

!



Langage de programmation orienté objet - Java -

Françoise Dubisy

UE Conception orientée objet

Table des matières

1	Introduction	4
1.1	Historique.....	4
1.2	Caractéristiques de Java	4
1.2.1	Syntaxe proche du langage C	4
1.2.2	Exploitable dans un environnement distribué	4
1.2.3	Architecture neutre	5
1.2.4	Performances.....	5
1.2.5	Réutilisation	6
1.3	Programmation O.O. <> procédurale	7
1.3.1	Programmation procédurale.....	7
1.3.2	Programmation orientée objet (O.O.)	7
2	Objets & classes	10
2.1	Objet	10
2.2	Classe.....	12
2.3	Variable d'instance et méthode	14
2.4	Constructeur.....	19
2.5	Accès aux variables d'instance	24
2.5.1	Accès en lecture.....	24
2.5.2	Accès en écriture	25
2.6	Appel de méthodes sur des objets	26
2.6.1	Méthode qui retourne une valeur.....	26
2.6.2	Méthode qui ne retourne pas de valeur	28
2.7	Difficulté de choisir entre variable d'instance et méthode.....	29
2.8	Portée des variables	35
2.9	Une méthode peut appeler une autre méthode de la même classe	37
2.10	Méthode retournant un objet.....	40
3	Protections et Information Hiding	44
3.1	Information Hiding : protections <i>private</i> et <i>public</i>	44
3.2	Méthodes publiques d'accès aux variables d'instance privées : getters et setters	46
3.3	Intérêt des setters : rôle de filtre	52
3.4	Introduction à la gestion des exceptions	55
3.5	Notion de package (+ import).....	59
3.6	Protection par défaut : protection de type package.....	63
4	Interface utilisateur.....	65
4.1	Types d'interface utilisateur.....	65
4.2	Entrées – sorties à la console.....	65
4.2.1	Entrées	65

4.2.2	Sorties.....	65
4.3	Séparation vue – modèle	66
4.4	La méthode <i>toString</i>	68
5	Surcharge (overloading) des constructeurs et méthodes	72
5.1	Surcharge des constructeurs	72
5.2	Surcharge des méthodes.....	77
6	Liens entre objets	82
6.1	Lien entre deux objets (entre deux classes)	82
6.2	Appel implicite à la méthode <i>toString</i> d'un objet référencé par une variable d'instance	87
6.3	Plus de deux classes reliées.....	89
7	Héritage.....	98
7.1	Sous-classe (déclaration et constructeur)	98
7.2	Héritage des variables d'instance et des méthodes	103
7.3	Accès aux variables d'instance privées héritées.....	105
7.4	Redéfinition (overwriting/overriding) de méthode.....	108
7.5	Hiérarchie d'héritage	111
7.6	Polymorphisme.....	119
7.7	Protection de type <i>protected</i>	126
8	Les mots-clés <i>static</i> et <i>final</i>	132
8.1	Variable de classe	132
8.2	Méthode de classe	137
8.2.1	Variable de classe et le principe de l'Information Hiding	138
8.2.2	Gestion des dates.....	140
8.3	Bloc d'instructions déclaré <i>static</i>	142
8.4	Classes <i>String</i> et classes de type <i>Wrapper</i>	144
8.5	Constante : variable déclarée <i>final</i>	146
8.6	Méthode déclarée <i>final</i>	148
8.7	Classe déclarée <i>final</i>	149
9	Classes abstraites et interfaces	150
9.1	Classe abstraite (<i>abstract</i>)	150
9.2	Interface	158
10	Les tableaux.....	164
10.1	Tableaux d'éléments de type primitif	164
10.2	Tableaux d'objets.....	171

1 Introduction

Java est un langage impératif orienté objet. On notera parfois O.O. pour **O**rienté **O**bjets.

Comme le langage C sur lequel il est basé, le langage Java est "Case Sensitive", ou "sensible à la casse" en français : les majuscules ont de l'importance. Par exemple, *nom* et *Nom* sont interprétés différemment.

1.1 Historique

① Langage machine (première génération) :

Un programme est une suite de 0 et de 1

② Langage assembleur (deuxième génération) :

Apparition d'instructions (exemples : *load*, *store*, *add*...)

③ Langage de troisième génération :

- Fortran (IBM) : 1954-1957
- Cobol (Common Business Oriented Language) : 1959
- Basic (université de Dartmouth) : 1965
- Langage C (par Ritchie aux laboratoires Bell) : 1972
- Pascal (utilisation académique : surtout dans les universités)
- Java : 1995
- ...

1.2 Caractéristiques de Java

1.2.1 Syntaxe proche du langage C

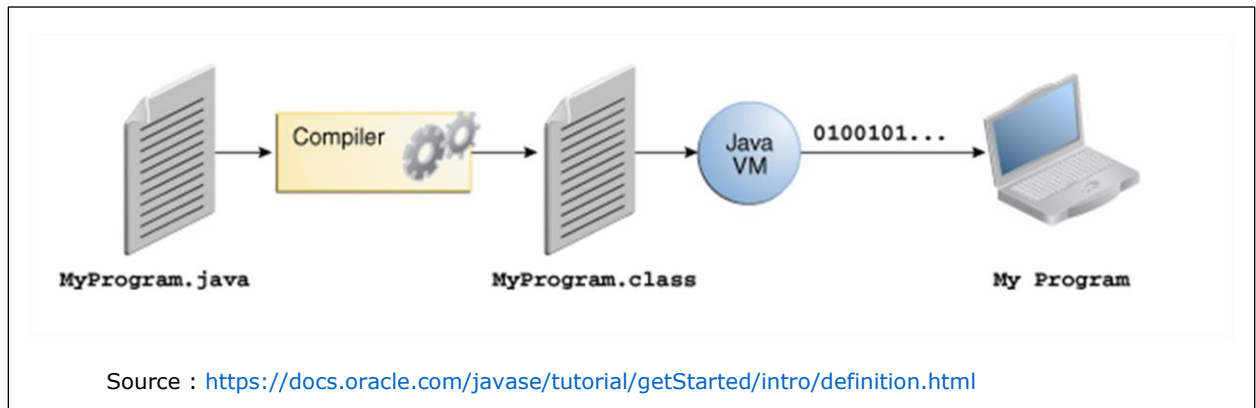
Cf. *if*, *else*, *for*, *while*, *switch*...

1.2.2 Exploitable dans un environnement distribué

Aspect sécurité important !

En vue de sécuriser davantage les programmes, **la notion de pointeur a disparu**. Il est impossible d'accéder à une zone de la mémoire qui n'est pas allouée au programme.

1.2.3 Architecture neutre



1. Le code du programme Java est **édité** dans un fichier **.java**
2. Le fichier **.java** est ensuite **compilé** : le compilateur génère un fichier **.class** contenant des instructions en **bytecode**. Le bytecode ne dépend d'aucune plateforme particulière (windows, linux...).
3. Le fichier **.class** est ensuite **interprété** à l'exécution par une machine dite virtuelle. Ce n'est qu'alors que le bytecode est traduit en code exécutable par la machine sur laquelle on exécute le programme. Le bytecode n'est exécutable que sur une machine virtuelle. Il n'est donc pas directement exécutable.

L'avantage de ce principe (compilation/interprétation) est que le même fichier compilé (.class) peut s'exécuter sur différentes plateformes (linux, windows...), à la seule condition que la plateforme d'accueil dispose d'un interpréteur Java (machine virtuelle).

1.2.4 Performances

En général, les performances sont satisfaisantes, sauf exception (par exemple pour les applications en temps réel). Elles sont cependant moins bonnes que pour des programmes compilés qui sont eux directement exécutables.

1.2.5 Réutilisation

Le but en programmation est de ne pas réinventer la roue, mais de diminuer le temps de programmation en réutilisant le plus possible les composants déjà existants.

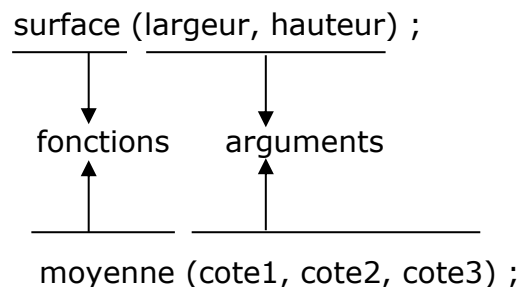
Le langage Java permet de réutiliser des composants existants. Il permet en outre de les réutiliser en les adaptant si nécessaire (cf. l'héritage en Java).

1.3 Programmation O.O. <> procédurale

1.3.1 Programmation procédurale

Un programme classique est un ensemble d'instructions combinant :

- Des affectations (*exemple : hauteur = 3*)
- Des structures de contrôle (*exemples : if-else, boucle while...*)
- Des appels de fonctions ou procédures. Exemple :



Une fonction est donc appelée avec des données que sont les arguments. Les données peuvent être simples ou sous forme de structures.

Un programme procédural peut être vu comme un ensemble d'appels de fonctions (+ structures de contrôle).

1.3.2 Programmation orientée objet (O.O.)

En programmation O.O., on voit le monde en termes d'**objets**. Exemples d'objet : *un étudiant, un ordinateur, un véhicule, la fenêtre d'accueil d'une application, un bouton dans cette fenêtre...*

Un programme O.O. est constitué d'un ensemble d'objets qui interagissent entre eux à l'aide de messages.

De plus, dans un programme, on pourrait avoir plusieurs objets de même type. Exemples : *plusieurs étudiants à encoder, plusieurs boutons dans une fenêtre...*

On prévoit alors des **moules**, des **fabriques** pour créer des objets d'un même type. Exemples : *le moule Etudiant, le moule Vehicule...*

En Java, un moule = une **classe**.

Une classe est donc le moule, le modèle pour créer des objets de même type. Autrement dit, une classe est une sorte d'usine permettant de fabriquer des

objets de même gabarit. Exemple : *la classe Etudiant ne permettra de générer que des objets de type Etudiant.*

Une classe contient aussi **toutes les fonctions** (qu'on appelle **méthodes**) que l'on peut appliquer sur les objets créés avec cette classe. Une classe peut être vue comme une grande boîte à outils rassemblant toutes les fonctions applicables sur des objets de cette classe.

On regroupe donc dans une même classe la définition des caractéristiques ou propriétés des objets ainsi que le code de toutes les fonctions (méthodes) que l'on peut appliquer sur ces objets. On parle alors d'**encapsulation**, car on encapsule, on regroupe dans une même classe à la fois les propriétés des objets (variables) et les actions (fonctions).

Une classe contient deux types d'information :

- Des caractéristiques/propriétés des objets de la classe (variables)
- Des fonctions (méthodes) que l'on pourra appeler sur tout objet de la classe

- *Exemple 1 : manipulation de rectangles*

1° *On a besoin d'un objet rectangle ⇒ Créer la classe Rectangle avec ses propriétés (largeur, hauteur...) & fonctions (surface, périmètre...)*

2° *On crée un exemplaire de rectangle, par exemple rectangleRouge, qui contient ses propres valeurs pour largeur et hauteur.*

3° *On appelle la fonction surface sur l'objet rectangleRouge :*
rectangleRouge.surface()

- *Exemple 2 : gestion des étudiants*

1° *On a besoin d'un objet étudiant ⇒ Créer la classe Etudiant avec ses propriétés (coordonnées, UE du PAE, ects réussis...) & fonctions (reussitePAE...)*

2° *On crée un exemplaire d'étudiant, par exemple etudiantJulien, qui contient ses propres valeurs pour les UE du PAE et les ects réussis*

3° *On appelle la fonction reussitePAE sur l'objet etudiantJulien :*
etudiantJulien.reussitePAE()

Tableau récapitulatif des différences entre C et Java

C	Java
Un programme est un ensemble de fonctions	Un programme est un ensemble d'objets
Syntaxe : nomFonction(arguments)	Syntaxe : objet.nomMethode(...)
Argument = donnée	Fonction // méthode
Fonction puis arguments	Objet puis méthode

En O.O., une fonction / méthode ne peut être appelée que sur un objet (exceptionnellement sur une classe : cf. chapitre 8).

Par conséquent, tout appel de fonction / méthode doit être préfixé par le nom d'un objet .

objet . nomMethode(...)

On dit aussi que les objets reçoivent des messages à traiter.

2 Objets & classes

En programmation O.O., on voit donc un programme comme étant constitué d'un ensemble d'objets qui interagissent.

De plus, comme on peut avoir plusieurs objets de même type, on prévoit des **fabriques** pour créer des objets d'un même type ; une classe est le moule, le modèle pour créer des objets de même type.

2.1 Objet

Dans le monde réel, les objets sont partout.

Tout objet est caractérisé par deux types d'informations : des informations relatives à son état et des informations relatives à ses comportements possibles.

Exemples

Objet	État	Comportement
<i>un chien</i>	<i>nom, couleur, race</i>	<i>aboyer, dormir, manger</i>
<i>une voiture</i>	<i>modèle, marque, plaque</i>	<i>rouler, tourner, stopper</i>

Propriétés

Fonctions (méthodes)

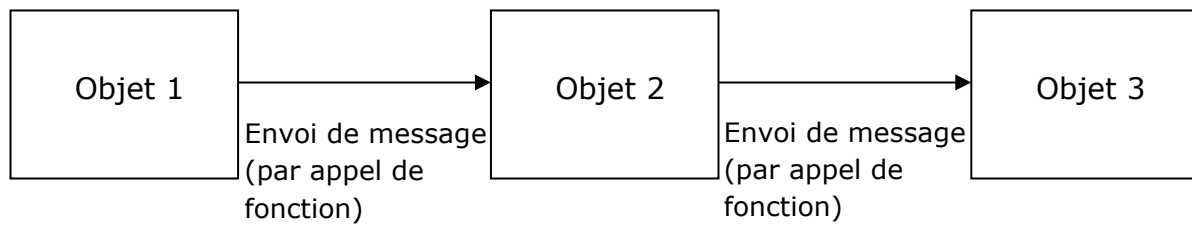
Tout objet est caractérisé par :

- Des propriétés → variables qui peuvent contenir des valeurs (1)
- Des portions de code correspondant à son comportement : on peut appeler ces portions de code sur l'objet lui-même. Il s'agit de fonctions qui peuvent lire ou modifier les variables (1) et/ou exécuter n'importe quelle autre instruction.

