Programmation - HMIN111M

Notes du cours d'introduction à la programmation en Java

Révision 2019 - Marianne Huchard

Master 1 IPS, BCD, SSV, Physique-info
Université de Montpellier
Faculté des Sciences (FDS)
Département informatique
et
Master 1 Géomatique
AgroParisTech,
Université Paul Valéry,
Université de Montpellier (FDS, Dpt Informatique)

Table des matières

1 Introduction et premiers éléments de Java **3**
- 1.1 La programmation par objets . . . 3
  - 1.1.1 Les grands principes . . . 3
  - 1.1.2 Les principaux langages à objets . . . 4
- 1.2 Le langage Java . . . 4
- 1.3 Quelques formules magiques pour débuter en Java . . . 5
  - 1.3.1 Le programme . . . 5
  - 1.3.2 Les commentaires . . . 5
- 1.4 Types, variables, expressions, instructions simples . . . 6
  - 1.4.1 Le type **booléen** (ou logique) . . . 6
  - 1.4.2 Le type **entier** . . . 6
  - 1.4.3 Le type **réel** . . . 7
  - 1.4.4 Le type **caractère** . . . 8
  - 1.4.5 Le type **chaîne de caractères** . . . 8
- 1.5 Déclaration de variables locales . . . 9
- 1.6 Instructions simples . . . 9
  - 1.6.1 Affectation . . . 9
  - 1.6.2 Saisie de valeur . . . 10
  - 1.6.3 Affichage de valeur . . . 12
- 1.7 Un algorithme simple . . . 13
  - 1.7.1 Somme et produit . . . 13
- 1.8 Un petit pas vers les classes . . . 14
  - 1.8.1 Une classe `Personne` . . . 14
  - 1.8.2 Classes et types . . . 15
  - 1.8.3 Utilisons la classe `Personne` . . . 15
  - 1.8.4 Représentation des associations . . . 19
  - 1.8.5 Initialisations . . . 20
- 1.9 Organisation par paquetages . . . 21

2 Instructions conditionnelles - Méthodes **23**
- 2.1 Instructions conditionnelles . . . 23
  - 2.1.1 Conditionnelle à une possibilité . . . 23
  - 2.1.2 Conditionnelle à deux possibilités (alternative) . . . 25

2.1.3 Conditionnelle à possibilités multiples . . . 27
2.2 Complétons les classes par des méthodes . . . 29
2.2.1 Généralités sur la notion de méthode . . . 29
2.2.2 Encapsulation . . . 34
2.3 Méthodes spéciales . . . 37
2.3.1 Constructeurs . . . 37
2.3.2 Accesseurs . . . 39
2.3.3 La méthode `toString` . . . 41
2.4 La classe `Personne` complète . . . 42

3 Répétitives - Tableaux - `ArrayLists` 45
3.1 Instructions répétitives . . . 45
3.1.1 Répétition contrôlée par un compteur . . . 45
3.1.2 Répétition contrôlée par une condition (forme 1) . . . 47
3.1.3 Répétition contrôlée par une condition (forme 2) . . . 49
3.2 Tableaux . . . 50
3.3 Les classes `Vector` et `ArrayList` . . . 55

4 Généralisation - Spécialisation - Héritage 60
4.1 Généralisation - Spécialisation . . . 60
4.2 Hiérarchie des classes et héritage dans Java . . . 61
4.3 Redéfinition de méthodes - Surcharge - Masquage . . . 64
4.4 Les constructeurs . . . 65
4.5 Protections . . . 67
4.6 Classes et méthodes abstraites . . . 67
4.7 Le polymorphisme . . . 68
4.7.1 Transformation de type ou coercition (casting) . . . 68
4.7.2 Polymorphisme des opérations . . . 69

5 Compléments sur les variables et les opérations 72
5.1 Constantes . . . 72
5.2 Variables et méthodes de classe . . . 73
5.3 Types énumérés . . . 76

6 Récursivité 79
6.0.1 Méthodes de classe récursives . . . 79
6.0.2 Méthodes d'instance récursives sur un même objet . . . 81
6.0.3 Méthodes d'instance récursives sur des objets définis récursivement . . . 84

Partie 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

Vous avez étudié 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 du monde réel. 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 calculs »). 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 approximativement en 1967, avec le langage Simula. Au moins deux grandes familles de langages se distinguent :
— les langages qui étendent un langage impératif ou fonctionnel existant, comme C++ (1985, qui étend C), Clos (construit avec Lisp).
— les langages intégralement développés selon le paradigme objet, tels qu'Eiffel (1988), ou Smalltalk (créé en 1972, où les structures de contrôles sont elles-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 se présente comme 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 « bytecodes » qui correspondent au langage d'une machine abstraite (la machine virtuelle Java),
— la machine abstraite est un *interprèteur* qui traduit à la volée le bytecode dans le langage machine natif de l'ordinateur hôte, 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 débuter 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. Acceptez-le tel quel pour l'instant, nous remplacerons 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` (Notez que c'est aussi le nom de la classe publique incluse dans ce fichier).

```
// Déclare que ce programme appartient à un ensemble de programmes qui se nomme
// Prog.MesExemples (cet ensemble de programmes est un paquetage ou package en anglais)

package Prog.MesExemples;

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

import java.util.Scanner;

// Emballage du programme

public class ExempleDeProgramme
{
  public static void main(String argv[])
  {

            ......................;
            ......................;
            ......................;

   }
}
```

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. **Vrai** s'écrit `true`, **Faux** 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 dans ce cours d'introduction).

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 || d) (résultat booléen) si a, b, c et d 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 puisque $2^{31} =$ 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 en principe 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)
Dans le cas de certains interprèteurs Java francophones, la saisie au clavier se fait maintenant avec la traditionnelle virgule.

Opérations.

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

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

Expressions.
11/3.0 vaut 3.666666666666666

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 :

`\t`	tabulation horizontale
`\n`	saut de ligne
`\"`	guillemets
`\'`	apostrophe

Opérations.

Algorithmique	$<$	$>$	$\leq$	$\geq$	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. Comme il s'agit d'une classe, l'application des opérations suit un schéma un peu différent de celui que nous avons vu pour les autres types.

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\"de" (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 élément 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 élément 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 » en algorithmique.

```
package Prog.MesExemples;
import java.util.Scanner;
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.util.Scanner;
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

La lecture d'une donnée se réalise en commençant par créer un `Scanner`, objet dont le rôle est de représenter le clavier (le flux de données entrant du programme). On s'adresse ensuite à cet objet pour récupérer les valeurs contenues dans ce flux par des méthodes dont le nom commence par `next` et se poursuit par le type de la donnée saisie : `nextInt` pour les entiers, `nextDouble` pour les réels, `nextBoolean` pour les booléens. L'exception est `next` qui sert à la saisie des chaînes de caractères.
Vous pouvez voir dans le programme suivant le procédé utilisé pour lire un entier au clavier.

```
 // declaration d'un objet representant le clavier
Scanner clavier = new Scanner(System.in);
// declaration de l'entier que l'on veut saisir
int i;
// lire clavier i
i = clavier.nextInt();
```

Une utilisation plus complète de la classe `Scanner` est décrite par le programme Java suivant qui saisit des valeurs de divers types.

