarchies de classes (bien que l'on eût souhaité qu'elle soit plus automatisable). Prenons l'exemple suivant.

```
//la classe Promotion  
  
.....  
public class Promotion {  
private Vector listeEtudiants = new Vector();  
.....  
public double moyenneGenerale() {  
double somme = 0;
```

```

    for (int i=0;i<listeEtudiants.size();i++)
    {
        //le casting a lieu ici :
        Etudiant etud = (Etudiant)(listeEtudiants.elementAt(i));
        somme = somme + etud.moyenne();
    }
    if (listeEtudiants.size()>0)
        return (somme / listeEtudiants.size());
    else return 0;
}

```

Les éléments d'un Vector sont des instances d'Object. La transformation de type `(Etudiant)(listeEtudiants.elementAt(i))` permet de préciser à `listeEtudiants.elementAt(i)` que l'on va s'adresser à lui non pas en tant qu'instance d'Object mais en tant qu'instance d'Etudiant, car l'appel de la méthode `moyenne()` sur une instance de la classe Object entraîne une erreur de compilation, cette classe ne définissant pas `moyenne`.

A l'exécution, il est cependant notable de remarquer que chaque objet extrait du vecteur `listeEtudiants` exécutera le code de la méthode `moyenne()` afférent à sa classe « mère », c'est-à-dire que les étudiants de licence exécuteront le code de la classe `Etud_Licence` et les étudiants apprentis exécuteront le code de la classe `Etud_Apprenti`. L'appel `etud.moyenne()` est donc polymorphe (puisqu'il recouvre diverses formes de calcul).

```

.....
public class Personne{
    .....
    public void f1() {
        System.out.println("Methode f1(), de la classe Personne");}
}
.....
public class Etud_Apprenti extends Etudiant { //qui est, elle-même, sous
classe de Personne
//Redéfinition de f1()
    public void f1() {
        System.out.println("Methode f1(), de la classe Etud_Apprenti");}

//Nouvelle methode f2 introduite dans Etud_Apprenti (pas de redéfinition)
    public void f2() {
        System.out.println("Methode f2(), de la classe Etud_Apprenti");}
}
.....
// la classe Appli_Cursus
public class Appli_Cursus {

```

```

public static void main(String args[]) {
    Personne p;
    Etud_Apprenti a1 = new Apprenti(); // constructeur par défaut

    p = a1;
        // OK - transformation implicite autorisée car un Etud_Apprenti est une personne !

    p.f1();
        // affiche : Methode f1(), de la classe Etud_Apprenti (et oui ...)
    p.f2();
        // Attention ERREUR à la compilation: methode f2() not found in Personne

    Etud_Apprenti a2 = (Etud_Apprenti) p;
        // OK - cast down autorisé à la compilation

    a2.f1();
        // affiche : Methode f1(), de la classe Etud_Apprenti
        // en fait p référençait un Etud_Apprenti a1 et a2 référence donc
aussi a1
    a2.f2();
        // affiche : Methode f2(), de la classe Etud_Apprenti

    p = new Personne();
    p.f1();
        // affiche : Methode f1(), de la classe Personne
    Etud_Apprenti a3 = p;
        // ERREUR de compilation
        // incompatible type found: Personne, required: Apprenti
        // une Personne n'est pas un Etud_Apprenti

    Etud_Apprenti a3 = (Etud_Apprenti) p;
        // le compilateur accepte
    a3.f1();
        // le compilateur accepte
        // MAIS ERREUR d'exécution java.lang.ClassCastException: Personne
    }
}

```

A l'exécution (une fois supprimé les erreurs), on obtient :

```

Methode f1(), de la classe Etud_Apprenti
Methode f1(), de la classe Etud_Apprenti

```

Methode f2(), de la classe Etud_Apprenti
Methode f1(), de la classe Personne

Une transformation de type d'objet ne peut s'appliquer qu'à des objets instances de classes d'un même chemin d'héritage au sein d'une hiérarchie de classe.

```
Etud_Apprenti a=new Etud_Apprenti();
```

```
Voiture v = (Voiture) a //ne marchera pas !
```

Pour vérifier qu'une instance appartient bien à une classe précise d'une hiérarchie, on peut utiliser l'opérateur `instanceof`.