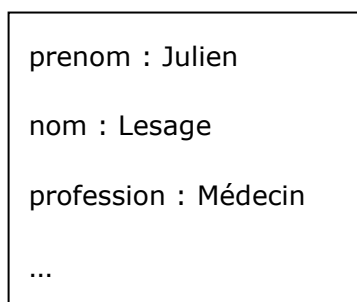
Exemples : afficher des informations dans une fenêtre, enregistrer des données en base de données...

Un programme est vu comme un ensemble d'objets qui interagissent entre eux par envois de messages. Un objet 1 **envoie un message à un autre** objet 2 **en appelant une fonction** sur cet objet 2.

Exemple de dialogue d'objets entre eux (par appels de fonctions)

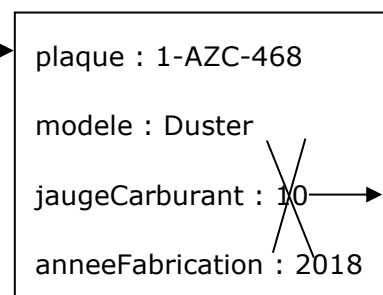


Objet *doc*



Objet *duster*

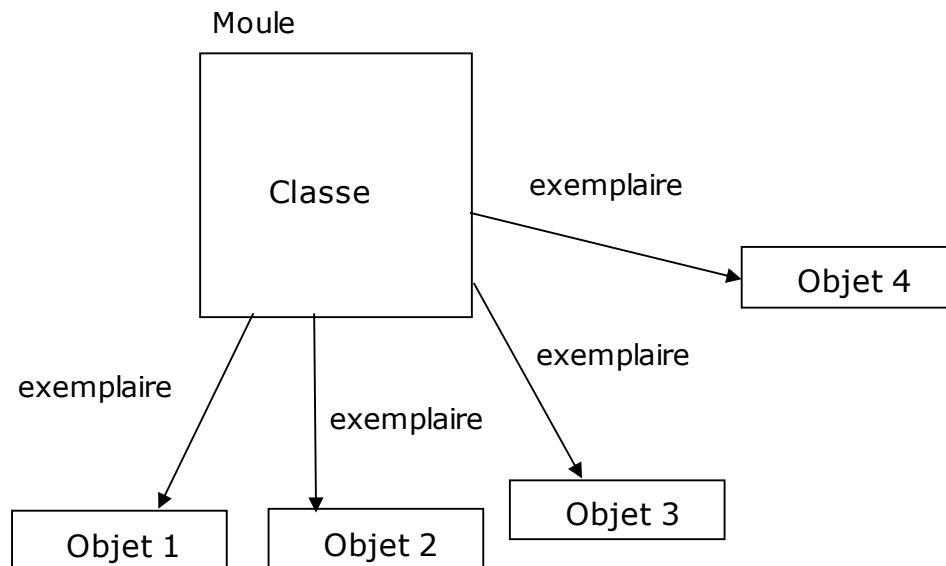
Message :
*Ajouter 30 litres
de carburant*



40

2.2 Classe

Comme expliqué précédemment, pour disposer d'un objet, il faut d'abord disposer de son "moule", de sa "fabrique", c'est-à-dire de sa classe. Ensuite, on créera des exemplaires à partir de cette classe.



Une classe est créée dans un fichier.java (généralement, un fichier par classe).

Un objet est donc un exemplaire (on dira aussi instance) "moulé" à partir d'une classe.

```
⇒ class Vehicule {  
    ...  
}
```

Par convention, **les noms de classe commencent par une majuscule.**

Une classe contient deux types d'informations.

- Les propriétés sont appelées **variables d'instance**
 - À tout objet (tout exemplaire) de la classe correspondront en mémoire **ses variables d'instance** avec leur propre valeur.
 - Exemple : l'objet *duster* (de type *Vehicule*) a en mémoire sa propre variable *plaque* avec la valeur *1-AZC-468*.

- Les fonctions sont appelées **méthodes**
 - Les méthodes déterminent **les comportements possibles des objets de la classe.**
 - Il y a **une seule copie en mémoire du code de chacune des méthodes** de la classe, mais n'importe laquelle de ces méthodes peut être exécutée/appelée sur n'importe quel objet de la classe

Si par exemple la classe *Vehicule* contient 5 méthodes et si 10 objets de type *Vehicule* sont créés (*vehicule1*, *vehicule2*, ... , *vehicule10*) :

- ⇒ Il y a en mémoire 10 variables d'instance *plaque* (une pour chaque objet de type *vehicule*)
- ⇒ Il y a une seule copie en mémoire de chacune des 5 méthodes

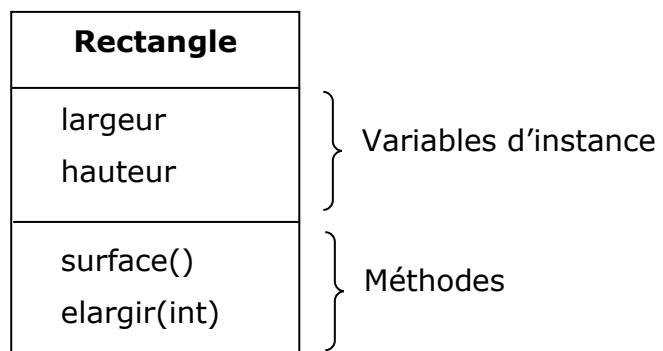
2.3 Variable d'instance et méthode

Par convention, tout nom de variable d'instance et de méthode commence par une **minuscule**.

Exemple 1

Supposons que nous devons manipuler des rectangles. Un rectangle est décrit par sa largeur et sa hauteur. On aimerait également pouvoir calculer la surface d'un rectangle. On voudrait en outre pouvoir élargir un rectangle (c'est-à-dire augmenter sa largeur).

Diagramme de classe UML :



Code de la classe en Java :

```
class Rectangle { // Début de la classe
    // Variables d'instance
    int largeur;
    int hauteur;

    int surface( ) { // Méthode qui retourne une valeur
        return largeur * hauteur;
    }

    void elargir(int delta) { // Méthode qui ne retourne pas de valeur
        largeur += delta;
    }
} // Fin de la classe
```

Pas d'argument car une méthode a directement accès aux variables d'instance de sa classe.

La caractéristique *surface* d'un rectangle est considérée comme une **méthode** plutôt qu'une variable d'instance. En effet, la surface d'un rectangle **peut être calculée** à partir d'autres caractéristiques déclarées sous forme de variables

d'instance (en l'occurrence, *largeur* et *hauteur*). Dès lors, sauf exception, *surface* sera déclarée sous forme de méthode.

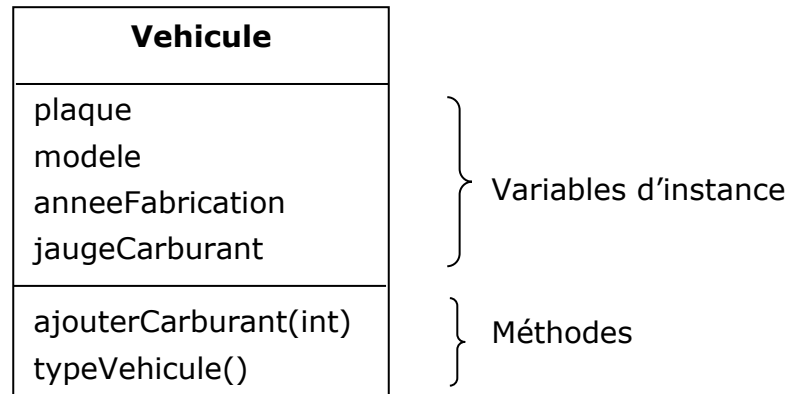
D'autre part, la méthode *surface* () ne demande **aucun argument**, car **toute méthode a directement accès aux variables d'instance de la classe**.

Lorsque la méthode *surface* () sera appelée sur un objet de la classe *Rectangle*, elle lira les valeurs des variables d'instance *hauteur* et *largeur* de l'objet sur lequel elle sera appelée.

Exemple 2

Supposons que nous devons manipuler des véhicules. Un véhicule est caractérisé par une plaque minéralogique, une année de fabrication, un modèle (exemple : Clio) et le niveau de sa jauge à carburant. On doit pouvoir rajouter du carburant dans le véhicule (et le refléter en augmentant le niveau de sa jauge) et retourner une chaîne de caractères décrivant le type de véhicule à partir de son modèle et son année de fabrication.

Diagramme de classe UML :



Code de la classe en Java :

```
class Vehicule {
    // Variables de type référence
    String plaque;
    String modele;

    // Variables de type primitif
    int anneeFabrication;
    int jaugeCarburant;

    void ajouterCarburant(int nbLitres) {    // Méthode qui ne retourne rien
        jaugeCarburant += nbLitres;
    }

    String typeVehicule() {    // Méthode qui retourne un objet de la classe String
        return modele (+) " " + anneeFabrication;
    }
}
```

An arrow points from the **(+)** symbol in the `return` statement to a box labeled "Opérateur de concaténation".

Notons que le principe du "**Camel Case**" est privilégié en Java : si un nom est composé de plusieurs mots, ils sont concaténés (pas de tirets), entièrement écrits en minuscules, avec une majuscule au début de chaque mot qui compose le nom. Plus précisément,

- Nom de classe
 - La première lettre est en majuscule
 - Exemples : **EntretienVehicule**, **EtudiantInformatiqueGestion**
- Nom de variable ou méthode
 - La première lettre est en minuscule
 - Exemples :
 - Variables d'instance : **jaugeCarburant**, **anneeFabrication**
 - Méthodes : **ajouterCarburant**, **typeVehicule**

Les variables peuvent être de type primitif ou de type référence.

Types primitifs (commencent par une minuscule)				
Entier	byte (1 byte)	short (2 bytes)	int (4 bytes)	long (8 bytes)
Réel	float (4 bytes)		double (8 bytes)	
Caractère	char (2 bytes)			
Booléen	boolean (seules valeurs possibles : true ou false)			

Exemples

```
int entier = 123;
double reelDouble = 12.3;
float reelFloat = 12.3f; // N.B. il faut ajouter 'f' à la fin de la valeur d'un float
boolean booleen = false;
char caractere = 's';
```

Les types références sont des **classes**. Ils commencent donc par une majuscule. Exemple : **String**. Cette classe permet de gérer facilement les chaînes de caractères. Il n'y a donc plus besoin de jouer avec des tableaux de caractères comme en langage C (ni de gérer un caractère de fin de tableau comme en C). Une variable de type chaîne de caractères sera désormais **un objet de la classe String**.

Attention, une variable de type *String* contient en fait une référence vers un objet ailleurs en mémoire. Il s'agit d'un pointeur caché.

Exemple

```
|| String nom = "Marie";
```

Représentation en mémoire



Bon nombre de méthodes intéressantes ont été prévues dans la classe *String* par les concepteurs de Java qui rendent ainsi la manipulation des chaînes de caractères plus aisée. La classe *String* peut être vue comme une boîte à outils de manipulation de chaînes de caractères.

Citons à titre d'exemples :

- Retourner la longueur d'une chaîne de caractères (*length*) ;
- Extraire un caractère (*charAt*) ;
- Extraire une sous-chaîne de caractères (*substring*) ;
- Comparer deux chaînes de caractères (*compareTo*) ;
- Vérifier le contenu d'une chaîne de caractères (*equals*) ;
- Transformer une chaîne de caractères en majuscules ou minuscules (*toUpperCase* ou *toLowerCase*).

L'opérateur + appliqué à deux variables de type numérique exécute l'addition.

Exemples :

```
|| int a = 4;  
|| int b = 2;  
|| int c = a + b;
```

⇒ La variable c contiendra 6

L'opérateur + appliqué à deux objets de type String est l'**opérateur de concaténation** (cf. méthode *typeVehicule*). Appliqué à deux objets de type chaînes de caractères, l'opérateur + construit une nouvelle chaîne de caractères en concaténant les deux chaînes de caractères correspondant aux objets donnés.

L'opérateur + peut être appliqué sur des **variables** ou directement sur des **valeurs**.

Exemples

```
String message1 = "Salut, ";  
String message2 = "ça va ?";  
String messageFinal = message1 + message2;
```

⇒ La variable *messageFinal* contiendra "Salut, ça va ?"

```
"Bonjour, " + "bienvenue !"
```

⇒ Construit la chaîne de caractères : "Bonjour, bienvenue !"

```
String personne = "Jules Dupond";  
String messageFinal = "Monsieur " + personne;
```

⇒ *messageFinal* contiendra "Monsieur Jules Dupond"

La méthode *typeVehicule* construit et retourne une chaîne de caractères en concaténant le modèle et l'année de fabrication du véhicule.

Exemple : *Clio 2018*

2.4 Constructeur

Comment créer des exemplaires de ces classes et les utiliser ?

À l'exécution, la machine virtuelle Java recherche et exécute le contenu d'une méthode dont la déclaration est réservée, à savoir la méthode **main** (cf. fonction *main* du langage C).

La déclaration de la méthode main est :

public static void main (String [] args)

Les mots réservés *public* et *static* seront étudiés ultérieurement dans le cours.

Soit une nouvelle classe qui contient la méthode **main**. Appelons cette nouvelle classe : *Principal*.