```
package Prog.MesExemples;
import java.util.Scanner;

public class TestScanner
{
 public static void main(String[] args)
 {
  // declaration d'un objet representant le clavier
  Scanner clavier = new Scanner(System.in);

  // declaration de l'entier que l'on veut saisir
  int i;
  // lire clavier i
  i = clavier.nextInt();
  System.out.println(" controle "+i);

  // saisie d'une chaine de caracteres
  System.out.println("Entrer une chaine");
  String p; p = clavier.next(); // lire clavier p
  System.out.println(" controle "+p);

  // saisie entier
  System.out.println("Entrer un entier ");
  int e; e = clavier.nextInt(); // lire clavier e
  System.out.println(" controle "+e);

  //saisie caractere
System.out.println("Entrer un caractere ");
// lire clavier une chaine e et recuperer le premier caractere
char c; c = clavier.next().charAt(0);
System.out.println(" controle "+c);

  // saisie d'un reel
  System.out.println("Entrer un reel ");
  double d; d = clavier.nextDouble(); // lire clavier d
  System.out.println(" controle "+d+ 1.2);

  // saisie d'un booleen
  System.out.println("Entrer un booleen ");
  boolean b; b = clavier.nextBoolean(); // lire clavier b
  System.out.println(" controle "+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 simplement par l'instruction suivante.

```
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 3.3333333333333335, ce qui est un peu long. Le petit programme suivant vous montre comment réduire cet affichage à 2 décimales :

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

public class saisie
{
  public static void main(String argv[])
  {
      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	écrire écran "entrez deux réels"
/* reçoivent	lire clavier x
les valeurs	lire clavier y
lues */	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 java.util.Scanner;

public class SommeEtProduit
{
  public static void main(String[] arg)
  {
     Scanner clavier = new Scanner(System.in);
      double x, y;
      System.out.println("entrez deux reels ");
      x = clavier.nextDouble();
      y = clavier.nextDouble();
      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 java.util.Scanner;

public class SommeEtProduit
{
  public static void main(String[] arg)
  {
     Scanner clavier = new Scanner(System.in);
      double x, y;
      System.out.println("entrez deux reels ");
      x = clavier.nextDouble();
      y = clavier.nextDouble();
      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, encore appelé structurel (attributs et relations). Nous allons partir d'une classe décrite en UML, étudier sa traduction en Java, créer des objets (également appelés 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 (nom et âge).

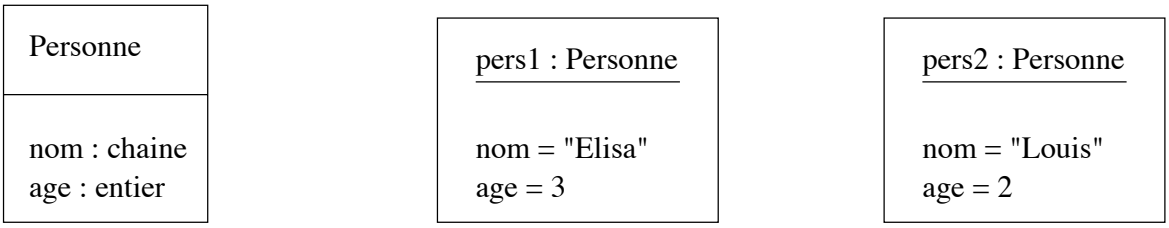

FIGURE 1.1 – Une classe *Personne* et deux instances.

La partie statique de cette classe se traduit syntaxiquement par la classe Java qui suit. Vous devez considérez 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 ! Elle sera complétée dans un prochain chapitre. 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
{
   String nom;
   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 : un type est un ensemble de valeurs muni d'un ensemble d'opérations. Son domaine, c'est-à-dire l'ensemble de ses valeurs, est l'ensemble des instances, appelé aussi *extension* de la classe. Le tuple formé des valeurs des attributs d'une instance est ce que nous appelerons 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, qui sont une description des objets et de leurs capacités, 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();
```

Ce qui donnera pour la classe `Personne` :

```
Personne elisa;
 elisa = new Personne();
```

1.8.3.2 Accès aux attributs

L'accès à un attribut se fait par une notation pointée ou par l'intermédiaire `d'accesseurs`, qui figureront dans la classe et que vous apprendrez à écrire dans un prochain chapitre. Nous vous expliquerons également les avantages de l'utilisation des accesseurs, qui, dans un premier temps, peut paraître plus complexe.

Accès en lecture pour connaître la valeur
`i.a` notation pointée `i.getAtt()` accès par un accesseur où `i` est une instance de la classe `C` qui dispose d'un attribut `att`. La valeur de l'expression est la valeur de l'attribut `att`.

Par exemple, les expressions `elisa.age` et `elisa.getAge()` ont pour valeur l'âge de l'objet `elisa`.

Accès en écriture pour modifier la valeur
`i.att = t` notation pointée `i.setAtt(t)` où `i` est une instance de la classe `C` qui dispose d'un attribut `att` de type T. La valeur `t` de type T est affectée à l'attribut `att`.

Par exemple, les instructions `elisa.age = 23` et `elisa.setAge(23)` affectent 23 à l'âge de l'objet `elisa`.

1.8.3.3 Application à la classe Personne

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

import java.util.Scanner;

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

      Scanner clavier = new Scanner(System.in);

      // 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.next());
      System.out.println("Donnez l'age de la personne");
      p3.setAge(clavier.nextInt());

      // affichage du nom des trois personnes créées
      System.out.println("Voici les noms des personnes creees : "
                         +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 creees : "
                           +((p1.getAge()+p2.getAge()+p3.getAge())/3.0)+"."
                           );

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

Nota. Dans le programme précédent, seuls les accesseurs ont été utilisés. A titre d'exercice, écrivez le même programme sans les utiliser.

1.8.3.4 Représentation schématique des instances

De la même manière que nous faisions 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 `p1` ou `elisa`). 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.

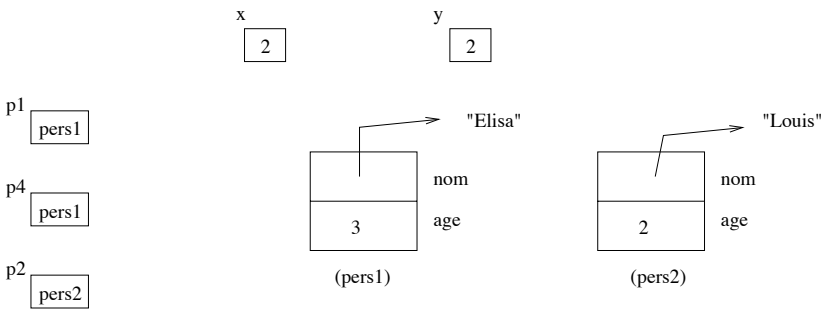

FIGURE 1.2 – Représentation schématique d'instances de *Personne* et de leur état.

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

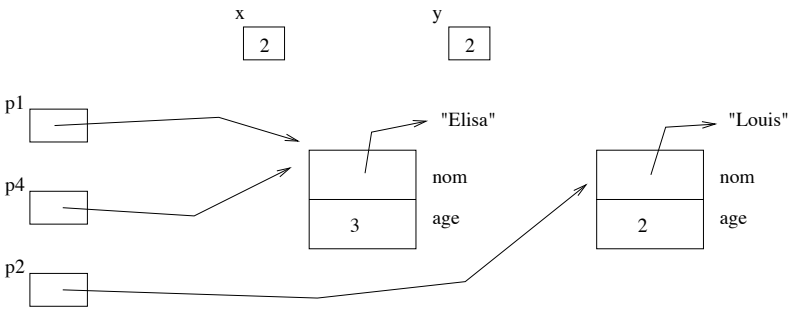

FIGURE 1.3 – Désignations et flèches

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 techniques 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 le fait d'une voiture appartient à 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).

FIGURE 1.4 – Personnes et voitures - diagramme de classes

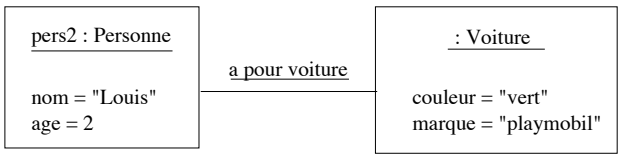

FIGURE 1.5 – Personnes et voitures - diagramme d'instances

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 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.