Par exemple, dans la classe Promotion, ajoutons une méthode qui incrémente les notes E1 des étudiants apprentis :

\\ dans la classe Promotion

```
public void incrNoteE1Apprentis() {
    for (int i=0;i<listeEtudiants.size();i++)
        {
            if (listeEtudiant.elementAt(i) instanceof Etud_Apprenti)
                {Etud_Apprenti a = (Etud_Apprenti) listeEtudiant.elementAt(i);
                 a.setE1(a.getE1()+1);
                }
        }
    // setE1 est une méthode qui n'existe que dans la classe Etud_Apprenti
}
}
```

4.7.2 Le polymorphisme

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

```

```

.....
public class Promotion {
private Vector listeEtudiants = new Vector();
.....
public double moyenneGenerale() {
    double somme = 0;
    for (int i=0;i<listeEtudiants.size();i++)
        {Etudiant etud = (Etudiant)(listeEtudiants.elementAt(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
    // "Objects" du vecteur (listeEtudiants).

    for (int i=0;i<listeEtudiants.size();i++)
        listeNom = listeNom +listeEtudiants.elementAt(i).toString() + " \n";
    // ici listeEtudiants.elementAt(i) est considéré comme instance d'Object
    // mais
    // Object définit toString()
    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 :

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 une liste d'étudiants en `Etud_Apprenti` ou `Etud_Licence` pour calculer leur `moyenne()`. Mais, la compilation nous oblige cependant à caster `Object` en `Etudiant` car la méthode `moyenne()` n'est pas définie dans la classe `Object`. Ensuite par contre, plus besoin de transformer le type explicitement, puisque le polymorphisme fonctionne, grâce à la liaison dynamique du langage qui permet de trouver à l'exécution la « bonne » méthode (c'est-à-dire celle de la classe correspondant soit à la classe dont l'objet est issu, soit celle d'une super-classe).

Cours 5

Compléments sur les variables et les opérations, récursivité, exceptions

5.1 Compléments sur les variables et les opérations

5.1.1 Constantes

On peut déclarer des constantes en Java, en utilisant le mot-clef `final`. Une variable `final` ne peut être initialisée qu'une fois. Cependant, cette initialisation peut ne pas être faite au moment de la définition de la variable.

En Java, par convention, les noms des constantes sont en majuscules. Toutes les sortes de variables peuvent être constantes (attributs, variables de classe, variables locales, paramètres).

Exemple 1. On veut créer une classe `compteBancaire` disposant d'un attribut constant correspondant au numéro de compte, et ce numéro de compte est attribué au moment de la création d'une instance de compte. Dans le programme ci-dessous, on vous montre également que vous ne pouvez pas tenter une autre affectation, donc cela exclut d'écrire une méthode d'accès de type `set`.

```
public class Compte
{
    private final String NUMERO;
    private float solde;

    public Compte(String num)
    {
        NUMERO=num;
        solde=0f;
    }
    public String getNUMERO()
    {
        return NUMERO;
    }
}
```

```

}

public void setNUMERO(String n)
{
    //cette instruction est interdite (erreur de compilation)
    //NUMERO=num;
}

.....
}

```

Exemple 2. Vous voulez vous empêcher de programmer comme un cochon en modifiant la valeur d'un paramètre dans une méthode : passez-le accompagné de ce mot-clef! Ainsi la méthode suivante provoque une erreur de compilation.

```

public void machin(final int n)
{
    n+=4;

}

```

5.1.2 Variables et méthodes de classe

Les attributs (ou variables d'instance) et les méthodes que nous avons rencontrés jusqu'à présent étaient propres (s'appliquaient) à une instance.

Il existe une autre catégorie de variables et de méthodes, dites « de classe » qui permettent de décrire des informations et des opérations :

- relatives à la classe, et non à une instance particulière,
- partagées par l'ensemble des instances d'une classe.

Quelques exemples de variables de classe :

- le nombre de côtés pour les carrés,
- le président de la république pour les français,
- la collection des noms de mois, de jours, pour une classe `Date`,
- la liste des fenêtres créées pour une classe `Fenêtre`,
- le nombre de comptes bancaires créés (qui peut servir à attribuer un numéro à un compte).

Mais les variables de classes associées à une classe `C1` sont parfois mieux modélisées comme variables d'instance d'une classe `C2` qui se présente comme un « conteneur » de `C1`. Par exemple, la liste des fenêtres peut apparaître alternativement comme variable d'instance d'une classe `GestionnaireFenetres`. Cela permet de plus d'avoir ensuite la possibilité de créer plusieurs gestionnaires de fenêtres différents, chacun disposant de sa propre liste de fenêtres gérées.

Quelques exemples de méthodes de classe :

- création d'une instance prototypique de la classe (`Pi` pour une classe `réel`),
- manipulation des variables de classe,

– statistiques sur les instances.

Déclaration de variables de classes

Les variables de classes sont déclarées comme des attributs et précédées du mot-clé `static`.

```
class Francais extends Personne
{
    ....
    private static Personne présidentDelaRépublique;
    ....
}
```

```
class Carre
{
    public final static int NBCOTES=4;
    private double lgCoté;
}
```

Définition de méthodes de classes Les méthodes de classe, comme les variables, sont précédées du mot-clé `static`.

```
public class Francais extends Personne
{
    private String numInsee;
    ....
    private static Personne présidentDelaRépublique;

    public static Personne getPrésidentDelaRépublique()
    {
        return présidentDelaRépublique;
    }

    public static void setPrésidentDelaRépublique(Personne p)
    {
        présidentDelaRépublique=p;
    }
    ....
}
```

Ces méthodes ne peuvent manipuler ni `this`, ni `super`, puisqu'elles ne concernent pas une instance. Elles ne peuvent pas non plus accéder directement (sans mention d'un objet) à des variables d'instance.