Un objet (appelé *premierRectangle*) de type *Rectangle* est déclaré dans la méthode *main* de cette classe.

```
class Principal {  
    public static void main(String [] args) {           // Méthode main  
        Rectangle premierRectangle;    // Déclaration d'un objet de type Rectangle  
    }  
}
```

Rappel : Par convention, tous les noms de variable commencent par une minuscule. Par conséquent, la variable de type *rectangle* est appelée ***premierRectangle*** et non ***PremierRectangle***.

Il s'agit d'une simple déclaration de variable. Cette ligne de code (c'est-à-dire *Rectangle premierRectangle*) ne fait que prévenir Java qu'on va utiliser une variable de type *Rectangle*.

Mais **aucune place mémoire n'est réservée** via cette instruction. Pour ce faire, il faut appeler un **constructeur** qui va réserver la place mémoire pour l'objet *premierRectangle* (ainsi que pour ses variables d'instance) et qui va éventuellement initialiser ses variables d'instance.

Cet appel se fait via le mot réservé **new** suivi du nom du constructeur.

Un constructeur est une méthode qui a une signature particulière. Tout constructeur porte le **même nom que la classe** et ne retourne aucun résultat.

Le rôle du constructeur, quand il est invoqué, est donc de réserver de la place mémoire pour tout nouvel objet que l'on crée et d'éventuellement initialiser ses variables d'instance.

Suggestion (**mais ce n'est pas une obligation**) : prévoir un constructeur avec autant d'arguments qu'il y a de variables d'instance dans la classe. En effet, disposer d'un constructeur qui peut recevoir en arguments les valeurs pour initialiser les variables d'instance est un gain d'écriture de lignes de code pour le programmeur.

Exemple 1

Rectangle
largeur hauteur
surface() elargir(int)

Par définition, un constructeur n'a jamais de type de retour
⇒ On ne note même pas void

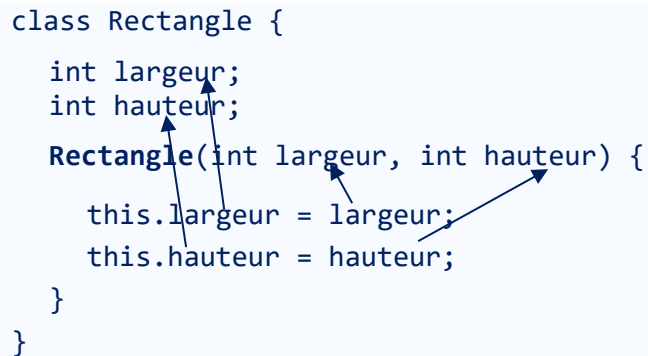
```
class Rectangle {  
    // Variables d'instance  
    int largeur;  
    int hauteur;  
  
    Rectangle(int initLargeur, int initHauteur) {           // Constructeur  
        largeur = initLargeur;                             // Ou this.largeur = initLargeur ;  
        hauteur = initHauteur;                             // Ou this.hauteur = initHauteur;  
    }  
    int surface() {                                         // Méthodes  
        return largeur * hauteur;  
    }  
    void elargir(int delta) {  
        largeur += delta;  
    }  
}
```

Notons que **this** fait référence à l'occurrence courante de la classe. Si on utilise *this* dans le constructeur, on fait alors référence à l'objet que l'on est en train de construire : *this.largeur* fait donc référence à la largeur de l'objet que l'on est en train de créer. Par contre, si on ne préfixe pas un nom de variable par *this*, le compilateur recherchera la déclaration de variable la plus proche en termes de portée.

N.B. S'il y a ambiguïté dans les noms de variable (par exemple, même nom d'argument qu'une variable d'instance), on peut lever l'ambiguïté en utilisant le mot réservé *this* ; l'instruction *this.largeur = largeur* a pour effet de prendre le contenu de l'argument *largeur* et de le placer dans la variable d'instance *largeur*.

Autre écriture possible du constructeur :

```
class Rectangle {
    int largeur;
    int hauteur;
    Rectangle(int largeur, int hauteur) {
        this.largeur = largeur;
        this.hauteur = hauteur;
    }
}
```



Pour réserver de la place mémoire pour un objet de type rectangle, on fait appel au constructeur précédé du mot réservé **new**.

```
class Principal {
    public static void main(String [] args) { // Signature de la méthode main
        Rectangle premierRectangle; /* Déclaration d'une variable de type Rectangle
                                     MAIS PAS d'allocation mémoire */
        premierRectangle = new Rectangle(3, 5);
                                /* Appel au constructeur de la classe Rectangle
                                ↪ Allocation mémoire + initialisation des variables d'instance */
    }
}
```

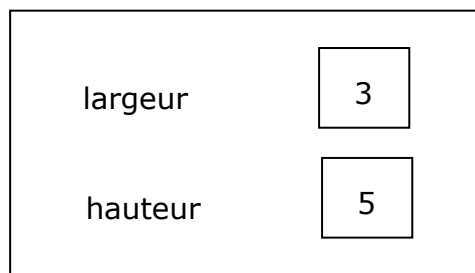
Ces deux instructions ont pour effet de créer en mémoire un objet référencé par la variable *premierRectangle*. La place mémoire nécessaire pour chacune des variables d'instance est réservée.

+ La variable d'instance *largeur* de l'objet *premierRectangle* est initialisée à 3.

+ La variable d'instance *hauteur* de l'objet *premierRectangle* est initialisée à 5.

En mémoire :

Objet ***premierRectangle***



Exemple 2

Vehicule
plaque modele anneeFabrication jaugeCarburant
ajouterCarburant(int) typeVehicule()

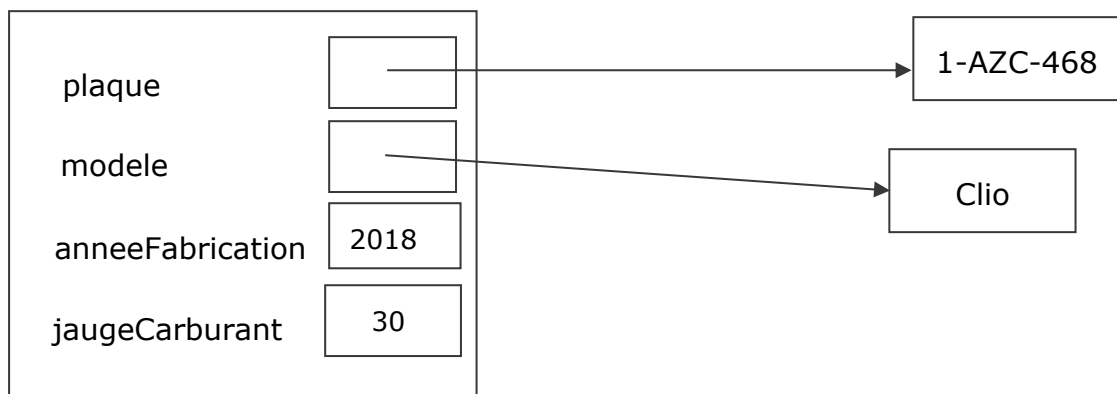
```
class Vehicule {  
    // Variables d'instance  
    String plaque;  
    String modele;  
    int anneeFabrication;  
    int jaugeCarburant;  
  
    // Constructeur  
    Vehicule(String plaque, String modele, int anneeFabrication,  
              int jaugeCarburant) {  
        this.plaque = plaque;  
        this.modele = modele;  
        this.anneeFabrication = anneeFabrication;  
        this.jaugeCarburant = jaugeCarburant;  
    }  
  
    // Méthodes  
    void ajouterCarburant(int nbLitresAjoutes) {  
        jaugeCarburant += nbLitresAjoutes;  
    }  
    String typeVehicule() {  
        return modele + " " + anneeFabrication;  
    }  
}
```

```

class Principal {
    public static void main(String [] args) { // Signature de la méthode main
        Vehicule clio = new Vehicule("1-AZC-468","Clio",2018,30);
        /* Déclaration + allocation de mémoire
           + Initialisation des variables d'instance en une seule instruction */
    }
}

```

En mémoire : Objet ***clio***



Les variables d'instance *plaque* et *modele* sont de type (référence) ***String***. La valeur de chacune de ces variables représente un objet. Ces variables d'instance sont donc des références vers des objets qui se trouvent ailleurs en mémoire.

N.B. Le constructeur n'est pas obligatoire. En effet, si aucun constructeur n'est explicitement prévu dans la classe, Java en utilise un **par défaut** ; celui-ci ne reçoit aucun argument et initialise les variables d'instance à des valeurs par défaut :

- Nombre : 0
- Booléen : false
- Objet : référence null

Attention cependant que dès qu'un constructeur avec argument est prévu par le programmeur dans une classe, le constructeur sans argument n'existe plus.

2.5 Accès aux variables d'instance

2.5.1 Accès en lecture

Comment accéder en lecture aux valeurs des variables d'instance des objets créés, par exemple pour afficher à l'écran ces valeurs ?

L'instruction pour afficher une chaîne de caractères à l'écran est :


```
System.out.println("blabla") ;
```

Exemple 1

Rectangle
largeur hauteur
surface() elargir(int)

```
class Principal {  
    public static void main(String [] args) {  
        Rectangle premierRectangle = new Rectangle(3,5);  
        System.out.println("Hauteur : " + premierRectangle.hauteur + " cm");  
    }  
}
```

Opérateur de concaténation



⇒ Affiche à l'écran : Hauteur : 5 cm

5 est la valeur de la variable *hauteur* de l'objet *premierRectangle*

Exemple 2

Vehicule
plaque modele anneeFabrication jaugeCarburant
ajouterCarburant(int) typeVehicule()


```

class Principal {
    public static void main(String [] args) {
        Vehicule clio = new Vehicule("1-AZC-468","Clio",2018,30);
        System.out.println("le véhicule immatriculé " + clio.plaque +
            "\ndispose d'une réserve de " + clio.jaugeCarburant + " litres");
    }
}

```

⇒ Affiche à l'écran : *le véhicule immatriculé 1-AZC-468*
dispose d'une réserve de 30 litres

2.5.2 Accès en écriture

Comment modifier les valeurs des variables d'instance des objets créés ?

Exemple

```

class Principal {
    public static void main(String [] args) {
        Rectangle premierRectangle = new Rectangle(3,5);
        premierRectangle.hauteur = 2 ;    // Affecte la valeur 2 à la variable hauteur
        System.out.println("Hauteur : " + premierRectangle.hauteur + " cm");
    }
}

```

⇒ Affiche à l'écran : *Hauteur : **2** cm*

```

Vehicule clio = new Vehicule("1-AZC-468","Clio",2018,30);
clio.jaugeCarburant = 80 ;    // Affecte 80 à la variable jaugeCarburant
System.out.println("le véhicule immatriculé " + clio.plaque +
    "\ndispose d'une réserve de " + clio.jaugeCarburant + " litres");

```

⇒ Affiche à l'écran : *le véhicule immatriculé 1-AZC-468*
*dispose d'une réserve de **80** litres*

2.6 Appel de méthodes sur des objets

2.6.1 Méthode qui retourne une valeur

Comment appeler sur un objet une méthode qui retourne une valeur ?

Syntaxe : **nomVariable = nomObjet.nomMethode(arg₁, arg₂, ..., arg_n)**

⇒ Affecte à la variable **nomVariable** la valeur retournée par la méthode **nomMethode (arg₁, arg₂, ..., arg_n)** exécutée sur l'objet **nomObjet** .

Exemple 1

Rectangle
largeur hauteur
surface() elargir(int)

```
class Principal {  
    public static void main(String [] args) {  
        Rectangle premierRectangle = new Rectangle(3,5);  
        int superficie ;  
        superficie = premierRectangle.surface();  
        /* La variable superficie reçoit le résultat de l'exécution de la  
           méthode surface() sur l'objet premierRectangle */  
        System.out.println("Superficie : " + superficie);  
    }  
}
```

⇒ Affiche à l'écran : *Superficie : 15*

Exemple 2

Pour afficher le type d'un véhicule (cf. méthode *typeVehicule*), deux solutions possibles sont proposées ci-dessous.

Vehicule
plaque modele anneeFabrication jaugeCarburant
ajouterCarburant(int) typeVehicule()

Solution 1 (3 instructions)

```
class Principal {  
    public static void main(String [] args) {  
        Vehicule clio = new Vehicule("1-AZC-468","Clio",2018,30);  
        ① String type;  
        ② type = clio.typeVehicule();  
        ③ System.out.println("Le véhicule est du type : " + type);  
    }  
}
```

⇒ Affiche à l'écran : *Le véhicule est du type : Clio 2018*

Solution 2 (1 seule instruction)

```
class Principal {  
    public static void main(String [] args) {  
        Vehicule clio = new Vehicule("1-AZC-468","Clio",2018,30);  
        ① System.out.println("Type du véhicule : " + clio.typeVehicule());  
    }  
}
```

⇒ Affiche à l'écran : *Type du véhicule : Clio 2018*

2.6.2 Méthode qui ne retourne pas de valeur

Comment appeler sur un objet une méthode qui ne retourne pas de valeur ?

Une méthode qui ne retourne pas de valeur débute sa signature par le mot réservé **void**. Une telle méthode ne contient donc pas d'instruction **return**.

Syntaxe : **nomObjet.nomMethode (arg₁, arg₂, ..., arg_n)**

⇒ Exécute la méthode **nomMethode (arg₁, arg₂, ..., arg_n)** sur l'objet **nomObjet**.