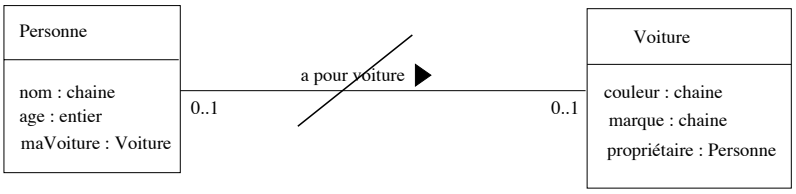

FIGURE 1.6 – Schéma de conception pour « a pour voiture ». L'association « a pour voiture » est supprimée au profit des deux attributs « ma voiture » et « propriétaire » correspondant aux deux noms de rôle. Ces deux attributs doivent être mis à jour de manière homogène.

```
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. Remarquez que nous avons ajouté le mot-clef «private » devant tous les attributs. Cela vous impose d'utiliser uniquement les accesseurs pour accéder aux attributs dans votre programme.

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), et `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 en UML (à gauche) et les répertoires dans le système de fichiers (à droite) correspondant 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 (sinon il faut les préfixer par le nom du paquetage).
Les noms de paquetage servent aussi au compilateur et à l'interpréteur Java à retrouver les classes. S'ils sont mal orthographiés, cela peut ainsi causer des erreurs de compilation ou d'exécution.

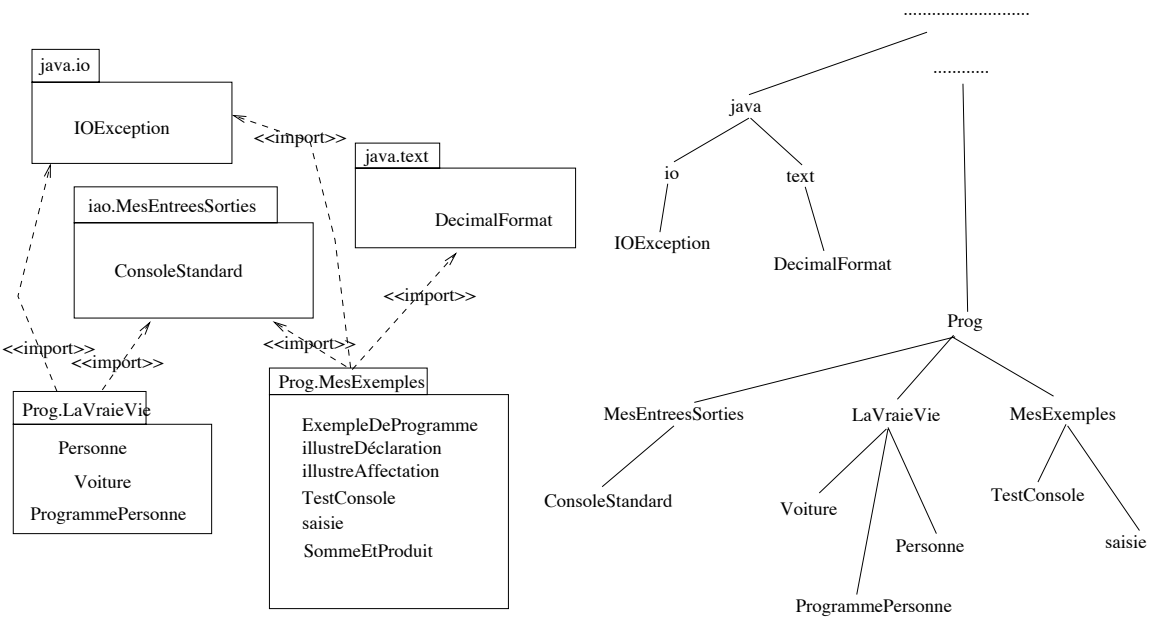

FIGURE 1.7 – Paquetages

Partie 2

Instructions conditionnelles - Méthodes

2.1 Instructions conditionnelles

2.1.1 Conditionnelle à une possibilité

Syntaxe algorithmique.

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

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

Syntaxe Java.

`if` C $\{ A_1; A_2; \ldots; A_n; \}$	`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 a, b	écrire écran "entrez deux entiers"
les entiers lus	lire clavier a, b
	`si` $a > b$ `alors`
	/* échange des valeurs */
entier vi	$vi \leftarrow a$
sert pour	$a \leftarrow b$
l'échange	$b \leftarrow vi$
	`fsi`
	écrire écran a, ' ', b, '\n'

```
package Prog.MesExemples;
import java.util.Scanner;

public class EcrireOrdre1
{
  public static void main(String argv[])
  {
       // variables locales
        int a,b,vi;
        Scanner clavier = new Scanner(System.in);

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

        if (a>b)
            {
                vi=a;
                a=b;
                b=vi;
            }
        System.out.println(a+" "+b);
  }  // fin de la partie "main"
}    // fin de la classe
```

2.1.2 Conditionnelle à deux possibilités (alternative)

Syntaxe Algorithmique.

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

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

Syntaxe Java.

if C
   { $A_1; A_2; \ldots; A_n$; }
else
   { $B_1; B_2; \ldots; B_n$; }
	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 a, b
les entiers lus
	écrire écran "entrez deux entiers"
	lire clavier a, b
	
	si $a > b$ alors
	   écrire écran b,' ',a,'\n'
	sinon
	   écrire écran a,' ',b,'\n'
	fsi

```
package Prog.MesExemples;
import java.util.Scanner;

public class EcrireOrdre2
{
  public static void main(String argv[])
{
        // variables locales
        int a,b;
        Scanner clavier = new Scanner(System.in);

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

        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
   v1 : A11; A12; ...; A1n1
   v2 : A21; A22; ...; A2n2
   .......
   vm : Am1; Am2; ...; Amnm
   autrement : B1; B2; ...; Bn
fcas
```

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

Syntaxe Java.

```
switch(E)
{
   case v1 : A11; A12; ...; A1n1; break;
   case v2 : A21; A22; ...; A2n2; break;
   .......
   case vm : Am1; Am2; ...; Amnm; break;
   default : B1; B2; ...; Bn ;
}
```

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 *ReconnaîtVoyelles*
Données : *un caractère lu au clavier*
Résultats : *affiche un message différenciant voyelles minuscules et voyelles majuscules des autres caractères*

Objets utilisés	**Instructions**

`caractère` c
lu au clavier
	écrire écran "Entrez un caractère"
	
	lire clavier c
	`cas où` c `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 java.util.Scanner;

public class ReconnaitVoyelles
{
  public static void main(String argv[])
  {
        char c;
        Scanner clavier = new Scanner(System.in);
        System.out.println("Entrez un caractère");
        c = clavier.next().charAt(0);
        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 des opérations pour les classes. Ces 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 (on écrira une expression telle que `inst.m(...)` avec `inst` une instance et `m` une méthode), 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 avec (manipuler) les attributs de l'instance sous réserve des mécanismes de protection dont nous parlons plus loin. La méthode s'exécute dans le contexte de l'instance concernée.
- Des classes différentes peuvent proposer des méthodes portant le même nom (ex : la plupart des classes possèdent une méthode nommée `toString`). 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 listes de paramètres soient différentes (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 pers1`
`qui désigne une valeur de Personne`

Fonction	*Méthode*
pers1 est un paramètre	on applique l'opération à l'instance
estMajeure(pers1)	pers1.estMajeure()

```
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 `this`, comme en Java.

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 (this.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).

protection typeRetour nomMéthode($typeParam_1$ $identParam_1$, ... $typeParam_n$ $identParam_n$)
{
.....
}

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,
- le *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, avec ou sans protection explicite. Dans la dernière méthode, nous illustrons l'utilisation de la variable spéciale `this`.

```
package Prog.LaVraieVie;
import java.util.*;

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

```
// 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;}

// autres méthodes
public  void saisirAge()
    {
        System.out.println("Veuillez entrer l'age de (d') "+nom);
        Scanner clavier = new Scanner(System.in);
        age = clavier.nextInt();
    }