En Java la méthode `main` est aussi, comme vous le comprenez, une méthode de classe, sa particularité est qu'elle est appelée par l'interprète.

Utilisation des variables et méthodes de classes Dans les méthodes de la classe où elles sont déclarées, c'est sans ambiguïté (voir les méthodes ci-dessus). Dans une autre classe, ou dans un programme principal, on les appelle de préférence (à condition qu'elles soient visibles) sur la classe et non sur une instance.

```
public static void main(String[] arg)
{
    Personne pdt=new Personne();

    Francais.setPrésidentDelaRépublique(pdt);

    System.out.println(Francais.getPrésidentDelaRépublique());

    System.out.println("nombre de cotes d'un carre "+Carre.NBCOTES);
}
```

Une variable de classe est partagée par les instances de la classe et de ses sous-classes. Elle n'est pas héritée. Ainsi, même si on définit une sous-classe de `Francais`, la classe `FrancaisCitadin` (munie par exemple d'un attribut qui représente la ville), il n'y a toujours qu'une variable `présidentDelaRépublique`, comme le montre la figure 5.1. Il y a une variable (attribut) `numInsee` dans chaque instance, mais une seule variable `présidentDelaRépublique` commune à toutes les instances (`f1`, `f2` sont des instances de `Francais` et `fc1` une instance de `FrancaisCitadin`).

Variables et méthodes de classe dans l'API Java L'API Java contient de nombreux exemples de variables et de méthodes de classe.

- `out` est une variable de classe constante de la classe `System`, elle contient une instance de la classe `PrintStream` qui est associée au flot de sortie standard du système. La classe `PrintStream` dispose de la méthode `println` que vous utilisez pour afficher.
- la classe `Math` est un recueil de méthodes statiques qui ne sont autres que des fonctions mathématiques usuelles. Par exemple `round` pour l'arrondi, ou `sqrt` pour la racine carrée. La classe contient aussi les constantes `PI` et `E`.
- les classes enveloppant les types primitifs, contiennent des fonctions de conversion, ou des constantes liées à ces types. Par exemple, `Integer` contient `int parseInt(String)` qui convertit une chaîne en `int`, et les constantes `MAX_VALUE` et `MIN_VALUE` qui contiennent les bornes du type.

5.2 Récursivité

Une méthode récursive est une méthode qui se rappelle elle-même. La récursivité permet de ce fait de répéter une opération et peut remplacer l'utilisation d'une structure de contrôle

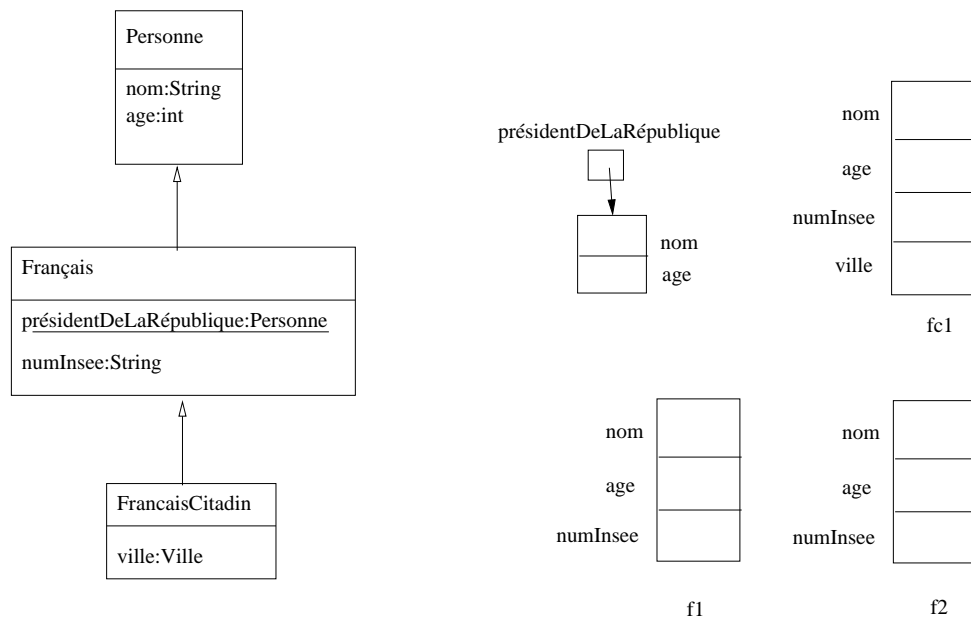


FIG. 5.1 – Variation sur la classe Français.

répétitive.