Exemples

Rectangle
largeur hauteur
surface() elargir(int)

Vehicule
plaque modele anneeFabrication jaugeCarburant
ajouterCarburant(int) typeVehicule()

```
class Principal {  
    public static void main(String [] args) {  
        Rectangle premierRectangle = new Rectangle(3,5);  
        premierRectangle.elargir(1);           // Augmente de 1 la largeur de l'objet  
        System.out.println("Largeur : " + premierRectangle.largeur + " cm");  
    }  
}
```

⇒ Affiche à l'écran : *Largeur : 4 cm*

```
Vehicule clio = new Vehicule("1-AZC-468","Clio",2018,30);  
clio.ajouterCarburant(10);           // Ajoute 10 à la variable jaugeCarburant  
System.out.println("Réserve de " + clio.jaugeCarburant + " litres");
```

⇒ Affiche à l'écran : *Réserve de 40 litres*

2.7 Difficulté de choisir entre variable d'instance et méthode

Ne sont déclarées variables d'instance que les caractéristiques qui ne sont pas dérivables, pas calculables à partir d'autres.

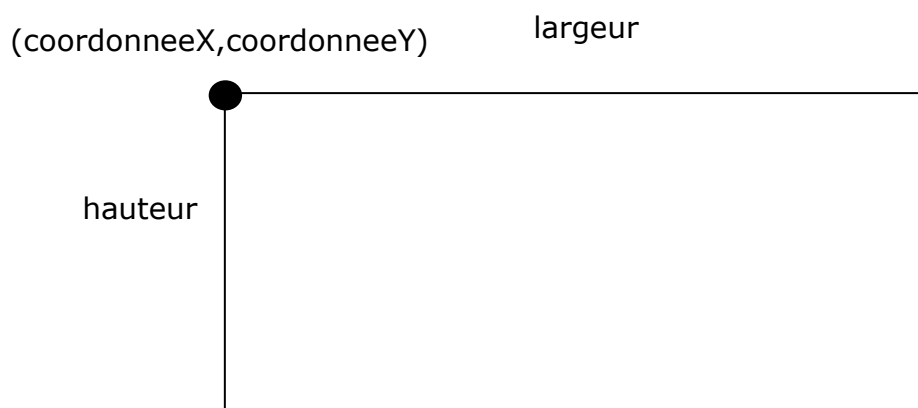
Toute caractéristique dérivable ou calculable à partir d'autres doit, sauf exception, être déclarée sous forme de méthode.

Exemple 1

Objectif : manipuler des rectangles

- Calculer le périmètre et la surface de rectangles
- Positionner et déplacer des rectangles
- Modifier la taille de rectangles

Pour positionner un rectangle dans un espace, il faut disposer des coordonnées x et y du coin supérieur gauche du rectangle.



Rectangle			
	coordonneeX (1)	<div> Calculables à partir des variables d'instances ⇒ méthodes </div>	
	coordonneeY (2)		
	largeur (3)		
	hauteur (4)		
Type de retour		Variables d'instance utilisées	Arguments supplémentaires
nombre	perimetre	accès en lecture à (3) et (4)	aucun
nombre	surface	accès en lecture à (3) et (4)	aucun
aucun	elargir	accès en lecture + écriture à (3)	valeur à ajouter à la largeur
aucun	agrandir	accès en lecture + écriture à (4)	valeur à ajouter à la hauteur
aucun	deplacerEn	accès en écriture à (1) et (2)	nouvelles coordonnées x et y

La largeur et la hauteur d'un rectangle sont des **propriétés** du rectangle qui ne sont **pas calculables ou dérivables** à partir d'autres. La largeur et la hauteur sont donc déclarées comme **variables d'instance**. Par contre, la surface et le périmètre d'un rectangle sont **calculables** à partir de sa largeur et de sa hauteur. La surface et le périmètre sont déclarés sous forme de **méthodes**.

Les méthodes surface et périmètre ne demandent aucun argument. En effet, **toute méthode a accès aux variables d'instance de la classe**. Dans le code de la méthode *surface*, on peut donc accéder à la valeur des variables d'instance *hauteur* et *largeur* du rectangle dont on doit calculer la surface.

Le code complet de la classe *Rectangle* est donc

```
class Rectangle {
    int coordonneeX;
    int coordonneeY;
    int largeur;
    int hauteur;

    Rectangle(int coordonneeX, int coordonneeY, int largeur, int hauteur) {
        this.coordonneeX = coordonneeX;
        this.coordonneeY = coordonneeY;
        this.largeur = largeur;
        this.hauteur = hauteur;
    }

    int perimetre() {
        return 2* (largeur + hauteur);
    }
}
```

```

int surface() {
    return largeur * hauteur;
}
void elargir(int augmentation) {
    largeur += augmentation;
}
void agrandir(int augmentation) {
    hauteur += augmentation;
}
void deplacerEn(int nouveauX, int nouveauY) {
    coordonneeX = nouveauX;
    coordonneeY = nouveauY;
}
}

```

Exemple 2

Objectif : gérer des étudiants inscrits en informatique

- Décrire tout étudiant (c'est-à-dire renseigner ses coordonnées : prénom, nom, genre, section, bloc, nom d'utilisateur (login name)...)
- Calculer la réussite totale ou partielle du PAE, le taux de réussite dans les UE de cours généraux et de spécialité...

EtudiantInformatique				
	prenom	(1)		
	nom	(2)		
	genre	(3)		
	section	(4)		
	bloc	(5)		
	totalEctsLangue	(6)		
	nbEctsReussisLangue	(7)		
	totalEctsMath	(8)		
	nbEctsReussisMath	(9)		
	totalEctsProgra	(10)		
	nbEctsReussisProgra	(11)		
ch.carac.	nomUtilisateur		accès en lecture à (1), (2), (4), (5)	aucun
nombre	nbTotalEctsDansPAE	(a)	accès en lecture à (6), (8), (10)	aucun
nombre	nbTotalEctsReussis	(b)	accès en lecture à (7), (9), (11)	aucun
booléen	reussiteTotalePAE	(c)	aucune : appel des méthodes (a) et (b)	aucun
booléen	reussitePartiellePAE		accès lecture à (5) + appel (b) et (c)	aucun
nombre	tauxReussiteCoursGeneraux		accès en lecture à (6), (7), (8) et (9)	aucun
nombre	tauxReussiteCoursSpecialite		accès en lecture à (10) et (11)	aucun

Calculables à partir des variables d'instance
⇒ **méthodes**

Var. d'instance utilisées

Arguments supplém.

Les caractéristiques prénom, nom, genre, section, bloc (numéro du bloc de 1 à 5), ainsi que les nombres d'Ects totaux dans le PAE de l'étudiant en langues, math et programmation et le nombres d'Ects correspondants réussis par l'étudiant ne sont pas calculables à partir d'autres. Elles sont donc déclarées sous forme de variables d'instance.

Notons que si nous devons traiter toutes les UE d'un PAE, il serait préférable d'utiliser un tableau. Les tableaux seront abordés au chapitre 10. Pour les besoins de l'exemple, nous garderons des variables d'instance simples.

Les caractéristiques ci-dessous peuvent être calculées à partir des variables d'instance ou d'autres méthodes :

- nomUtilisateur (nom d'utilisateur pour l'accès au réseau informatique, s'il est constitué à partir de certaines caractéristiques de l'étudiant comme les nom, prénom, section et numéro de bloc, par exemple, DupondPierreIn3)
- Nombre total d'ECTS dans le PAE
- Nombre total d'ECTS du PAE réussis
- Réussite totale ou non
- Réussite partielle (par exemple, une réussite à 45 crédits si l'étudiant est en bloc 1)
- Taux de réussite dans les cours généraux
- Taux de réussite dans les cours de spécialité

Ces caractéristiques sont donc déclarées sous forme de méthodes.

Le code de la classe *EtudiantInformatique* est donc :

```
class EtudiantInformatique {
    String prenom;
    String nom;
    char genre;
    String section;
    int bloc;
    int totalEctsLangue;
    int nbEctsReussisLangue;
    int totalEctsMath;
    int nbEctsReussisMath;
    int totalEctsProgra;
    int nbEctsReussisProgra;

    EtudiantInformatique(String prenom, String nom, char genre,
                          String section, int bloc,
                          int totalEctsLangue, int nbEctsReussisLangue,
                          int totalEctsMath, int nbEctsReussisMath,
                          int totalEctsProgra, int nbEctsReussisProgra) {
```



```

        this.prenom = prenom;
        this.nom = nom;
        this.genre = genre;
        this.section = section;
        this.bloc = bloc;
        this.totalEctsLangue = totalEctsLangue;
        this.nbEctsReussisLangue = nbEctsReussisLangue;
        this.totalEctsMath = totalEctsMath;
        this.nbEctsReussisMath = nbEctsReussisMath;
        this.totalEctsProgra = totalEctsProgra;
        this.nbEctsReussisProgra = nbEctsReussisProgra;
    }
    String nomUtilisateur() {
        return nom + prenom + section.substring(0,2) + bloc;
        // section.substring(0,2) retourne les 2 premières lettres de la section
    }
    int nbTotalEctsDansPAE() {
        return totalEctsLangue + totalEctsMath + totalEctsProgra;
    }
    int nbTotalEctsReussis() {
        return nbEctsReussisLangue + nbEctsReussisMath + nbEctsReussisProgra;
    }
    boolean reussiteTotalePAE() {
        ...
    }
    boolean reussitePartiellePAE() {
        ...
    }
    double tauxReussiteCoursGeneraux() {
        return ((nbEctsReussisLangue + nbEctsReussisMath) /
            (double) (totalEctsLangue + totalEctsMath)) * 100;
    }
    double tauxReussiteCoursSpecialite() {
        return (nbEctsReussisProgra / (double)totalEctsProgra) * 100 ;
    }
    ...
}

```

Dans la méthode *tauxReussiteCoursGeneraux*, il faut, comme en langage C, "caster" en double un des deux termes de la division pour obtenir un réel.

En conclusion

- ① Si une caractéristique est **calculable ou dérivable** à partir d'autres caractéristiques (variables d'instance ou méthodes), alors cette caractéristique est déclarée sous forme de **méthode** et non sous forme de variable d'instance. Des exceptions existent toutefois, par exemple pour des raisons de performance.
- ② Les **variables d'instance** d'une classe ne sont **jamais** passées comme **arguments** des méthodes de la classe, ni renvoyées par une méthode : en effet, toute variable d'instance est directement accessible par les méthodes.

2.8 Portée des variables

On peut identifier trois catégories de variables dans une classe

1) Les variables d'instance

Les variables d'instance déclarées dans la classe sont accessibles de partout dans cette classe (par toutes les méthodes et les constructeurs).

2) Les arguments

Les variables passées en arguments dans une méthode (ou dans un constructeur) sont des variables locales à cette méthode ou au constructeur. Aucune autre méthode ou constructeur ne peut y avoir accès.

Pour rappel, il ne faut **pas passer en arguments les variables d'instance** ; celles-ci sont accessibles par toutes les méthodes/constructeurs au sein de la classe.

3) Les variables locales aux méthodes

Les variables déclarées à l'intérieur d'une méthode sont locales à cette méthode. On ne peut donc pas appeler depuis une méthode ou un constructeur une variable définie à l'intérieur d'une autre méthode ou d'un autre constructeur.



Exemple

Entrainement
nomCoureur longueurParcours tempsParcours
vitesseMoyenne() ecartParRapportAuMeilleur(int)

```
class Entrainement {  
    String nomCoureur;  
    int longueurParcours;  
    int tempsParcours;  
  
    Entrainement(String nom, int longueur, int temps) {  
        nomCoureur = nom;  
        longueurParcours = longueur;  
        tempsParcours = temps;  
    }  
}
```

Variables d'instance

Arguments du constructeur

```
double vitesseMoyenne() {  
    double moyenne;  Variable locale  
    moyenne = longueurParcours / (double) tempsParcours;  
    return moyenne;  
}  
//Comparaison par rapport au temps mis par le meilleur  
int ecartParRapportAuMeilleur(int tempsMeilleur) {  
    return tempsParcours - tempsMeilleur;  Argument de la méthode  
}  
}
```

La variable *moyenne* déclarée dans la méthode *vitesseMoyenne* reste locale à cette méthode. Aucune autre méthode (ni constructeur) ne peut y avoir accès !

Les arguments *nom*, *longueur* et *temps* sont locaux au constructeur. Aucune autre méthode (ni constructeur) ne peut y avoir accès !

De même, l'argument *tempsMeilleur* est local à la méthode *comparerAuMeilleur*. Aucune autre méthode (ni constructeur) ne peut y avoir accès !

En conclusion

- ① Une variable d'instance est accessible par n'importe quelle méthode/constructeur de la classe. Les variables d'instance ne doivent donc en aucun cas être passées en arguments des méthodes et constructeurs.
- ② Une variable déclarée en argument reste locale à la méthode/constructeur. Elle ne peut en aucun cas être appelée directement /utilisée à l'extérieur de la méthode/constructeur.
- ③ Une variable déclarée dans une méthode/constructeur reste locale à cette méthode/constructeur.