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


    public boolean estMajeure()
    {
        return getAge() > 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;
public class ProgrammePersonne2
{
  public static void main(String argv[])
  {
      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 meme age ? "+
                         pers1.aMemeAge(pers2));
  }  // fin de la partie "main"
}    // fin de la classe
```

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, jour, heure, minute, 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-clefs 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;
public class ProgrammePersonne
{
  public static void main(String argv[])
  {
      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.*;

public class TestPersonne
{
  public static void main(String argv[])
    {
     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 privés, et les méthodes généralement publiques (mais nous verrons des

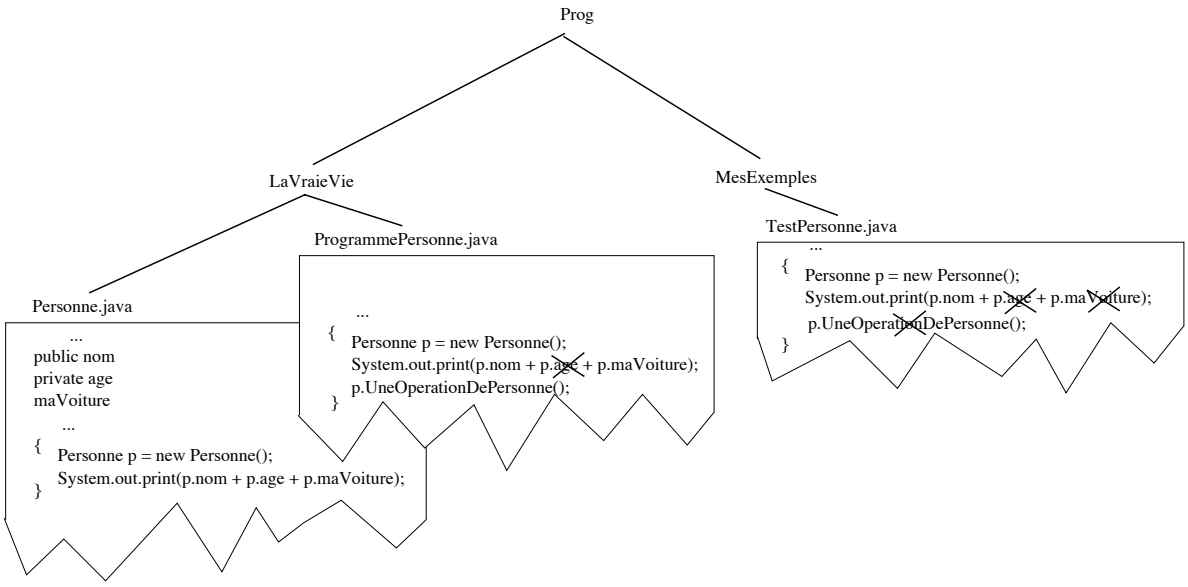

FIGURE 2.1 – Protection

variantes sur ce deuxième point). Pour accéder aux attributs dans une méthode, si cette méthode est hors de leur classe, mais parfois aussi si elle est dans leur classe, on utilisera donc préférentiellement 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*.
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;
public class ProgrammePersonne
{
  public static void main(String argv[])
  {
      Personne pers1 = new Personne();

      Personne pers2 = new Personne("Louis");

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

Un constructeur peut en appeler un autre. Par exemple, le troisième constructeur peut être réécrit de la manière qui suit, en utilisant la pseudo-variable `this` et en appelant le second constructeur. L'appel d'un constructeur dans un autre doit toujours être effectué au début (comme première instruction).

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

   (.......)

    public Personne(String n) // rappel du second constructeur
    {nom = n; age = -1;}

    public Personne(String n, int a) // nouvelle forme du troisième constructeur
    {this(n); age = a;}

    (.........)
..........
}
```

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.util.*;

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()
    {
        System.out.println("Veuillez entrer l'age de (d') "+nom);
        Scanner clavier = new Scanner(System.in);
        age = clavier.nextInt();
    }

    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 java.util.*;

public class ProgrammePersonne3
{
  public static void main(String argv[])
  {
      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 :

```
Veuillez entrer l'age de (d') Elisa
12
annee de naissance de (d') Elisa = 1997
Elisa est-elle majeure ? false
Veuillez entrer l'age de (d') Louis
1
Elisa et Louis ont le même âge ? false
Elisa 12  Louis 1  Eloise 2
```

Partie 3

Répétitives - Tableaux - `ArrayLists`

3.1 Instructions répétitives

3.1.1 Répétition contrôlée par un compteur

Syntaxe algorithmique.

`pour` $(vc \leftarrow E\,;\, C\,;\, I)$ `faire`
$\quad A_1; A_2; \ldots; A_n$
`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 décrémenter le compteur,
$A_1; A_2; \ldots; A_n$ sont des instructions.

Syntaxe Java.

`for` $(vc = E\,;\, C\,;\, I)$ $\quad \{A_1; A_2; \ldots; A_n;\}$	`for` $(vc = E\,;\, C\,;\, I)$ $\quad$ 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 .... *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	écrire écran "Entrez un entier $y \geq 0$"
vaut x^v après l'étape v	lire clavier y
	$res \leftarrow 1$
	`pour` $(v \leftarrow 1; v \leq y; v \leftarrow v + 1)$
	`faire`
`entier` v	$res \leftarrow res * x$
var. de contrôle	`fpour`
	écrire écran "x^y vaut ", res

```
package Prog.MesExemples;
import java.util.Scanner;

public class PuissanceEntiere1
{
  public static void main(String argv[])
  {
      int x, y, res, v;
      Scanner clavier = new Scanner(System.in);

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

      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`
$A_1; A_2; \ldots; A_n$
`ftant`

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

Syntaxe Java.

`while` (C) $\{A_1; A_2; \ldots; A_n;\}$	`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 .... *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 écrire écran "Entrez un entier $y \geq 0$" lire clavier y	
entier *res* vaut x^i à l'étape i	$res \leftarrow 1$ $v \leftarrow 1$ `tant que` $(v \leq y)$ `faire` $\quad res \leftarrow res * x$	
entier v var. de contrôle	$\quad v \leftarrow v + 1$ `ftant` écrire écran "x^y vaut ", res	

```
package Prog.MesExemples;
import java.util.Scanner;
public class PuissanceEntiere2
{
  public static void main(String argv[]) throws java.io.IOException
  {
      int x, y, res, v;
      Scanner clavier = new Scanner(System.in);
      System.out.println("entrez un entier > 0 ");
      x = clavier.nextInt();
      System.out.println("entrez un entier >= 0 ");
      y = clavier.nextInt();
      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.

`faire` $A_1; A_2; \ldots; A_n$ `tant que` C où C est une condition, $A_1; A_2; \ldots; A_n$ sont des instructions

Syntaxe Java.

`do` $\{A_1; A_2; \ldots; A_n;\}$ `while` (C) ;	`do` A; `while` (C) ;

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 :

`entier` x	`faire` écrire écran "Entrez un entier $x >$ 0" lire clavier x `tant que` $(x \leq 0)$	

Ce qui se traduira en Java par :

```
do
{
    System.out.println("entrez un entier > 0 ");
    x = clavier.nextInt();
}
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` où T est un type	`T[]  monTableau;`

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

```
monTableau = new T[n];
```

où `monTableau` est une variable de type `T[]`

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

```
monTableau[i]
```

où `monTableau` est une variable de type `T[n]`,
et `i` est un entier compris entre 0 et `n-1`.
Cette expression est de type `T`.

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.
	$l \leftarrow$ 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'écrirons 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 java.util.Scanner;
public class InverseTableaux
{
    public static void main(String[] arg)
    {
        Scanner clavier = new Scanner(System.in);
        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.nextInt();
            }

        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).

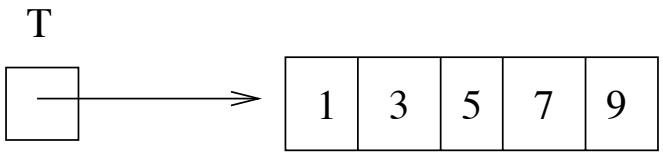

FIGURE 3.1 – Un tableau d'entiers.

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 java.util.Scanner;
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(Scanner cons)
   {
    System.out.println("Saisie des données d'un voyageur \nEntrer son nom : ");
    nom = cons.next();
    System.out.println("Entrer son age : ");
    age = cons.nextInt();
    System.out.println("Entrer un caractère pour le sexe (M ou F) : ");
    sexe = cons.next().charAt(0);
   }
```

```
     public boolean carteVermeil()
     // nota : une version plus simple avec une unique expression booléenne est vue en TD/T
     {
        if (age >= 65) return true;
        if  ((age >= 60) && (sexe == 'F')) return true;
        return false;
     }
}
```

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

```
package Prog.Voyage;
import java.util.Scanner;
public class AppliVoyageurTableau
{
  public static void main(String[] argv)
  {
    Scanner cons = new Scanner(System.in);

    // 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.nextInt();
    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).

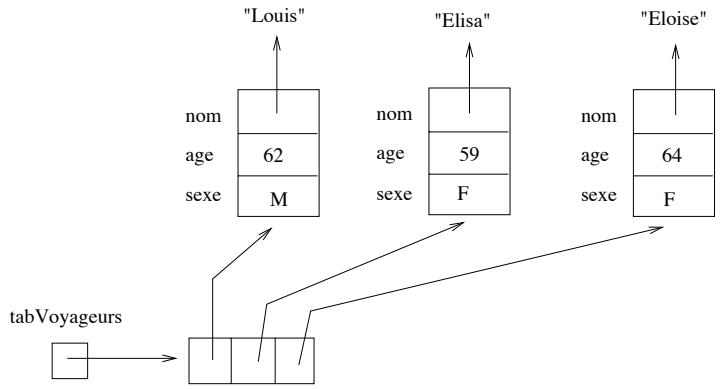

FIGURE 3.2 – Un tableau de voyageurs.

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 Les classes `Vector` et `ArrayList`

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 deux classes (`Vector` et `ArrayList`) 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.

Les `Vector` et les `ArrayList` proposent un même ensemble d'opérations, en ce qui concerne l'usage que nous en ferons. La différence entre les deux classes est que l'accès aux `Vector` est synchronisé : ils sont donc préférables si vous êtes amenés à programmer plusieurs processus (threads) faisant accès à un même `Vector`. Cette synchronisation ayant un coût en temps, les `ArrayList` fonctionnent de manière plus efficace et sont actuellement préférées pour les programmes ordinaires.
Comparativement aux tableaux, ces deux classes présentent néanmoins un inconvénient : seules des instances peuvent être stockées dans les `ArrayList`s. 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 dans ce cours.
Nous allons regarder certaines des méthodes de la classe `ArrayList`, parmi les plus utiles. Nous les illustrons en écrivant progressivement un programme à peu près équivalent à `AppliVoyageurTableau`.

Déclaration d'une `ArrayList`

C'est une instance de la classe `ArrayList`. Il suffit donc de la déclarer. Vous noterez que derrière le nom de la classe `ArrayList` se trouve placé (entre < et >) le type des éléments stockés dans la structure (ici des voyageurs).