Les objets peuvent aussi être récursifs. Par exemple, l'objet `Dessin-étiquette-de-vache-qui-rit` contient un `Dessin-de-vache` qui lui-même contient deux `Dessin-de-Boucles-d'oreilles` qui contiennent chacun un `Dessin-étiquette-de-vache-qui-rit`. Une séquence d'éléments peut se définir comme un premier élément suivi d'une séquence d'éléments.

5.2.1 Méthodes de classe récursives

Pour traduire d'une manière très directe les fonctions récursives, on peut écrire des méthodes de classe. Ce n'est pas très « orienté objet », mais dans un langage comme Java où les nombres, les caractères et les tableaux ne sont pas vraiment des objets, c'est parfois une solution satisfaisante.

factorielle Le calcul récursif se base sur la définition suivante :

```

pour n entier >=0,
si n=0, factorielle(n)=1,
si n>0, factorielle(n)=n*factorielle(n-1)
  
```

Nous en donnons tout d'abord une version récursive, une version itérative, puis leur utilisation dans un `main`.

```

public class Nombres
{
    /** version recursive
  
```

```

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

/** version itérative
    n est suppose >=0 */
public static int factorielleI(int n)
{
    int fact=1;
    for (int i=1; i<=n; i=i+1)
        fact = fact * i;
    return fact;
}

....
}

.....
public static void main(String[] arg) throws java.io.IOException
{
    ConsoleStandard c = new ConsoleStandard();

    System.out.println("Entrer un nombre >=0");
    int n=c.lireEntier();
    System.out.println("fact R = "+Nombres.factorielle(n)
        +" fact I = "+Nombres.factorielleI(n));
}

```

pgcd Dans ce deuxième exemple, nous montrons que le corps de la fonction peut contenir plusieurs appels récursifs.

Le plus grand commun diviseur de deux nombres entiers a et b peut se calculer ainsi :

- si $a = 1$ ou $b = 1$, $pgcd(a, b) = 1$, sinon,
- si $a = b$, $pgcd(a, b) = a = b$,
- si $a > b$, $pgcd(a, b) = pgcd(a - b, b)$,
- si $a < b$, $pgcd(a, b) = pgcd(a, b - a)$,

Un exemple de fonctionnement :

```

pgcd(18,15)
  pgcd(3,15)
    pgcd(3,12)
      pgcd(3,9)

```

```
    pgcd(3,6)
    pgcd(3,3) ----> 3
```

```
public class Nombres
{
....
/** version récursive */
public static int pgcd(int a, int b)
{
    if ((a==1) || (b==1)) return 1;
    if (a > b) return pgcd(a-b, b);
    if (a < b) return pgcd(a, b-a);
    // a==b
    return a;
}
/** version itérative */
public static int pgcdI(int a, int b)
{
    int aa = a, bb = b;
    while ((aa!=bb) && (aa!=1) && (bb!=1))
    {
        if (aa > bb) aa=aa-bb;
        else bb=bb-aa;
    }
    if ((aa==1) || (bb==1)) return 1;
    else return aa;
}

...
}

...
public static void main(String[] arg) throws java.io.IOException
{
    ConsoleStandard c = new ConsoleStandard();

    System.out.println("Entrer a puis b");
    int a=c.lireEntier();
    int b=c.lireEntier();
    System.out.println("pgcd R = "+Nombres.pgcd(a,b)
        +" pgcd I = "+Nombres.pgcdI(a,b));
}
}
```

5.2.2 Méthodes d'instance récursives sur un même objet

Une saisie récursive. Dans ce premier exemple, on cherche à écrire pour la classe `Etudiant` une méthode permettant de saisir un code d'inscription correct (égal à 1 ou 2). Seul l'extrait de classe nécessaire à cet exemple et au suivant est donné. Le principe de la méthode est le suivant : saisir une fois, si la valeur est incorrecte, recommencer !

```
public class Etudiant
{
    private String nom;

    public Etudiant(String n)
    {nom=n;}

    public String getNom()
    {return nom;}

    public void setNom(String n)
    {nom=n;}

    private int codeIns;
    public int getCodeIns()
    {return codeIns;}

    public void setCodeIns(int ci)
    {if ((ci==1) || (ci==2)) codeIns=ci; else codeIns=0;}

    public void saisieCodeIns(ConsoleStandard c) throws java.io.IOException
    {
        System.out.println("Veuillez entrer un code d'inscription (1 ou 2)");
        codeIns=c.lireEntier();
        if ((codeIns!=1) && (codeIns!=2))
            saisieCodeIns(c);
    }
}
```

Cette méthode peut être appelée ainsi.

```
.....
public static void main(String[] arg) throws java.io.IOException
{
    ConsoleStandard c = new ConsoleStandard();
    Etudiant e1 = new Etudiant("Jules");
    e1.saisieCodeIns(c);
}
```

Une recherche récursive. Nous proposons une version récursive de la méthode de recherche d'un étudiant dans une promotion (on suppose qu'il n'y a pas d'homonymes). Là aussi, seul l'extrait utile de la classe est donné. La méthode de recherche principale appelle une méthode récursive qui fonctionne sur le principe suivant.