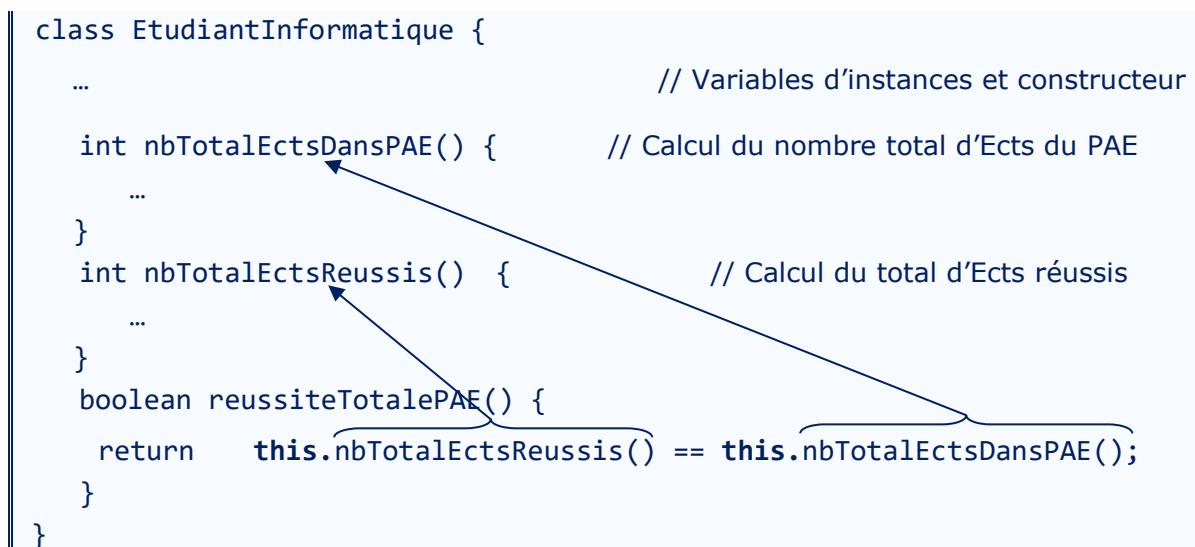
2.9 Une méthode peut appeler une autre méthode de la même classe

Rappel : En O.O., tout appel de méthode est préfixé par un nom d'objet :

nomObjet.nomMethode(arg₁, arg₂, ..., arg_n)

Reprenons la méthode *reussiteTotalePAE* de la classe *EtudiantInformatique*. On peut déterminer si un étudiant a réussi totalement son PAE à partir du nombre total d'Ects dans son PAE, calculé par la méthode *nbTotalEctsDansPAE*, et du nombre total d'Ects qu'il a réussis, calculé par la méthode *nbTotalEctsReussis*.

```
class EtudiantInformatique {  
    ... // Variables d'instances et constructeur  
  
    int nbTotalEctsDansPAE() { // Calcul du nombre total d'Ects du PAE  
        ...  
    }  
  
    int nbTotalEctsReussis() { // Calcul du total d'Ects réussis  
        ...  
    }  
  
    boolean reussiteTotalePAE() {  
        return this.nbTotalEctsReussis() == this.nbTotalEctsDansPAE();  
    }  
}
```



Il est possible de demander à Java d'exécuter une méthode sur l'objet courant. La syntaxe est : **this.nomMethode(...)**. Pour rappel, le mot réservé **this** fait référence à l'objet courant.

Dans ce cas-ci, l'objet courant est celui sur lequel on appellera la méthode *reussiteTotalePAE*. Autrement dit, pour calculer la réussite totale du PAE d'un objet étudiant, on appellera les méthodes *nbTotalEctsDansPAE* et *nbTotalEctsReussis* sur ce même objet étudiant et on comparera leurs résultats.

Soit la classe *Principal* ci-dessous qui crée et manipule un objet de type *EtudiantInformatique*.

```
class Principal {
    public static void main(String [] args) {
        EtudiantInformatique maxime = new EtudiantInformatique(...);
        if ((maxime.reussiteTotalePAE()) {
            ...
        }
    }
}
```

À l'exécution, équivaut à :

```
|| if ((maxime.nbTotalEctsReussis() == maxime.nbTotalEctsDansPAE()))
```

N.B.1 : Il existe encore une écriture plus courte : **this.nomMethode(...)** peut s'écrire directement **nomMethode(...)**.

Ainsi la méthode

```
|| boolean reussiteTotalePAE() {
||     return this.nbTotalEctsReussis() == this.nbTotalEctsDansPAE();
|| }
||
```

peut s'écrire comme suit :

```
|| boolean reussiteTotalePAE() {
||     return nbTotalEctsReussis() == nbTotalEctsDansPAE();
|| }
||
```

this est considéré par défaut

Autres exemples d'appel de méthode de la même classe au sein d'une méthode :

```
|| boolean reussitePartiellePAE() {
||     return !this.reussiteTotalePAE() && bloc == 1
||         && this.nbTotalEctsReussis() >= 45;
|| }
||
```

N.B.2: L'ordre des méthodes dans la classe n'a pas d'importance. Une méthode peut appeler une autre méthode de la classe même si celle-ci est déclarée plus loin dans la classe. Cela ne dispense pas de prévoir un ordre logique des méthodes, surtout si celles-ci sont nombreuses.

Exemple :

```
boolean reussiteTotalePAE() { ... }  
int nbTotalEctsReussis() { ... }  
int nbTotalEctsDansPAE() { ... }
```

Même si les méthodes sont déclarées dans cet ordre, la méthode *reussiteTotalePAE* peut appeler les méthodes *nbTotalEctsReussis* et *nbTotalEctsDansPAE* déclarées après.

En conclusion

- ① Une méthode a accès directement aux variables d'instance de sa classe.
- ② Une méthode peut appeler n'importe quelle autre méthode de sa classe.
- ③ L'ordre des méthodes au sein de la classe n'a pas d'importance.

2.10 Méthode retournant un objet

Une méthode peut retourner une variable de type primitif (int, float, boolean...) ou une référence vers un objet. Ce principe a déjà été appliqué : bon nombre de méthodes déjà rencontrées avaient un type de retour *String*. Ce qui signifie que ces méthodes retournent une référence vers un objet de la classe *String*.

Ce principe peut être généralisé à n'importe quelle autre classe créée par un programmeur ou à n'importe quelle autre classe préexistante.

Une méthode peut donc avoir la signature suivante :

NomClasse nomMethode(arg₁, arg₂, ... , arg_n)

NomClasse étant le nom d'une classe créée par le programmeur.

Exemple

La classe *Point* permet de créer des objets représentant des points. Un point est défini par ses coordonnées x et y.

Prévoyons une classe *FabriqueRectangle* qui contient la méthode *petitRectangleAvec2Points* qui retourne un objet de type **Rectangle**, classe que nous avons créée précédemment.

Point	FabriqueRectangle
x	
y	
	petitRectangleAvec2Points(Point , Point) : Rectangle

Le principe de la méthode *petitRectangleAvec2Points* est de créer un rectangle à partir de deux points donnés. Ce rectangle devra être le plus petit rectangle contenant ces deux points. Autrement dit, ces deux points seront des coins du rectangle à créer.

Notons que ce rectangle pourrait avoir une hauteur ou largeur nulle si les points sont "alignés" par rapport à l'axe X ou Y.

Pour rappel, un rectangle est défini par les coordonnées x et y du coin supérieur gauche ainsi que par sa largeur et sa hauteur.

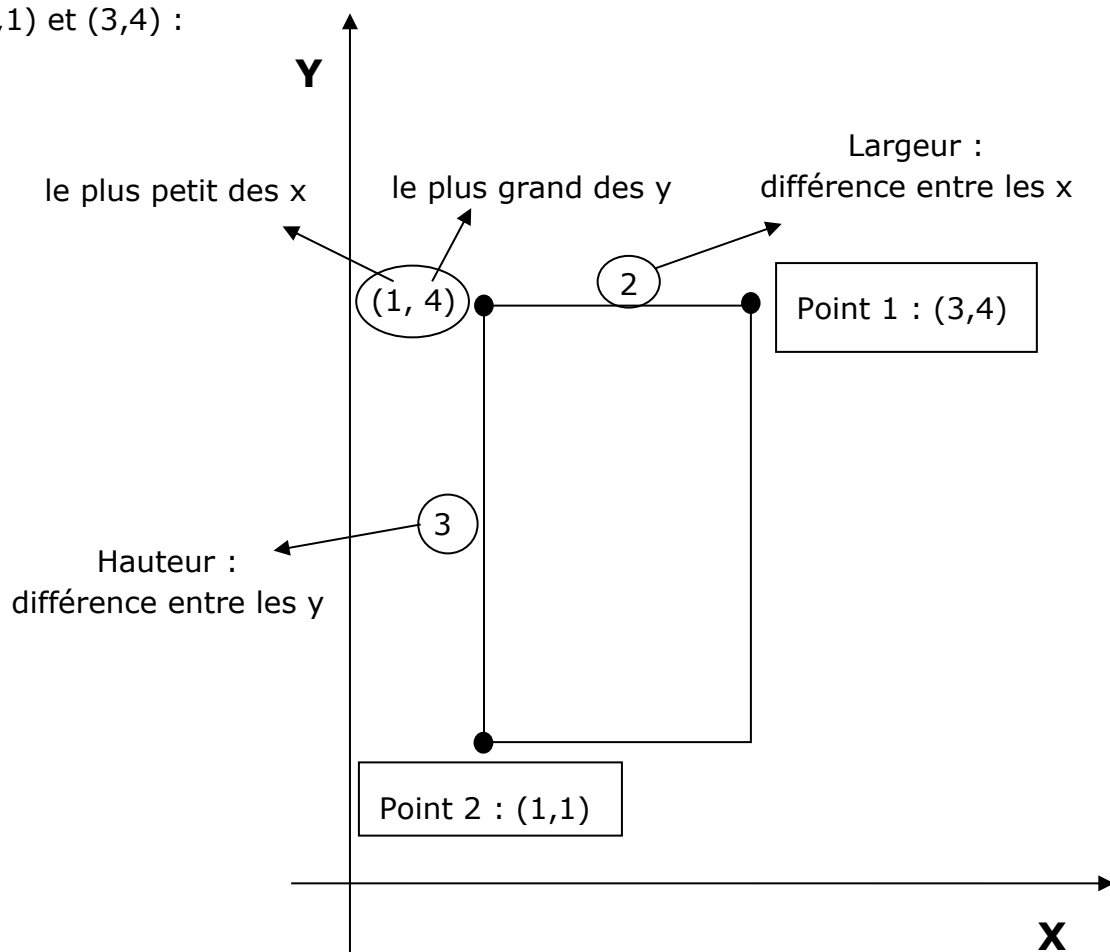
Les coordonnées x et y, la largeur et la hauteur du nouveau rectangle à créer à partir de deux points donnés sont calculés comme suit :

- La coordonnée x du coin supérieur gauche du nouveau rectangle est la plus petite des coordonnées x des deux points ;
- La coordonnée y du coin supérieur gauche du nouveau rectangle est la plus grande des coordonnées y des deux points ;
- La largeur du nouveau rectangle est égale à (la valeur absolue de) la différence entre les coordonnées x des deux points ;
- La hauteur du nouveau rectangle est égale à (la valeur absolue de) la différence entre les coordonnées y des deux points.

La méthode *petitRectangleAvec2Points* retourne un objet de type **Rectangle**, autrement dit, une référence vers un objet de type **Rectangle**. Cet objet de type Rectangle est créé dans le code de la méthode *petitRectangleAvec2Points*. La méthode ***petitRectangleAvec2Points*** contiendra donc l’instruction :

... **new Rectangle(...)**.

En guise d’illustration, construisons le plus petit rectangle contenant les points (1,1) et (3,4) :



```

class Point {
    int x;
    int y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

```

```

class FabriqueRectangle {
    Rectangle petitRectangleAvec2Points(Point point1, Point point2) {
        return new Rectangle(point1.x < point2.x ? point1.x : point2.x,
                               point1.y > point2.y ? point1.y : point2.y,
                               Math.abs(point1.x-point2.x),
                               Math.abs(point1.y-point2.y));
    }    // N.B. L'instruction Math.abs(...) retourne la valeur absolue.
}

```

```

class Principal {
    public static void main(String [] args) {
        FabriqueRectangle fabrique = new FabriqueRectangle();
        Point point1 = new Point(1,-2);
        Point point2 = new Point(4,-4);
        Rectangle rectangle;
        rectangle = fabrique.petitRectangleAvec2Points(point1,point2);
        System.out.println( "X = " + rectangle.coordonneeX
                               + ", Y = " + rectangle.coordonneeY
                               + ", largeur = " + rectangle.largeur
                               + ", hauteur = " + rectangle.hauteur);
    }
}

```

⇒ Affiche à l'écran : *X = 1, Y = -2, largeur = 3, hauteur = 2*

```

Point point3 = new Point(2,1);
Point point4 = new Point(4,5);
rectangle = fabrique.petitRectangleAvec2Points(point3,point4);
System.out.println( "X = " + rectangle.coordonneeX
                     + ", Y = " + rectangle.coordonneeY
                     + ", largeur = " + rectangle.largeur
                     + ", hauteur = " + rectangle.hauteur);