```
ArrayList<Voyageur> vectVoyageurs;
```

Création d'une `ArrayList`

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

```
vectVoyageurs = new ArrayList<Voyageur>();
```

Taille d'une `ArrayList`

A tout moment après sa création, on peut consulter la taille de l'`ArrayList` 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 add(v)`. Le nouvel élément est ajouté à la fin de l'`ArrayList`.

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

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

Le schéma de la figure 3.2 est valable également, en première approximation, pour se représenter l'`ArrayList`.

Consultation (accès) d'un élément

L'accès à l'élément d'indice `i` s'effectue grâce à la méthode `get(int i)` qui retourne l'élément. Lorsque l'on applique cette méthode à une `ArrayList<Voyageur>`, l'expression résultante, par exemple

```
vectVoyageurs.get(i)
```

est une valeur du type `Voyageur`. On peut donc lui appliquer une méthode de la classe `Voyageur` telle que :

```
vectVoyageurs.get(i).carteVermeil()
```

De manière générale, si `a` est une `ArrayList<T>`, `a.get(i)` est de type `T`.
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 = vectVoyageurs.get(i);
    System.out.println(voyageurCourant.getNom()
                       +" a droit a la carte vermeil ? "
                       +voyageurCourant.carteVermeil()+"\n");
  }
```

Il existe également une itération `for` d'une forme adaptée au parcours complet de collections (tableaux, ArrayList notamment). Cette forme introduit une variable que l'on appelle un itérateur (ci-dessous la variable `voyageurCourant`). Le type de la variable est le type des éléments de la collection (ci-dessous le type `Voyageur`). La variable itérateur reçoit successivement tous les éléments de la collection et leur applique le traitement prévu dans le corps de l'itération.

```
//Affichage des informations relatives à la carte vermeil
for (Voyageur voyageurCourant : vectVoyageurs)
  {
     System.out.println(voyageurCourant.getNom()
                       +" a droit a la carte vermeil ? "
                       +voyageurCourant.carteVermeil()+"\n");
  }
```

Chaîne de caractères associée à une `ArrayList`

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 de la `ArrayList` : 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 la liste de voyageurs \n"+vectVoyageurs);
```

La chaîne associée à la `ArrayList` par la méthode `toString` prédéfinie est construite en concaténant un crochet ouvrant, le résultat de l'appel de `toString` sur chaque élément (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 de la `ArrayList` (correspondant aux valeurs de la figure 3.2) peut être :

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

Nous vous présentons à présent le programme complet. Notez l'import (nécessaire) de la classe `ArrayList` qui est dans le paquetage `java.util`.

```
package Prog.MesExemples;
import java.util.Scanner;
import java.util.ArrayList;
public class AppliVoyageurArrayList
{
  public static void main(String[] argv)
  {
    Scanner cons = new Scanner(System.in);

    // Déclaration et creation d'un tableau de voyageurs
    ArrayList<Voyageur> vectVoyageurs=new ArrayList<Voyageur>();

    // Saisie du nombre de voyageurs
```

```
    System.out.println("Combien de voyageurs désirez-vous créer ?");
    int nombreVoyageurs = cons.nextInt();

    // Création et saisie des données des voyageurs
    for (int i=0; i<nombreVoyageurs; i=i+1)
      {
        Voyageur v = new Voyageur();
        v.saisie(cons);
        vectVoyageurs.add(v);
      }

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

Et pour aller plus loin.

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

Partie 4

Généralisation - Spécialisation - Héritage

4.1 Généralisation - Spécialisation

Nous abordons dans ce chapitre, un des constituants majeurs et spécifique de la programmation par objets. Il 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 classer les concepts manipulés dans des classifications plus ou moins complexes, appelées hiérarchies d'héritage. Ce sont ces classifications et les mécanismes qui leur sont liés qui donnent au développement par objets ses qualités de réutilisabilité et d'extensibilité.
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 montrant qu'elles partagent des attributs et des opérations, on en extrait (par un mécanisme d'abstraction) une super-classe `Véhicule` qui regroupe les éléments communs aux deux sous-classes `Voiture` et `Bateau`. Les deux sous-classes héritent des caractéristiques communes définies dans leur super-classe `Véhicule` : `nom`, `couleur`, `puissance` et la capacité à `avancer`. Elles déclarent en plus les caractéristiques qui les distinguent (`chavirer`, `crever`) ou bien elles redéfinissent selon leur propre point de vue une ou plusieurs caractéristiques communes (chacune a une procédure spécifique pour `avancer`).
Le mécanisme dual de 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-

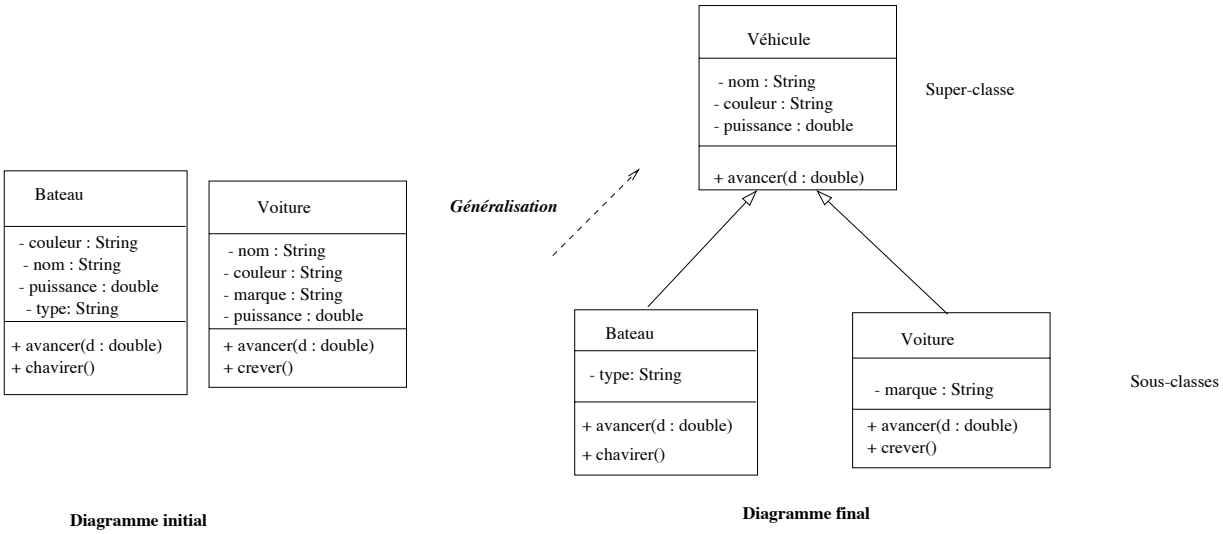

FIGURE 4.1 – Une généralisation

généralisation.