- Elle admet deux paramètres : le nom de l'étudiant cherché, et l'indice de début de la recherche dans le vecteur.
- Si l'indice de début de la recherche atteint l'indice de fin du tableau (donné par le nombre d'étudiants) ou s'il n'y a pas d'étudiants dans la promotion, on retourne `null`.
- sinon,
 - soit l'étudiant repéré par l'indice est celui cherché, et on le retourne,
 - soit il ne correspond pas, et on cherche dans la suite du vecteur.

La méthode de recherche principale appelle la méthode secondaire en partant à l'indice 0. Si le vecteur est vide, l'indice qui vaut 0 atteint le nombre d'étudiants et on arrête immédiatement la recherche.

A titre d'exercice, vous pouvez écrire une version de recherche lorsqu'on suppose qu'il peut y avoir des homonymes et que l'on veut obtenir tous les étudiants ayant un nom donné.

```
public class Promotion
{
    private Vector le = new Vector();

    public int nbEtudiants()
    {return le.size();}

    public void inscrit (Etudiant e)
    {le.addElement(e);}

    public Etudiant etudiant(int i)
    {return (Etudiant)(le.elementAt(i));}

    private Etudiant rechR(String nom, int deb)
    {
        if (deb==nbEtudiants())
            return null;
        else
            if (etudiant(deb).getNom().equals(nom))
                return etudiant(deb);
            else
                return rechR(nom, deb+1);
    }

    public Etudiant recherche(String nom)
```

```

    {return rechR(nom,0);}
}

.....
public static void main(String[] arg) throws java.io.IOException
{
    ConsoleStandard c = new ConsoleStandard();

    Etudiant e1 = new Etudiant("Jules");
    Etudiant e2 = new Etudiant("Elise");
    Etudiant e3 = new Etudiant("Nina");
    Etudiant e4 = new Etudiant("Simon");

    Promotion p = new Promotion();
    System.out.println("trouve Jules avant insertion ? "+p.recherche("Jules"));
    p.inscrit(e1);
    p.inscrit(e2);
    p.inscrit(e3);
    p.inscrit(e4);
    System.out.println("Promotion "+p.le);

    System.out.println("trouve Jules après insertion ? "+p.recherche("Jules"));
    System.out.println("trouve Nina après insertion ? "+p.recherche("Nina"));
    System.out.println("trouve Simon après insertion ? "+p.recherche("Simon"));
}

```

Dans l'exemple précédent (la saisie récursive) on réappliquait la saisie au même objet. C'est encore le cas dans la recherche, mais à y regarder de plus près, le passage d'un paramètre représentant l'indice de début de la recherche peut s'interpréter comme un appel récursif sur une partie réduite de l'objet (la promotion sans les éléments allant de 0 à indice de début - 1). C'est typique de la récursivité : on réduit le problème à chaque étape. Par ailleurs on considère, dans cette approche, une définition récursive du vecteur, au moins abstraitement. Selon cette définition, un vecteur est soit vide, soit un (premier) élément suivi d'un autre vecteur.

5.2.3 Méthodes d'instance récursives sur des objets définis récursivement

C'est une variante de la récursivité spécifique des objets lorsque le diagramme de classe présente un circuit de composition (figure 5.2), dans les deux exemples suivants, il s'agit d'une boucle :

- un mot est soit vide, soit une lettre suivie d'un mot,
- une personne a deux parents qui sont des personnes (les parents peuvent être inconnus).

Nous allons regarder plus en détails ce dernier exemple, et nous intéresser par exemple à l'affichage de tous les ascendants d'une personne.

La classe est écrite d'une manière très simplifiée, en particulier, on pourrait vérifier des contraintes sur le père et la mère d'une personne, ou accepter de naviguer l'association dans les deux sens.

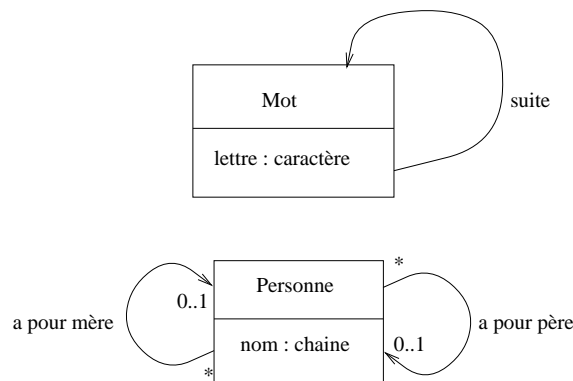


FIG. 5.2 – Définitions récursives de classes.