```

⇒ Affiche à l'écran : *X = 2, Y = 5, largeur = 2, hauteur = 4*

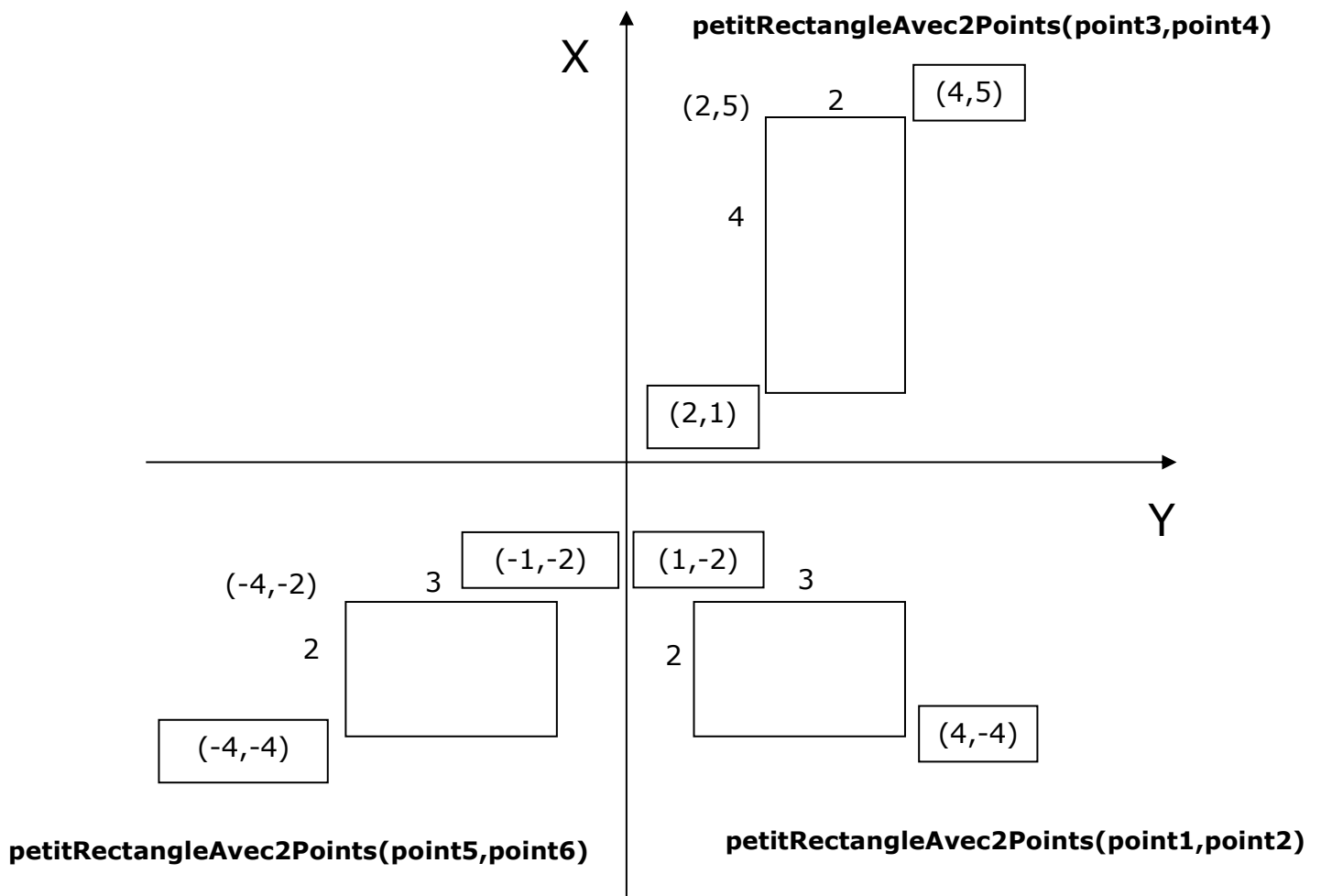
```

Point point5 = new Point(-1,-2);
Point point6 = new Point(-4,-4);
rectangle = fabrique.petitRectangleAvec2Points(point5,point6);
System.out.println( "X = " + rectangle.coordonneeX
                    + ", Y = " + rectangle.coordonneeY
                    + ", largeur = " + rectangle.largeur
                    + ", hauteur = " + rectangle.hauteur);

```

⇒ Affiche à l'écran : $X = -4$, $Y = -2$, $largeur = 3$, $hauteur = 2$

Les rectangles construits dans la classe *Principal* sont illustrés sur le graphique suivant.



3 Protections et Information Hiding

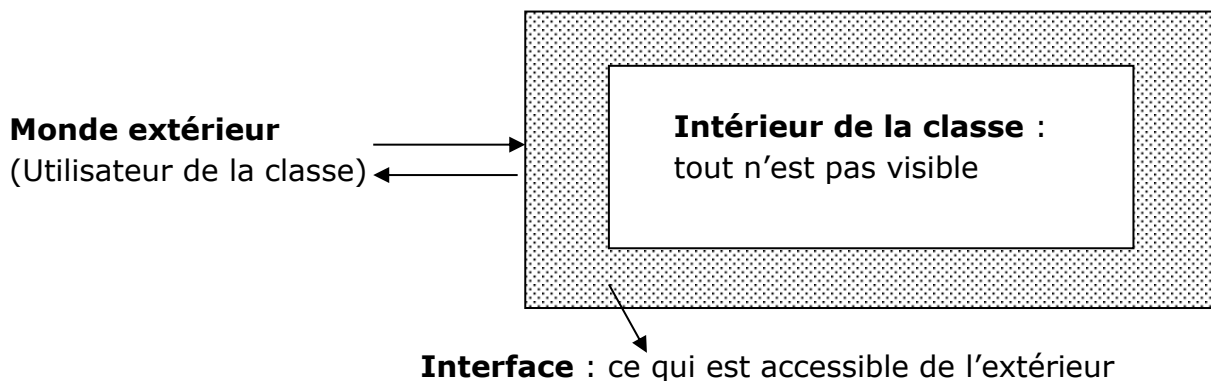
3.1 Information Hiding : protections *private* et *public*

En général, on essaye de diminuer le temps de programmation en réutilisant le plus possible des composants existants. En O.O., un programmeur peut par exemple réutiliser des classes qu'il a créées précédemment ou des classes créées par d'autres programmeurs.

Pour réutiliser des classes écrites par d'autres programmeurs, il faut savoir ce qu'on peut réutiliser dans ces classes. En effet, le concepteur (programmeur) d'une classe peut décider de cacher certaines informations qui sont internes à la classe et d'en rendre publiques d'autres. Seules les informations rendues publiques sont accessibles, et donc manipulables, par d'autres programmeurs.

Les informations qui sont cachées sont déclarées privées : il s'agit d'informations qui sont nécessaires au bon fonctionnement du composant mais qu'on ne désire pas rendre visibles du monde extérieur. En effet, le concepteur d'un composant peut vouloir cacher en partie l'implémentation de son composant.

On parlera d'interface pour désigner ce qui dans une classe est accessible par le monde extérieur. On entend par monde extérieur, tout utilisateur de la classe, c'est-à-dire tout programmeur qui créera une instance de la classe et l'utilisera.



Tout concepteur d'une classe décide du type d'accès (du type de protection) qu'il attribue aux **variables d'instances**, **aux constructeurs** et **aux méthodes**. En définissant les protections ou types d'accès, il délimite l'interface à laquelle le monde extérieur aura accès.

Plusieurs types d'accès sont possibles.

En voici deux premiers :

- **public** : accessible par le monde extérieur ;
- **private** : accessible seulement à l'intérieur de la classe
↳ inaccessible de l'extérieur (autres programmeurs, autres classes...)

Dans le monde de la programmation en Java, certaines **conventions** internationales sont respectées. Par convention, **souvent les variables d'instance d'une classe sont déclarées *private***. Selon cette convention, aucun programmeur utilisant une classe créée par un autre concepteur n'a donc accès aux variables d'instance de cette classe.

N.B. Il est possible cependant de choisir d'autres protections que *private* pour les variables d'instance (voir plus loin). Dans tous les cas, le programmeur doit réfléchir et décider de ce qu'il souhaite rendre accessible comme variable d'instance en dehors de la classe.

Si le concepteur d'une classe désire permettre l'accès aux informations enregistrées dans certaines variables d'instance privées, il prévoit des méthodes publiques qui se chargeront des accès en lecture/écriture à ces variables d'instance.

Notons que bon nombre de Frameworks¹ se basent sur cette convention lorsqu'ils doivent créer des objets de façon automatique (cf. entre autres les principes d'injection de dépendance et d'inversion de contrôle abordés au bloc 3).

Le principe qui consiste à cacher certaines informations porte le nom anglais de **Information Hiding**.

¹ Un Framework peut se traduire par un cadre de travail visant à simplifier le travail des développeurs, en les guidant dans leur conception, en leur proposant par exemple des squelettes d'architecture ou des patrons de conception.

3.2 Méthodes publiques d'accès aux variables d'instance privées : getters et setters

Par convention donc, toutes les variables d'instance sont déclarées **private**.

Cependant, le concepteur de la classe a la possibilité de permettre l'accès aux informations contenues dans certaines variables d'instance, et ce, en prévoyant des méthodes publiques qui y accèdent. L'accès aux informations contenues dans les **variables d'instance déclarées private** se fera donc via l'appel à des **méthodes déclarées public**.

Par convention :

- Les méthodes d'accès en **lecture** aux variables d'instance privées
 - sont appelées **getters** en anglais
 - Sélecteurs ou accesseurs en français
 - portent un nom commençant par **get**
 - Suivi du nom de la variable d'instance (+ Camel Case)
 - retourne en général la valeur de la variable d'instance
- Les méthodes d'accès en **écriture** aux variables d'instance privées
 - sont appelées **setters** en anglais
 - Modificateurs ou mutateurs en français
 - portent un nom commençant par **set**
 - Suivi du nom de la variable d'instance (+ Camel Case)
 - modifie la valeur de la variable d'instance à partir d'une valeur permise (cf. le rôle de filtre)

Prenons comme exemple la classe *Rectangle*. Appliquons le principe de l'Information Hiding : déclarons toutes les variables d'instance **private**.

Il paraît intéressant de permettre au programmeur qui utilise une telle classe de pouvoir obtenir les coordonnées x et y du point d'ancrage ainsi que la largeur et la hauteur de tout rectangle qu'il aurait créé.

De plus, on voudrait également permettre à ce programmeur de modifier quand il le désire ces caractéristiques.

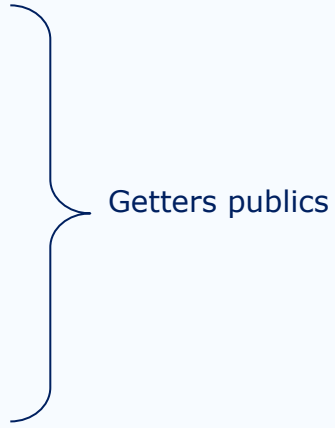
Supposons, pour les besoins de l'exemple, qu'il soit permis au programmeur qui a créé un rectangle, d'obtenir son périmètre et sa surface et de le déplacer ensuite mais pas de l'agrandir ni l'élargir (juste pour les besoins de l'exemple).

Pour rappel, dans les schémas UML, les protections sont symbolisées par "+" (accès public) et "-" (accès privé).

Rectangle
- coordonneeX - coordonneeY - largeur - hauteur
+ perimetre() + surface() - elargir(int) - agrandir(int) + deplacerEn(int,int)

Dans la classe *Rectangle* sont donc prévues 4 méthodes de type getters et 4 méthodes de type setters, toutes étant déclarées **public** (non représentées dans le diagramme de classe UML ci-dessus). Le constructeur et les méthodes *surface*, *perimetre* et *deplacerEn* seront déclarées **public** tandis que les méthodes *elargir* et *agrandir* seront déclarées **private**.

```
public class Rectangle {  
    private int coordonneeX;  
    private int coordonneeY;  
    private int largeur;  
    private int hauteur;  
  
    public Rectangle(int coordonneeX, int coordonneeY, int largeur,  
                     int hauteur) {  
        this.coordonneeX = coordonneeX;  
        this.coordonneeY = coordonneeY;  
        this.largeur = largeur;  
        this.hauteur = hauteur;  
    }  
    public int getCoordonneeX() {  
        return coordonneeX;  
    }  
    public int getCoordonneeY() {  
        return coordonneeY;  
    }  
    public int getLargeur() {  
        return largeur;  
    }  
    public int getHauteur() {  
        return hauteur;  
    }  
}
```



```

public void setCoordonneeX(int nouvelleCoordonneeX) {
    coordonneeX = nouvelleCoordonneeX;
}
public void setCoordonneeY(int nouvelleCoordonneeY) {
    coordonneeY = nouvelleCoordonneeY;
}
public void setLargeur(int nouvelleLargeur) {
    largeur = nouvelleLargeur;
}
public void setHauteur(int nouvelleHauteur) {
    hauteur = nouvelleHauteur;
}
public int perimetre() {
    return 2* (largeur + hauteur);
}
public int surface() {
    return largeur * hauteur;
}
private void elargir(int augmentation) {
    largeur += augmentation;
}
private void agrandir(int augmentation) {
    hauteur += augmentation;
}
}
public void deplacerEn( int nouvelleCoordonneeX,
                        int nouvelleCoordonneeY) {
    coordonneeX = nouvelleCoordonneeX;
    coordonneeY = nouvelleCoordonneeY;
}
}

```

Setters publics

Ce qui est déclaré **private** reste cependant toujours **accessible au sein de la classe**. C'est pourquoi les variables d'instance pourtant déclarées **private** sont accessibles en lecture et en écriture dans les constructeurs et les méthodes.

Exemple

```
|| coordonneeX = nouvelleCoordonneeX;
```

Qu'en est-il de l'utilisation de la classe *Rectangle*, éventuellement par un autre programmeur, pour créer des objets et les manipuler ?


```

public class Principal {
    public static void main(String [] args) {
        Rectangle rectangle;           // Déclaration OK car classe public
        rectangle = new Rectangle(0,0,4,2); // OK car constructeur public

        System.out.println("X : " + rectangle.coordonneeX
            + "\nY : " + rectangle.coordonneeY
            + "\nLargeur : " + rectangle.largeur
            + "\nHauteur : " + rectangle.hauteur);
            // Pas OK car variables déclarées private

        System.out.println("X : " + rectangle.getCoordonneeX()
            + "\nY : " + rectangle.getCoordonneeY()
            + "\nLargeur : " + rectangle.getLargeur()
            + "\nHauteur : " + rectangle.getHauteur());
            // OK car getters déclarés public

        rectangle.coordonneeX = 1; // Pas OK car coordonneeX déclarée private
        rectangle.coordonneeY = 2; // Pas OK car coordonneeY déclarée private
        rectangle.largeur = 3; // Pas OK car largeur déclarée private
        rectangle.hauteur = 4; // Pas OK car hauteur déclarée private
        rectangle.setCoordonneeX(1); // OK setCoordonneeX(...) déclarée public
        rectangle.setCoordonneeY(2); // OK setCoordonneeY(...) déclarée public
        rectangle.setLargeur(3); // OK setLargeur(...) déclarée public
        rectangle.setHauteur(4); // OK setLHauteur(...) déclarée public

        System.out.println("surface : " + rectangle.surface());
            // OK car surface() déclarée public

        System.out.println("Périmètre : " + rectangle.perimetre());
            // OK car perimetre() déclarée public

        rectangle.elargir(5); // Pas OK car elargir(...) déclarée private
        rectangle.agrandir(10); // Pas OK car agrandir(...) déclarée private
        rectangle.deplacerEn(8,12); // OK car deplacerEn(...) déclarée public
    }
}

```

La protection **public** a été attribuée à la classe *Rectangle*. Cela a pour effet de la rendre accessible du monde extérieur. La déclaration : **public class Rectangle** permet de déclarer des variables de type *Rectangle* en dehors de cette classe (à l'extérieur à la classe), par exemple dans la classe *Principal*.