4.2 Hiérarchie des classes et héritage dans Java

Pour les langages à objets, le programmeur définit des hiérarchies de classes provenant d'une conception dans laquelle il a utilisé le principe de généralisation-spécialisation. On dit le plus souvent que la sous-classe hérite de la super-classe, ou qu'elle étend la super-classe ou encore qu'elle dérive de la super-classe.
Un des avantages des langages à objets est que le code est réutilisable. Il est commode de construire de gros projets informatiques en étendant des classes déjà testées. Le code produit devrait être plus lisible et plus robuste car on peut contrôler plus facilement son évolution, au fur et à mesure de l'avancement du projet. En effet, grâce à la factorisation introduite par la spécialisation-généralisation, on peut modifier une ou plusieurs classes sans avoir à les réécrire complètement (par exemple en modifiant le code de leur super-classe).
Du point de vue des objets (instances de classes), pour Java, une instance d'une sous-classe possède la partie structurelle définie dans la superclasse de sa classe génitrice plus la partie structurelle définie dans celle-ci. Au niveau du comportement, les objets d'une sous-classe héritent du comportement défini dans la superclasse (ou la hiérarchie de super-classes) avec quelques possibilité de variations, comme nous le verrons plus tard. Au niveau de l'exécution, les langages à objets utilisent un mécanisme d'héritage (ou résolution de messages) qui consiste à résoudre dynamiquement l'envoi de message sur les objets (instances), c'est-à-dire à trouver et exécuter le code le plus spécifique correspondant au message.
En Java,

— toutes les classes dérivent de la classe `Object`, qui est la racine de toute hiérarchie de classes.

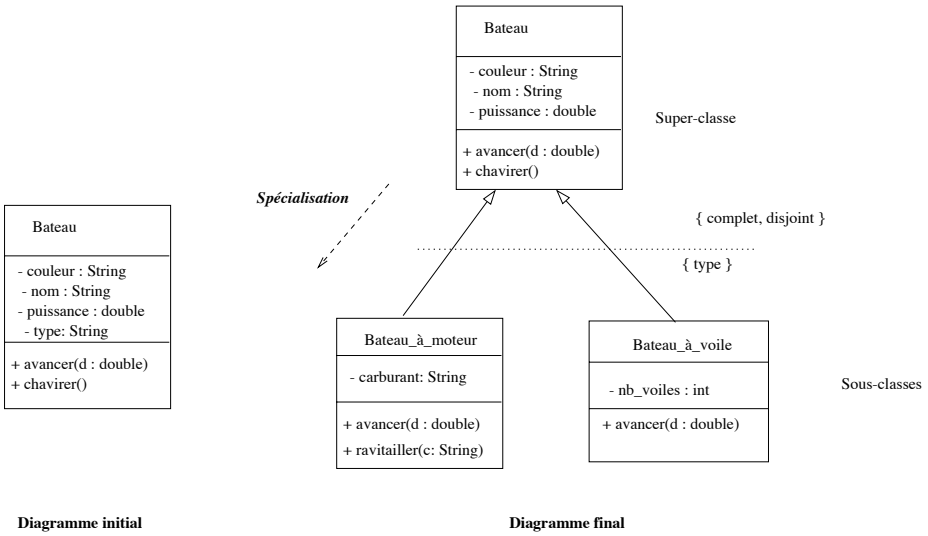

FIGURE 4.2 – Une spécialisation.

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

Pour la suite du chapitre nous prenons une hiérarchie de classes représentée dans le diagramme 4.3.
Nous commençons par définir la structure de la hiérarchie : vous voyez comment les entêtes des classes incluent le mot clef `extends` pour traduire en Java la relation d'héritage.

```
package Prog.MesExemples;

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

.....
public class Etudiant extends Personne {


      ..................


}    // fin de la classe Etudiant

.............
public class Etud_Licence extends Etudiant {
```

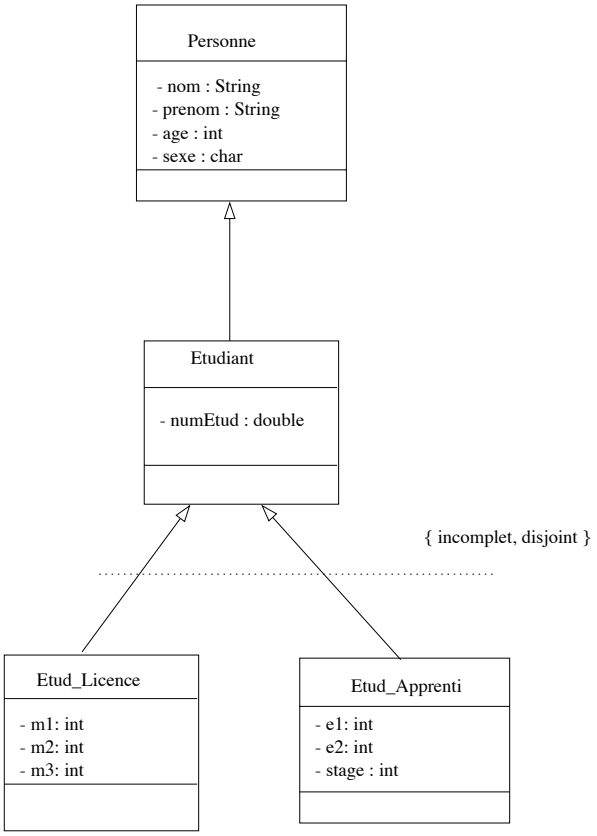

FIGURE 4.3 – Une hiérarchie de classes.

```
      ..................

}    // 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).
La classe `Etudiant` possède l'attribut supplémentaire `numEtud` qui représente le numéro de l'étudiant. La classe `Etudiant` définit des opérations de calculs de moyenne, d'admission, de mention.
Les étudiants de licence (classe `Etud_Licence`) suivent 3 modules (M1, M2, M3) de coefficient 1. Ils sont admis si leur moyenne générale est supérieure ou égale à 10 et ils conservent les modules obtenus s'ils ne sont pas admis.
Les étudiants apprentis (classe `Etud_Apprenti`) suivent 2 modules (E1, E2) de coefficient 1 et un stage de coefficient 2. Ils sont admis si leur moyenne est supérieure ou égale à 10 et si leur note de stage est supérieure à 8.
Une instance étudiant apprenti (ainsi qu'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()+"était âgé de " + a.getAge()+" ans");
```

A l'exécution, on obtient :

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

4.3 Redéfinition de méthodes - Surcharge - Masquage

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

- au sein d'une même classe donner le même nom à deux méthodes si la liste des paramètres est différente,
- dans des classes distinctes, donner le même nom et la même liste de paramètres à deux méthodes. 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 souvent de *surcharge* (plusieurs méthodes de même nom). 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* (deux méthodes de même nom et avec une même liste de types de paramètres, l'une dans une classe, l'autre dans sa sous-classe). Donc lorsque l'on code une sous-classe, on a la possibilité de masquer la définition d'une (ou de plusieurs) méthode(s) de la super-classe, simplement en redéfinissant une méthode qui a le même nom et les mêmes paramètres que celle de la super-classe.

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

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

// Classe Etud_Apprenti

public class Etud_Apprenti extends Etudiant {

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

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

4.4 Les constructeurs

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

```
// la classe Personne

public Class Personne{
.............
// constructeur par défaut
public Personne() {
```

```
nom = "";
prenom = "";
age = 0;
sexe = 'F';}
// autre constructeur
public Personne(String n, String p, int a, char s) {
nom = n;
prenom = p;
age = a;
sexe = s;}
// la classe Etudiant
public Class Etudiant extends Personne{
.............
// constructeur par défaut
public Etudiant()
       {
       // super() sera automatiquement réalisé
        numEtu=-1;
        }

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


// la classe Etu_Apprenti
public Class Etu_Apprenti extends Etudiant{
// constructeur par défaut
public Etu_Apprenti ()
       {
       // Par défaut, super() est appelée.
       E1=-1; E2=-1; stage=-1;
       }
// autre constructeur
public Etu_Apprenti (String n, String p, int a, char s, double n, int e1, int e2, int st)
        {
        super(n, p, a, s, n); // appel au 2ième constructeur de Etudiant
        E1 = e1; E2 = e2; stage = st;
      }
```

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

4.5 Protections

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

- `public` : la méthode (ou l'attribut) est accessible par tous et de n'importe où.
- `protected` : la méthode (ou l'attribut) n'est accessible que par les classes du même package et par les sous-classes (même si elles se trouvent dans un autre package). *Remarque :* pour le contrôle d'accès `protected`, il semblait logique de n'autoriser l'accès qu'à la classe concernée et à ses sous classes (comme en C++) mais ce n'est pas le choix qui a été fait en Java.
- `private` : la méthode (ou l'attribut) est accessible uniquement par les méthodes de la classe. *Remarque :* Cependant les instances d'une même classe ont accès aux méthodes et attributs privés des autres instances de cette classe.