Pour obtenir les noms de tous les ascendants d'une personne, on récupère le nom de sa mère, et les noms des ascendants de sa mère, puis le nom de son père et les noms des ascendants de son père. L'appel récursif est donc fait sur un autre objet que l'objet courant : on se déplace sur les liens entre les objets. La méthode contient une fantaisie supplémentaire, un décalage est appliqué qui dépend de la profondeur de l'ascendance, ainsi l'affichage est indenté et rend l'arbre généalogique plus lisible. A titre d'exercice : que se passe-t-il si dans l'ascendance on retrouve une même personne par plusieurs chemins ?

```
public class Personne
{
    private String nom;
    private Personne mere;
    private Personne pere;

    public Personne(String n){nom=n;}

    public String getNom(){return nom;}
    public void setNom(String n){nom=n;}
    public Personne getMere(){return mere;}
    public void setMere(Personne p){mere=p;}
    public Personne getPere(){return pere;}
    public void setPere(Personne p){pere=p;}
    public String retourneNomsAscendants(String decalage)
    {
        String S="";
        if (mere!=null) S=S+decalage+mere.getNom()+"\n"
                        +mere.retourneNomsAscendants(decalage+'t');
        if (pere!=null) S=S+decalage+pere.getNom()+"\n"
                        +pere.retourneNomsAscendants(decalage+'t');
        return S;
    }
}
```

```

}

public static void main(String[] arg)
{
    Personne a = new Personne("Zoé");
    Personne am = new Personne("Isabelle");
    Personne amm = new Personne("Denise");
    Personne amp = new Personne("Francois");
    Personne ampm = new Personne("Edmée");
    Personne ap = new Personne("Bernard");
    Personne apm = new Personne("Josette");
    Personne app = new Personne("Henri");

    a.setMere(am);
    a.setPere(ap);
    am.setMere(amm);
    am.setPere(amp);
    ap.setMere(apm);
    ap.setPere(app);
    amp.setMere(ampm);

    System.out.println(a.getNom()+'\n'+a.retourneNomsAscendants("\t"));
}
}

```

Les objets créés sont représentés figure 5.3.
Voici le résultat de l'exécution du programme.

```

Zoé
    Isabelle
        Denise
        Francois
            Edmée
    Bernard
        Josette
        Henri

```

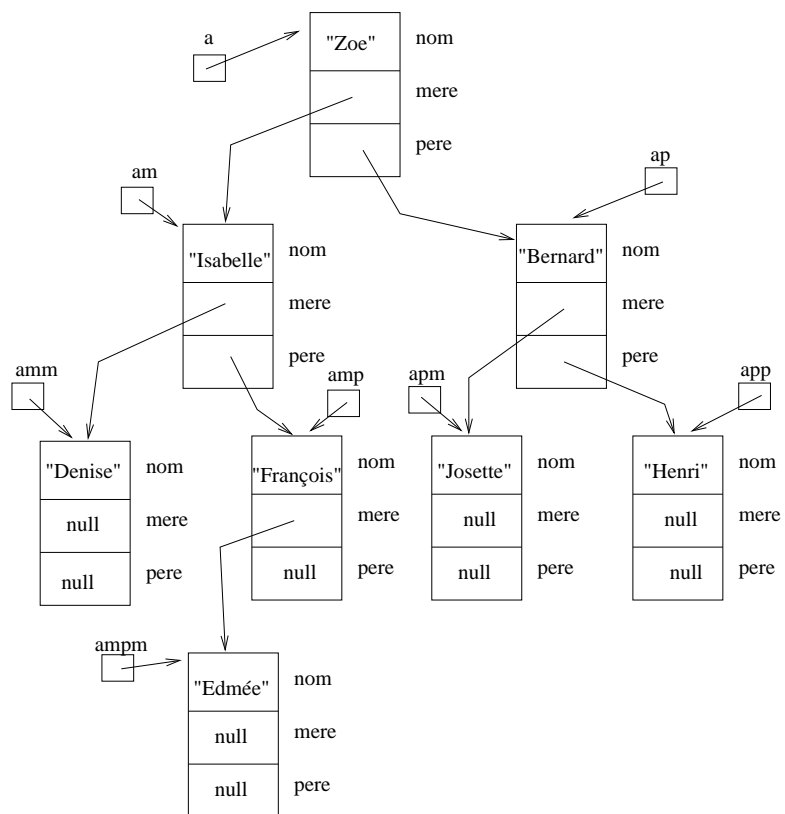


FIG. 5.3 – Un arbre généalogique.