Attention, la protection **private** ne peut être associée à une classe (à l'exception toutefois d'une classe interne à une autre classe (cf. Bloc 2)) !

Le constructeur à 4 arguments est déclaré **public** : on peut donc créer des occurrences de rectangle en y faisant appel dans une autre classe, par exemple la classe *Principal*.

Dans le code précédent, le fait d'accéder directement en lecture ou en écriture aux variables d'instance *coordonneeX*, *coordonneeY*, *largeur* ou *hauteur* provoque une erreur à la compilation, car ces variables d'instance sont déclarées **private**. Il faut donc y accéder en lecture via, respectivement, les getters **publics** *getCoordonneeX*, *getCoordonneeY*, *getLargeur* et *getHauteur*, et y accéder en écriture via, respectivement, les setters **publics** *setCoordonneeX*, *setCoordonneeY*, *setLargeur* et *setHauteur*.

Les méthodes *surface*, *perimetre* et *deplacerEn* peuvent être appelées sur des objets de type rectangle créés dans une autre classe (*Principal*). Elles sont en effet déclarées **public**. Les méthodes *elargir* et *agrandir* quant à elles ne peuvent être appelées sur de tels objets : elles sont déclarées **private**.

En conclusion

- ① Le concepteur d'une classe décide de ce qu'il cache et de ce qu'il rend visible au monde extérieur (c'est-à-dire accessible du monde extérieur).
- ② Ce qui est **accessible du monde extérieur** est déclaré **public**
- ③ Ce qui doit être **caché au monde extérieur** est déclaré **private**. Ce qui est déclaré *private* reste cependant accessible au sein de la classe.
- ④ Par convention, **toutes les variables d'instance** sont déclarées **private**
- ⑤ Si le concepteur de la classe désire permettre l'accès en lecture/écriture à certaines variables d'instance, il prévoit des méthodes déclarées **public** qui y permettent l'accès.
- ⑥ Par convention, les méthodes publiques permettant l'accès en **lecture** aux variables d'instance déclarées *private* sont des **getters** (terme anglais). Le nom de ces méthodes débute par **get** suivi du nom de la variable d'instance à laquelle elle permet l'accès en lecture. Ces méthodes ne demandent **aucun argument** supplémentaire et retournent habituellement une valeur du type de la variable d'instance à laquelle on accède.

⑦ Par convention, les méthodes publiques permettant l'accès en **écriture** aux variables d'instance déclarées *private* sont des **setters** (terme anglais). Le nom de ces méthodes débute par **set** suivi du nom de la variable d'instance à laquelle elle permet l'accès en écriture. Une méthode de type setter reçoit en argument la valeur à affecter à la variable d'instance correspondante. Habituellement, ces méthodes ne retournent aucune valeur.

3.3 Intérêt des setters : rôle de filtre

Les setters peuvent jouer le **rôle de filtre** et empêcher que l'on attribue une valeur erronée ou non permise aux variables d'instance. En effet, l'affectation de valeurs non permises à des variables d'instance pourrait avoir pour effet que certaines méthodes ne puissent pas s'exécuter voire provoquent une erreur à l'exécution (par exemple, une division par 0).

Exemple 1

Soit une classe permettant de gérer des lots d'articles. Un lot n'a de sens que s'il contient au moins un article. Attribuer un nombre d'articles égal à 0 provoquera une erreur lors de l'appel de la méthode *prixMoyenArticle* qui divise le prix du lot par le nombre d'articles (\Rightarrow division par 0). Afin d'éviter ce cas d'erreur, on peut choisir de n'admettre qu'un nombre d'articles dans le lot supérieur strictement à 0. Toute tentative d'affecter une valeur ≤ 0 pour le nombre d'articles aura pour effet que ce nombre sera automatiquement mis à 1 par le setter correspondant. Notons que cette protection, **ce filtre, n'est efficace que si on appelle ce setter dans tout constructeur**. Dans le cas contraire, il serait toujours possible de créer via le constructeur un lot avec un nombre d'articles ≤ 0 .

```
public class LotArticles {
    private String libelleLot;
    private int nombreArticles;
    private double prixLot;
    public void setNombreArticles(int nombreArticles) {
        if (nombreArticles <= 0) {
            this.nombreArticles = 1;
        }
        else {
            this.nombreArticles = nombreArticles;
        }
    }
    public LotArticles(String libelleLot, int nombreArticles,
                       double prixLot) {
        this.libelleLot = libelleLot;
        this.nombreArticles = nombreArticles;
        ↗ setNombreArticles(nombreArticles);
        this.prixLot = prixLot;
    }
}
```

LotArticles
- libelleLot - nombreArticles - prixLot
+ prixMoyenArticle()

```

public double prixMoyenArticle() {
    return prixLot / nombreArticles;
}
}

```

Exemple 2

La classe *Personne* contient une variable d'instance *genre*. Imaginons que celle-ci soit utilisée dans la méthode *titre* qui retourne "Monsieur", "Madame" ou la chaîne de caractères vide en fonction du genre, respectivement *M*, *F* ou *X*. Il faudrait donc prévoir un setter qui filtre les valeurs affectées à *genre* et qui n'accepte que les valeurs *M*, *F* ou *X*. Pour toute autre valeur que l'on tentera d'affecter à *genre*, le setter correspondant affectera automatiquement la valeur 'X' par défaut ; cela aura pour effet que la méthode *titre* retournera la chaîne de caractères vide par défaut, c'est-à-dire pour toute valeur de *genre* non admise.

Rappelons que ce filtre n'est efficace que si on appelle ce setter dans tout constructeur.

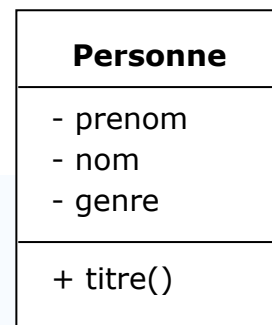
```

public class Personne {
    private String prenom;
    private String nom;
    private char genre;

    public void setGenre(char genre) {
        if (genre == 'M' || genre == 'F') {
            this.genre = genre;
        }
        else {
            this.genre = 'X';
        }
        // ou this.genre = (genre=='M' || genre=='F'?genre:'X');
    }

    public Personne(String prenom, String nom, char genre) {
        this.prenom = prenom;
        this.nom = nom;
        this.genre = genre;    ⇨    setGenre(genre);
    }
}

```



```
public String titre() {  
    return (genre=='F')?"Madame":((genre=='M')?"Monsieur":"","");  
}  
}
```

```
public class Principal {  
    public static void main(String [] args) {  
        Personne sacha = new Personne("Sacha", "Petit", 'F');  
        System.out.println("Bonjour " + sacha.titre());  
    }  
}
```

⇒ Affiche à l'écran : *Bonjour Madame*

```
sacha = new Personne("Sacha", "Petit", 'U');  
System.out.println("Bonjour " + sacha.titre());
```

⇒ Affiche à l'écran : *Bonjour*

En conclusion

- ① Les setters peuvent jouer le **rôle de filtre** et empêcher que l'on attribue une valeur erronée ou non permise aux variables d'instance.
- ② Un filtre n'est efficace que si on **appelle le setter qui joue le rôle de filtre dans tout constructeur et toute méthode qui modifie la variable correspondante.**

3.4 Introduction à la gestion des exceptions

Il est à noter qu'il existe un mécanisme plus élaboré pour gérer les cas d'erreurs.

Quand une méthode détecte un cas d'erreur, elle peut refuser l'exécution de la méthode et prévenir le niveau appelant de l'échec de l'exécution de la méthode.

Ceci est possible grâce au **mécanisme des exceptions**.

Le principe est le suivant.

- Lorsqu'un cas d'erreur est détecté, un **objet d'une classe de type Exception est créé** ; celui-ci renferme des informations sur l'erreur qui s'est produite.
- La suite de la méthode d'où provient l'erreur n'est alors pas exécutée.
- Le code qui **appelle** une méthode susceptible de lever une exception (de détecter un cas d'erreur) doit être précédé du mot clé "**try**". En effet, on "essaye" d'exécuter la méthode.
- S'il n'y a pas d'erreur, la méthode est exécutée entièrement.
- Dans le cas contraire (c'est-à-dire s'il y a une exception, un cas d'erreur), la méthode appelée est abandonnée et on peut préciser le code à exécuter en cas d'erreur.

Ce principe peut être résumé comme suit :

Programmeur 1

```
public class NomClasse {  
    ...  
    public typeRetour methodeX( ... ) throws ClasseException {  
        if ( ... )    ⇒ Détection d'un cas d'erreur  
        ... new ClasseException(...);    ⇒ Création d'un objet d'une classe de type Exception  
        ...    ⇒ Suite de la méthode si on n'est pas dans le cas d'erreur  
    }  
}
```

Programmeur 2

```
NomClasse objet = new NomClasse(...);  
try {  
    objet.methodeX(...);  
    ↪ Appel d'une méthode susceptible de détecter un cas d'erreur  
}  
catch ( ... ClasseException ) {  
    ... ⇒ Code à exécuter en cas d'erreur  
}
```

Exemple

```
public class Calculette {  
    public double resultatDivision(double dividende, double diviseur)  
                                throws ExceptionDivision {  
        if (diviseur == 0) {  
            throw new ExceptionDivision();  
        }  
        else {  
            return dividende / diviseur;  
        }  
    }  
}
```

La classe *ExceptionDivision* doit bien évidemment être écrite par le programmeur.

```
public class Principal {  
    public static void main(String[] args) {  
        try {  
            Calculette calculette = new Calculette();  
            System.out.println(calculette.resultatDivision(100.0, 2.0));  
        }  
        catch (ExceptionDivision exception) {  
            System.out.println("Erreur lors de la division !");  
        }  
    }  
}
```

⇒ Affiche : 50.0


```

public class Principal {
    public static void main(String[] args) {
        try {
            Calculette calculette = new Calculette();
            System.out.println(calculette.resultatDivision(100.0, 0.0));
        }
        catch (ExceptionDivision exception) {
            System.out.println("Erreur lors de la division !");
        }
    }
}

```

⇒ Affiche : *Erreur lors de la division !*

Grâce à ce mécanisme, on pourrait imaginer des setters refusant de modifier l'état d'un objet à partir d'arguments non recevables (valeurs invalides).

Exemple

```

public class LotArticles {
    private String libelleLot;
    private int nombreArticles;
    private double prixLot;

    ... // Constructeur et méthodes

    public void setNombreArticles(int nombreArticles)
        throws ExceptionNombreArticlesDansLot {
        if (nombreArticles <= 0)
            throw new ExceptionNombreArticlesDansLot();
        else
            this.nombreArticles = nombreArticles;
    }

    public int getNombreArticles() {
        return nombreArticles;
    }
}

```

```

public class Principal {
    public static void main(String[ ] args) {
        try {
            LotArticles lot = new LotArticles(...);
            lot.setNombreArticles(20);
            System.out.println(lot.getNombreArticles());
        }
        catch (ExceptionNombreArticlesDansLot exception) {
            System.out.println("Erreur sur le nombre d'articles dans le lot");
        }
    }
}

```

⇒ Affiche : 20

```

public class Principal {
    public static void main(String[ ] args) {
        try {
            LotArticles lot = new LotArticles(...);
            lot.setNombreArticles(0);
            System.out.println(lot.getNombreArticles());
        }
        catch (ExceptionNombreArticlesDansLot exception) {
            System.out.println("Erreur sur le nombre d'articles dans le lot !");
        }
    }
}

```

⇒ Affiche : *Erreur sur le nombre d'articles dans le lot !*

La gestion des cas d'erreurs via le mécanisme des exceptions sera étudiée en profondeur dans le cours de programmation orientée objet avancée (bloc 2).

3.5 Notion de package (+ import)

Les classes constituant un programme sont généralement rassemblées dans un même répertoire appelé **package**.

Notons qu'il n'est pas recommandé de travailler dans le package *default*. En effet, il est impossible d'importer des classes du package par défaut dans un autre package du même projet.

Il faut placer les classes dans un package différent du package par défaut si on souhaite pouvoir réutiliser ces classes dans un autre projet. Il est donc conseillé de créer (au moins) un nouveau package pour chaque nouveau projet.

Un nom de package commence par une minuscule.

Il est possible d'importer des classes existantes situées dans un autre répertoire (dans un autre package). On peut par exemple importer des classes prédéfinies dans le langage Java ou des classes créées précédemment pour un autre programme.

Pour pouvoir accéder à une classe située dans un autre package, il faut l'importer via l'instruction :

import specificationDuPackage.NomClasse ;

où **specificationDuPackage** est l'adresse du package (adressage absolu ou relatif) et où **NomClasse** est le nom de la classe qu'on veut importer.