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

4.6 Classes et méthodes abstraites

Dans une hiérarchie de classes, plus une classe occupe un rang élevé, plus elle est générale donc plus elle est abstraite. On peut donc envisager de l'abstraire complètement en lui ôtant d'une part le rôle de génitrice (elle ne sera pas autorisée à créer des instances) et en lui permettant d'autre part de factoriser structure et possibilité d'avoir un certain comportement (sans savoir exactement comment ce comportement sera implémenté) uniquement pour rendre service à sa sous-hiérarchie.
Une méthode *abstraite* est une méthode déclarée avec le mot clef `abstract` et qui ne possède pas de corps (pas de définition de code). Par exemple, le calcul de la moyenne d'un étudiant ne peut pas être décrit au niveau de la classe `Etudiant`.
Néanmoins, on sait que les étudiants de licence et les étudiants apprentis doivent être capable de calculer leur moyenne. La méthode `moyenne` doit être déclarée abstraite au niveau de la classe `Etudiant`. Par contre, il faudra obligatoirement définir le corps de la méthode `moyenne` dans les sous-classes de la classe `Etudiant`.
Si une classe contient au moins une méthode abstraite, alors il faut déclarer cette classe abstraite, mais on peut aussi définir des classes abstraites (parce qu'on ne veut pas qu'elles engendrent des objets) sans aucune méthode abstraite.

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

.........
```

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

Remarques : Dans le formalisme UML les classes et méthodes abstraites ont leur nom en italique ou précédé du stéréotype « `abstract` ».
Une classe abstraite peut être sous-classe d'une classe concrète (ici `Etudiant` est une sous-classe abstraite de `Personne` qui est concrète).
L'intérêt de définir une méthode abstraite est double : il permet au développeur de ne pas oublier de redéfinir une méthode qui a été définie `abstract` au niveau d'une des super-classes ; il permet de mettre en œuvre le polymorphisme.

4.7 Le polymorphisme

4.7.1 Transformation de type ou coercition (casting)

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

```
Personne p;
Etud_Licence e=new Etud_Licence();
p=e;
// tout étudiant de licence est un étudiant et a fortiori une personne
```

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

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

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

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

4.7.2 Polymorphisme des opérations

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

```
\\une autre hiérarchie de classes

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

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

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

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

Remarquer ici, que le main utilise un tableau de Felin, que le casting implicite est utilisé, et que tab[i] étant un Felin pour le compilateur et la méthode pousseSonCri étant définie dans

Felin (sous forme abstraite), il n'y a pas d'erreur de compilation et à l'exécution, on obtient :

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

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

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

public String nomDesEtudiants() {
       String listeNom = "";
// Le polymorphisme a lieu ici: la méthode toString() est appelée sur tous les
// elements  de la ArrayList listeEtudiants.

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

       return listeNom;
   }
 } // fin classe Promotion



public class AppliPromo {
    public static void main(String args[]) {
    Promotion promo = new Promotion (2000);
```

```
    Etud_Licence e1 = new Etud_Licence("Cesar", "Julio", 26, 'M', 127, 14, 12, 10);
    promo.inscrit (e1);
    Etud_Licence e2 = new Etud_Licence("Curie", "Marie", 22, 'F', 124, 8, 17, 20);
    promo.inscrit (e2);
    Etud_Apprenti a1 = new Etud_Apprenti ("Bol", "Gemoï", 22, 'M', 624, 8, 8, 10);
    promo.inscrit (a1);
    Etud_apprenti a2 = new Etud_Apprenti ("Einstein", "Albert", 22, 'M', 123, 13, 17, 17);
    promo.inscrit (a2);

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

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

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

En général, il est préférable de ne pas avoir à recourir à la transformation de type explicite. Dans notre exemple Etudiant, il n'est pas nécessaire de caster les étudiants d'une liste d'étudiants en Etud_Apprenti ou Etud_Licence pour calculer leur moyenne().
Pour faire court, le polymorphisme fonctionne grâce à la liaison dynamique qui cherche à l'exécution la méthode la plus spécifique pour résoudre un envoi de message. Cette méthode plus spécifique se trouve soit dans la classe de l'instance (classe déterminée par le type dynamique suivant le `new`), soit dans une super-classe, en remontant à partir de la classe de l'instance et en utilisant la première méthode trouvée répondant au message.

Partie 5

Compléments sur les variables et les opérations

5.1 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, variales 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;

  }
```

5.2 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. Par exemple chaque personne possède son propre nom et son propre âge.
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 (il n'est pas propre à un carré mais partagé),
- le président de la république pour les français (il n'est pas propre à un français mais partagé),
- 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 dans une banque (qui peut servir à attribuer un numéro à un compte).

On ne recommande pas d'abuser de l'usage de ces variables de classes en modélisation. En effet 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`. Ou encore le président de la république serait une variables d'instance d'une classe République. 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("Vincent Auriol");

    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`.

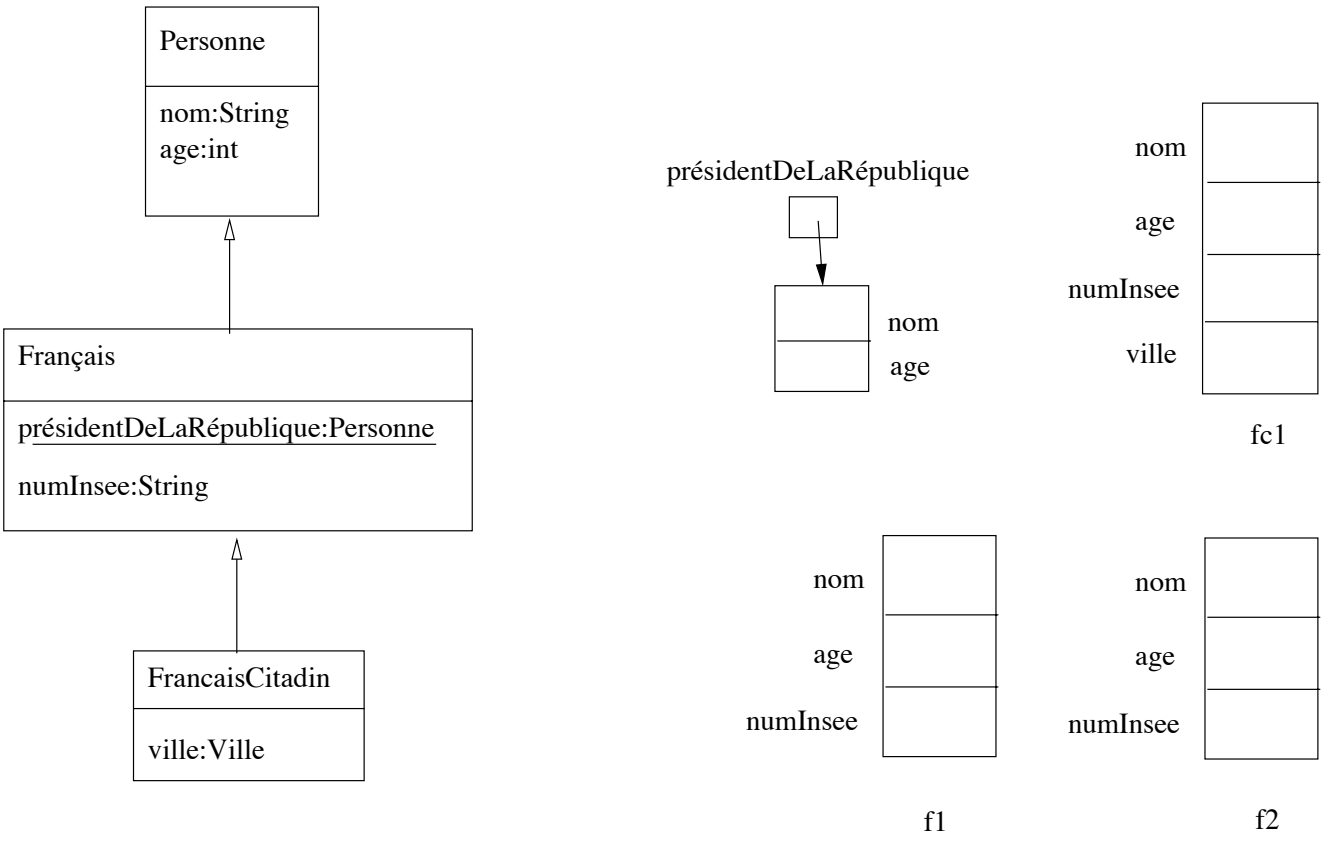

FIGURE 5.1 – Variation sur la classe `Francais`.

— 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.

5.3 Types énumérés

Un type énuméré (introduit par le mot clef `enum`) définit ses propres valeurs littérales ainsi que plusieurs opérations pour faciliter leur utilisation.
Une variable de ce type prend obligatoirement l'une des valeurs littérales. Voici deux exemples typiques, avec deux énumérations permettant de représenter respectivement les jours de la semaine et les points cardinaux :