Cours 6

Exceptions

6.1 Introduction aux exceptions

Dans l'utilisation d'un programme il peut se produire un nombre incalculable d'erreurs (ou situations exceptionnelles) :

- erreurs de saisie (on saisit un caractère là où est attendu un nombre),
- erreurs de programmation (accès à un indice incorrect dans un tableau, division par zéro, non-respect de contraintes sur les valeurs d'un attribut),
- erreurs matérielles (disque plein, mémoire vive saturée)

Lorsqu'un tel problème se produit, il est bon de :

- garder les données dans un état cohérent,
- expliciter les problèmes par des messages d'erreur clairs,
- sortir du programme correctement.

Pour éviter de trop mélanger le code correspondant au comportement normal d'un programme et le code correspondant au signalement et à la récupération de ces erreurs, les langages modernes proposent en général un mécanisme dit de « gestion des exceptions ». Il comporte essentiellement trois aspects :

- la représentation des erreurs,
- le signalement des erreurs,
- la capture.

6.1.1 Représentation des erreurs

Le degré zéro de la représentation des erreurs serait d'utiliser un code (un entier par ex.) pour chaque sorte d'exception. En Java, les exceptions sont des objets, instances de classes particulières et il y a, grâce à l'héritage, une véritable classification des exceptions. La figure 6.1 présente un extrait de cette hiérarchie.

Throwable représente la racine des erreurs. **Error** représente les erreurs graves de fonctionnement (machine virtuelle, édition de liens), contre lesquelles il n'y a pas grand chose à faire, et **Exception** les erreurs qu'il est préférable de récupérer et de traiter. **RuntimeException** représente les exceptions qui peuvent se produire dans n'importe quel programme.

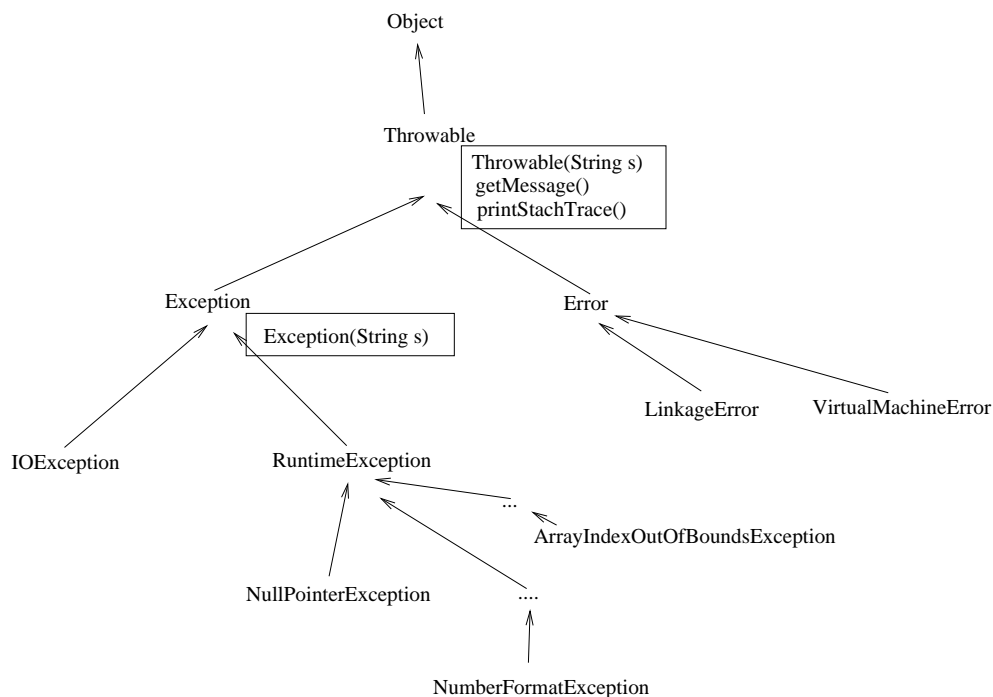


FIG. 6.1 – Les exceptions de Java.

Lorsque l'on développe une classe, on doit se poser la question des erreurs qui peuvent survenir pendant son utilisation. Ces erreurs doivent être représentées par de nouvelles sous-classes que l'on ajoute à `Exception`. Par exemple, commençons à développer une gestion des exceptions pour la classe `Etudiant`. On donne tout d'abord une racine à l'ensemble des exceptions associées à `Etudiant` : `EtudiantException`. Puis deux sous-classes servent à représenter par exemple les deux erreurs `CodeInsException` et `CodePaysException`.

```

class EtudiantException extends Exception
{
    EtudiantException(){}
    EtudiantException(String S){super("Exception-Eudiant "+S);}
}

class CodeInsException extends EtudiantException
{
    private int codeProblematique;
    CodeInsException(){}
    CodeInsException(String S){super("sur le code inscription "+S);}
    CodeInsException(String S, int ci)
    {super("sur le code inscription "+ci+S);
    codeProblematique=ci;}
}
  
```

```

class CodePaysException extends EtudiantException
{
    private int codeProblematique;
    CodePaysException(){
    CodePaysException(String S){super("sur le code pays "+S);}
    CodePaysException(String S, int cp)
    {super("sur le code pays "+cp+S);
     codeProblematique=cp;}
}

```

6.1.2 Signalement et déclaration dans l'entête de méthode

Le signalement d'une exception se fait grâce à l'instruction `throw`, la déclaration du fait qu'une méthode peut signaler une exception s'effectue grâce à une clause `throws` qui complète l'entête de la méthode.