```
public enum JoursDeLaSemaine{
lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche
}
```

```
public enum PointCardinal{
nord, sud, est, ouest
}
```

L'intérêt d'un type énuméré est de définir un nombre limité de valeurs possibles pour cette variable. Le revers est que l'ensemble de valeurs n'est pas extensible une fois défini, sans modifier et re-compiler le programme. Si on prévoit qu'un ensemble de valeurs sera extensible, il est préférable de mettre les valeurs dans une liste qui pourra augmenter lors de l'exécution, ou réifier en créant une classe dont les instances seront les valeurs du type.
Un type énuméré dérive implicitement de la classe `Enum` qui dérive elle-même de la classe `Object`. Par ces deux classes, il dispose de méthodes, comme `equals` et `compareTo` (pour comparer deux valeurs d'une énumération) ou `toString` (pour obtenir la valeur sous forme d'une chaîne de caractères).
D'autres méthodes sont intéressantes à connaître :

- `valueOf(String s)` retourne la valeur du type énuméré dont l'identifiant est s. Par exemple `JoursDeLaSemaine.valueOf("lundi")` a pour valeur `lundi` (la valeur et non la chaîne de caractères).
- `values()` retourne un tableau contenant toutes les valeurs.
  Par exemple `System.out.println(Arrays.toString(JoursDeLaSemaine.values()));` afficherait les valeurs du type énuméré `JoursDeLaSemaine`.

Un type énuméré étant une classe, il peut contenir des attributs, constructeurs (obligatoirement de visibilité de niveau package ou privé) et des méthodes.
Dans l'exemple suivant, les points cardinaux sont décrits par un attribut donnant la liste de leurs synonymes, un constructeur pour initialiser la liste de synonymes, une méthode pour ajouter un synonyme et une méthode `PointCardinal nomMajeur(String)` de manière à donner, pour une chaîne de caractères supposée être un nom de point cardinal ou un de ses synonymes passée en paramètre, le nom majeur du point cardinal. Si cette chaîne n'est pas un nom de point cardinal ni un synonyme, la méthode retourne la valeur `null`.

```
public enum PointCardinal{

     nord ("septentrion"),
     sud ("midi méridien"),
     est ("orient levant"),
     ouest ("occident couchant ponant");

     private ArrayList<String> synonymes;

     PointCardinal(String syno) {
          String mots[] = syno.split(" ");
          synonymes = new ArrayList(Arrays.asList(mots));
     }

     static PointCardinal nomMajeur(String nom) {
```

```
          for (PointCardinal pt : PointCardinal.values())
               if (nom.equals(pt.toString())
                         || pt.synonymes.contains(nom))
               return pt;
               return null;
     }

     public void ajouteSynonyme(String syno) {
          synonymes.add(syno);
      }
}

....

public static void main(String[] args) {
     System.out.println(PointCardinal.nomMajeur("septentrion"));
     System.out.println(PointCardinal.nomMajeur("méridien"));
     System.out.println(PointCardinal.nomMajeur("soleil"));
     PointCardinal.nord.ajouteSynonyme("borée");
     System.out.println("borée "+PointCardinal.nomMajeur("borée"));
}
```

Partie 6

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 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.

6.0.1 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)
  {
    Scanner c = new Scanner(System.in);

    System.out.println("Entrer un nombre >=0");
    int n=c.nextInt();
    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)
  {
    Scanner c = new Scanner(System.in);

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

6.0.2 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;
   private int codeIns;

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

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

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

  public void saisieCodeIns(Scanner c)
  {
    System.out.println("Veuillez entrer un code d'inscription (1 ou 2)");
    codeIns=c.nextInt();
    if ((codeIns!=1) && (codeIns!=2))
      saisieCodeIns(c);
  }
}
```

Cette méthode peut être appelée ainsi.

```
.....
  public static void main(String[] arg)
      {
      Scanner c = new Scanner(System.in);
      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 l'`ArrayList`.
- 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 de l'`ArrayList`.

La méthode de recherche principale appelle la méthode secondaire en partant à l'indice 0. Si l'`ArrayList` 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 ArrayList<Etudiant> le = new ArrayList<Etudiant>();

  public int nbEtudiants() {return le.size();}
  public void inscrit (Etudiant e) {le.add(e);}
  public Etudiant etudiant(int i) {return (Etudiant)(le.get(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)
    {
      Scanner c = new Scanner(System.in);

      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 de l'`ArrayList`, au moins abstraitement. Selon cette définition, une `ArrayList` est soit vide, soit un (premier) élément suivi d'une autre `ArrayList`.

6.0.3 Méthodes d'instance récursives sur des objets définis récursivement

C'est une variante de la récursivité spécifique des objets lorsque le diagramme de classes présente un circuit de composition (figure 6.1). 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.

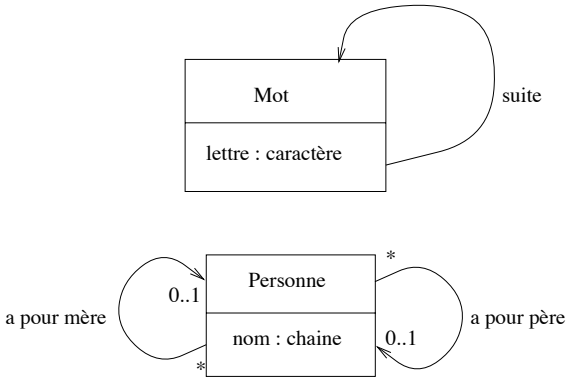

FIGURE 6.1 – Définitions récursives de classes.

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.

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)
   {// on sait que this est différent de null
    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 6.2.
Voici le résultat de l'exécution du programme.

```
Zoé
        Isabelle
                Denise
                Francois
                        Edmée
        Bernard
                Josette
                Henri
```

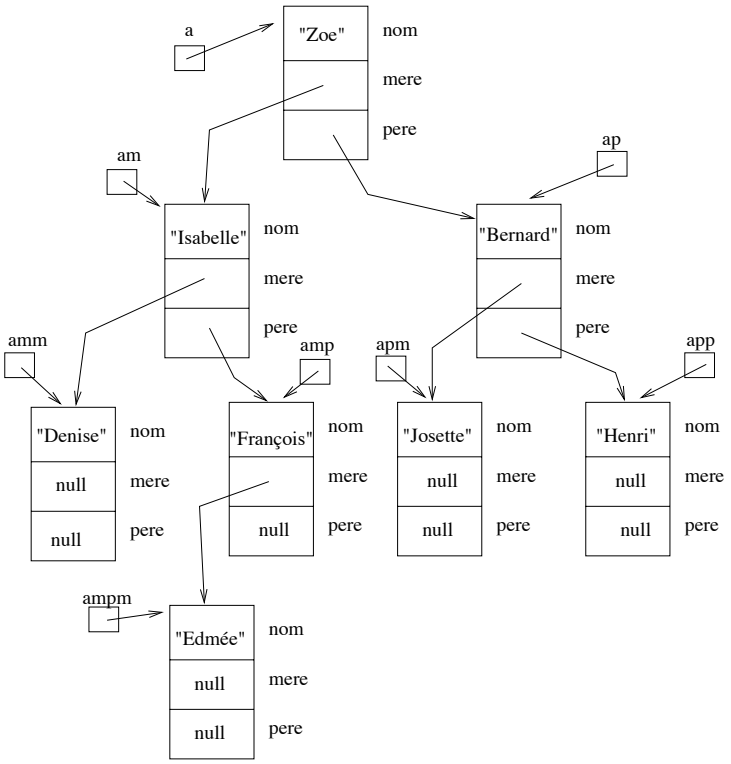

FIGURE 6.2 – Un arbre généalogique.

Conclusion

Ce cours a présenté les éléments essentiels de la programmation par objets en Java. La suite de ce cours au second semestre (module HMIN215, Structures de données) présente des notions plus avancées, liées à la définition et à l'utilisation des structures de données en Java, notamment les classes imbriquées, les exceptions, les assertions, les interfaces, les streams (Java 8) et la généricité (polymorphisme paramétrique).