Par exemple, nous pouvons réécrire la méthode `setCodeIns` de la classe `Etudiant` de la manière suivante.

```

class Etudiant
{
    ....
    private int codeIns;
    ....
    public void setCodeIns(int ci) throws CodeInsException
    {
        if ((ci==1) || (ci==2))
            codeIns=ci;
        else throw new CodeInsException(" dans setCodeIns",ci);
    }
}

```

Les sous-classes de `Error` et de `RuntimeException` sont ce que l'on appelle des exceptions « non vérifiées », par opposition, toutes les autres le sont. Ceci a des conséquences sur les entêtes des méthodes : une méthode dans laquelle peut se produire une exception vérifiée doit le préciser dans son entête. Ainsi, `IOException` (comme `CodeInsException`) est une exception vérifiée, et c'est pour cela que nous devons la déclarer partout où les méthodes de saisie au clavier sont utilisées, car ces dernières peuvent la signaler, par exemple nous l'avons fait dans l'entête de la plupart des méthodes `main` et dans celle de `saisieCodeIns`.

6.1.3 Capture

Lorsqu'une exception est signalée, le programme interrompt son cours normal. Si rien n'est prévu, il s'arrête et affiche le libellé de l'erreur et l'état de la pile d'exécution, ce qui permet de repérer où elle s'est produite.

Lorsque l'on veut capturer l'exception et terminer ou reprendre l'exécution proprement, on utilise une nouvelle structure de contrôle, `try ... catch`.

Par exemple, pour capturer l'exception `CodeInsException` on peut écrire

```
public static void main(String[] arg) throws java.io.IOException,CodeInsException
{
    try
    {
        ConsoleStandard c = new ConsoleStandard();
        Etudiant e1 = new Etudiant("Jules");
        e1.setCodeIns(c.lireEntier());
        System.out.println("bonne saisie de "+e1.getCodeIns());
    }
    catch(CodeInsException e)
    {
        System.out.println(e.getMessage());
    }

    Etudiant e2 = new Etudiant("Elise");

    ....
}
```

Dans cet exemple de programme, si une mauvaise valeur est saisie comme code d'inscription pour `e1`, le programme s'interrompt pour signaler l'erreur, recherche le bloc `try ... catch` qui peut la traiter (celui du `main`), exécute le bloc derrière le `catch` qui correspond au type de l'erreur, et reprend ensuite l'exécution à la première instruction qui suit le `try ... catch`. On peut mettre autant de clauses `catch` que l'on veut, si l'on souhaite surveiller simultanément la survenue de plusieurs exceptions.

Nous compliquons un peu en proposant une nouvelle implémentation de `saisieCodeIns` avec récupération au vol de l'exception (qui n'a donc pas besoin d'être signalée).

```
public void saisieCodeIns(ConsoleStandard c) throws java.io.IOException
{
    System.out.println("Veuillez entrer un code d'inscription (1 ou 2)");
    try
    {
        setCodeIns(c.lireEntier());
    }
}
```

```

    }
    catch(CodeInsException e)
    {
        System.out.println(e.getMessage());
        saisieCodeIns(c);
    }
}

```

Et enfin un nouveau programme ...

```

public static void main(String[] arg) throws java.io.IOException,CodeInsException
{
    ConsoleStandard c = new ConsoleStandard();

    Etudiant e1 = new Etudiant("Jules");
    try
    {
        e1.setCodeIns(c.lireEntier());
        System.out.println("bonne saisie de "+e1.getCodeIns());
    }
    catch(CodeInsException e)
    {
        System.out.println(e.getMessage());
    }

    Etudiant e2 = new Etudiant("Elise");
    e2.saisieCodeIns(c);
}

```

Premier exemple d'exécution (bonne saisie pour e1, deux mauvaises saisies pour e2).

```

1
bonne saisie de 1
Veuillez entrer un code d'inscription (1 ou 2)
3
Exception-Eudiant sur le code inscription 3 dans setCodeIns
Veuillez entrer un code d'inscription (1 ou 2)
4
Exception-Eudiant sur le code inscription 4 dans setCodeIns
Veuillez entrer un code d'inscription (1 ou 2)
1

```

Deuxième exemple d'exécution (mauvaise saisie pour e1, bonne pour e2).

```

3

```

Exception-Eudiant sur le code inscription 3 dans setCodeIns
Veuillez entrer un code d'inscription (1 ou 2)
2

Il est important de noter que jamais le programme ne termine mal, on garde toujours le contrôle.