Si l'on désire importer non pas une, mais l'ensemble des classes du package, il faut écrire :

import specificationDuPackage.* ;

Exemples d'adressage

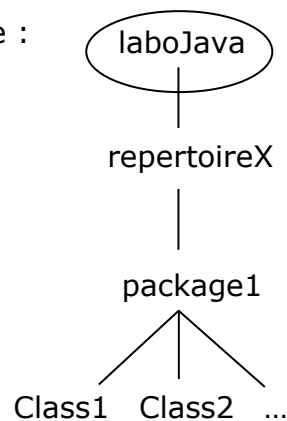
1.

```
|| import java.time.LocalDate;
```

⇒ Importe la classe *LocalDate* qui se trouve dans le package *time* du répertoire *java*, lui-même situé dans le répertoire (racine) contenant tous les répertoires des classes prédéfinies du langage Java. L'adresse du répertoire racine des classes prédéfinies du langage Java est connue de la machine virtuelle.

N.B. La classe *LocalDate* facilite la gestion des dates.

2. Soit la structure de répertoires suivante :



Si le projet est placé dans le répertoire *laboJava*, et si le package à importer s'appelle *package1* et qu'il se trouve dans le sous-répertoire *repertoireX*, l'instruction d'import à écrire dans n'importe quelle classe du même projet est :

```
|| import repertoireX.package1.*;
```

L'adresse du package est ici donnée en adressage relatif : on ne précise que l'adresse du chemin à partir du répertoire associé au projet.

Toute instruction d'import doit être placée entre l'instruction spécifiant le nom du package de la classe que l'on est en train de créer et l'instruction débutant la déclaration de ladite classe.

Exemple

```
|| package rectanglePackage;
|| public class Rectangle {
||     private int coordonneeX;
||     private int coordonneeY;
||     private int largeur;
||     private int hauteur;
||
||     ...
|| }
```

Imaginons que les classes *Rectangle* et *Principal* soient placées dans des packages différents : la classe *Rectangle* est déclarée dans le package

rectanglePackage et la classe *Principal* est déclarée dans le package **autrePackage**.

```
package autrePackage ;

import rectanglePackage.* ;
    // ↳ Importe toutes les classes du package rectanglePackage

public class Principal {
    public static void main(String [] args) {
        Rectangle rectangle;    // OK car la classe Rectangle est déclarée public
        rectangle = new Rectangle(1,2,4,3) ; // OK car constructeur public

        System.out.println("X : " + rectangle.coordonneeX
                            + "\nY : " + rectangle.coordonneeY
                            + "\nLargeur : " + rectangle.largeur
                            + "\nHauteur : " + rectangle.hauteur);
                                // Pas OK car variables déclarées private

        System.out.println("X : " + rectangle.getCoordonneeX()
                            + "\nY : " + rectangle.getCoordonneeY()
                            + "\nLargeur : " + rectangle.getLargeur()
                            + "\nHauteur : " + rectangle.getHauteur());
                                // OK car getters déclarés public

        rectangle.coordonneeX = 1;    // Pas OK car coordonneeX déclarée private
        rectangle.coordonneeY = 2;    // Pas OK car coordonneeY déclarée private
        rectangle.largeur = 3;        // Pas OK car largeur déclarée private
        rectangle.hauteur = 4;        // Pas OK car hauteur déclarée private
        rectangle.setCoordonneeX(1);  // OK car setCoordonneeX(...) déclarée public
        rectangle.setCoordonneeY(2);  // OK car setCoordonneeY(...) déclarée public
        rectangle.setLargeur(3);      // OK car setLargeur(...) déclarée public
        rectangle.setHauteur(4);      // OK car setLHauteur(...) déclarée public

        System.out.println("surface : " + rectangle.surface());
                                // OK car surface() déclarée public

        System.out.println("Périmètre : " + rectangle.perimetre());
                                // OK car perimetre() déclarée public

        rectangle.elargir(5);         // Pas OK car elargir(...) déclarée private
        rectangle.agrandir(10);       // Pas OK car agrandir(...) déclarée private
        rectangle.deplacerEn(8,12);   // OK car deplacerEn(...) déclarée public
    }
}
```

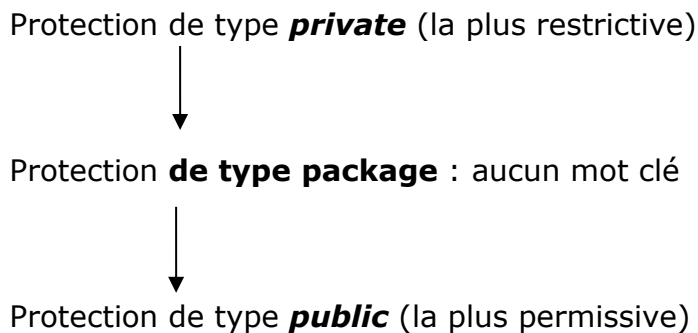
Attention, ce qui a été dit précédemment à propos des protections (*public* et *private*) reste valable. Les variables d'instance de la classe *Rectangle* étant déclarées ***private***, il faut utiliser les getters ou setters (de la classe importée) qui sont déclarés ***public***. En effet, on a seulement accès aux composants déclarés *public* depuis le monde extérieur. Le monde extérieur du point de vue de la classe *Rectangle* est la classe qui importe le package ***rectanglePackage***, à savoir la classe *Principal*.

3.6 Protection par défaut : protection de type package

Les deux protections déjà étudiées sont **private** et **public**. Il en existe une troisième qui est la protection **de type package**. La protection de type package est la **protection par défaut**. Contrairement aux protections *private* et *public*, la protection de type package **ne nécessite aucun mot clé**. Le fait de n'écrire aucune protection explicite devant un nom de classe, une variable d'instance, une méthode ou constructeur, y associe implicitement la protection par défaut, à savoir, la protection de type package.

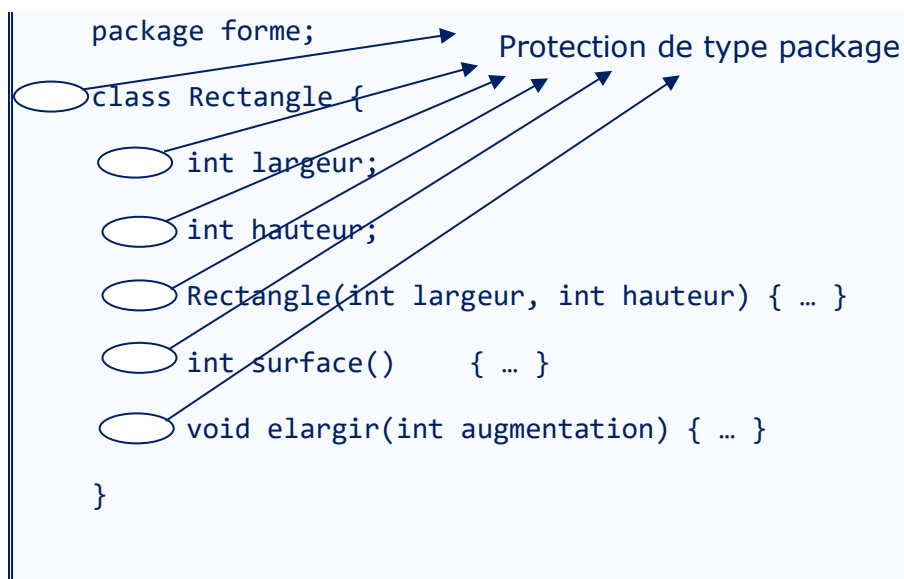
Tout ce qui a la protection de type **package est accessible** par n'importe quelle classe qui fait partie du **même package**.

La protection de type package se situe donc entre la protection de type *private* (très restrictive) et la protection de type *public* (très permissive) :



Exemple :

Soient les classes *Rectangle* et *Principal* déclarées dans le même package.



```

package forme ;
public class Principal {
    public static void main(String [] args) {
        Rectangle premierRectangle;
        premierRectangle = new Rectangle(3, 5);
        System.out.println("hauteur : " + premierRectangle.hauteur + " cm");
        System.out.println("surface : " + premierRectangle.surface() + " cm");
    }
}

```

Aucune protection n'étant explicitement associée à la classe *Rectangle*, aux variables d'instance, au constructeur et aux méthodes, ces derniers reçoivent la protection par défaut, à savoir la protection de type package.

La classe *Principal* étant déclarée dans le même package que la classe *Rectangle*, elle a donc accès à tout ce qui est déclaré avec la protection package dans la classe *Rectangle*. Elle peut notamment déclarer un objet de type *Rectangle*, car la classe *Rectangle* a la protection package. Le constructeur peut aussi être appelé, ainsi que la méthode *surface*. L'accès aux variables d'instance est aussi permis.

N.B. Notons qu'importer une classe dont les variables d'instance et méthodes auraient une protection de type package ne serait d'ailleurs d'aucune utilité : étant dans un autre package, on ne pourrait de toutes façons pas y avoir accès.

En conclusion

- ① La protection par défaut est de type *package* ; elle n'est associée à aucun mot clé.
- ② Tout ce qui a la protection de type **package est accessible** par n'importe quelle classe qui fait partie du **même package**.
- ③ Les protections étudiées jusqu'à présent sont, de la plus restrictive à la plus permissive, *private* (accès uniquement de l'intérieur de la classe), *package* et *public* (accessible par tout le monde).

4 Interface utilisateur

4.1 Types d'interface utilisateur

L'affichage des données à destination de l'utilisateur peuvent prendre différentes formes : affichage à la console, présentation à travers des composants graphiques en Java (composants Swing, JavaFX...), affichage via des pages HTML, présentation via des écrans d'applications mobiles (cf. fichiers.xml décrivant le design des pages en Android)...

4.2 Entrées – sorties à la console

4.2.1 Entrées

Les instructions permettant de saisir des données à partir du clavier sont :

```
import java.util.Scanner;
...
Scanner clavier = new Scanner(System.in);
int entier = clavier.nextInt();           // Pour des entiers
String texte = clavier.next();            // Pour des chaînes de caractères
boolean booleen = clavier.nextBoolean(); // Pour des booléens
float reelFloat = clavier.nextFloat();    // Pour des réels de type float
double reelDouble = clavier.nextDouble(); // Pour des réels double
...
```

4.2.2 Sorties

Pour rappel, l'instruction d'affichage à la console d'une chaîne de caractères est :

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

4.3 Séparation vue – modèle

On privilégie la réutilisation des classes qu'on appelle classes modèles, comme par exemple les classes *Personne*, *Rectangle*, *Vehicule*, *EtudiantInformatique*... Ces classes font partie de ce qu'on appelle la **couche modèle** (Model Layer en anglais).

Le but est la réutilisation d'une même classe (par exemple, la classe *Personne*) dans des applications avec **interfaces utilisateurs différentes** : affichage à la console, dans des composants graphiques en Java (composants Swing, JavaFX...), dans des pages HTML, dans des écrans d'application Android...

Par conséquent on délègue à d'autres composants / classes la tâche de gérer l'interface utilisateur (affichage des données et réception des données introduites par l'utilisateur). Ces composants, ces classes font partie de ce qu'on appelle la **couche vue** (View Layer en anglais).

On évitera donc de prévoir dans les classes modèles des méthodes qui fixent ("hardcodent") les canaux de communication avec l'utilisateur, comme par exemple les sorties sur la console. On évitera les instructions de type *System.out.println* ou d'utiliser la classe *Scanner* dans les méthodes des classes modèles.

Bonne pratique :

Comment faire si une chaîne de caractères décrivant un objet modèle doit être affichée à l'utilisateur ?

- Dans la classe modèle
 - Prévoir une méthode avec un type de retour *String* retournant la chaîne de caractères
- Dans la classe qui se charge de la gestion de l'interface avec l'utilisateur (couche vue), par exemple la méthode *main*
 - Appeler cette méthode du modèle sur l'objet correspondant et afficher à l'utilisateur la chaîne de caractères ainsi obtenue.

Exemple

```
public class Personne {  
    private String prenom;  
    private String nom;  
    private char genre;  
    ...  
    public String titre() {  
        return (genre=='F')?"Madame":((genre=='M')?"Monsieur":"","");  
    }  
}
```

```
public class Principal {  
    public static void main(String [] args) {  
        Personne sacha = new Personne("Sacha","Petit", 'F');  
        System.out.println(sacha.titre());  
    }  
}
```

4.4 La méthode *toString*

Il s'agit d'une méthode dont le nom est un mot réservé : *toString*, littéralement : "d'un objet vers une chaîne de caractères", autrement dit, à partir d'un objet, on produit une chaîne de caractères le décrivant.

Sa déclaration est immuable :

```
public String toString() {  
    ...  
}
```

Cette méthode permet de décrire tout objet de n'importe quelle classe sous la forme d'une chaîne de caractères.

De plus, elle est appelée **automatiquement** par Java chaque fois qu'un nom d'objet apparaît "là où on attend" une chaîne de caractères.

Exemple 1

```
public class Rectangle {  
    private int coordonneeX;  
    private int coordonneeY;  
    private int largeur;  
    private int hauteur;  
  
    ...           // Constructeurs et méthodes  
  
    public String toString() {  
        return "Point d'ancrage : (" + coordonneeX + ", " + coordonneeY + ")"  
            + "\nLargeur : " + largeur + "\nHauteur : " + hauteur;  
    }  
}
```

La méthode *toString* est destinée à décrire tout objet de la classe *Rectangle*. C'est le programmeur (le concepteur de la classe *Rectangle*) qui décide de la façon de décrire tout rectangle, c'est-à-dire via quelle chaîne de caractères.

```
public class Principal {  
    public static void main(String [] args) {  
        Rectangle premierRectangle = new Rectangle(2, 3, 4, 5);  
        System.out.println(premierRectangle);  
    }  
}
```

↳ Appel implicite à ***premierRectangle.toString()***

⇒ Affiche à l'écran : *Point d'ancrage : (2,3)*
Largeur : 4
Hauteur : 5

```
String message = "Coordonnées du rectangle \n" + premierRectangle;
```

Appel implicite à ***premierRectangle.toString()*** ↗

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

⇒ Affiche à l'écran : *Coordonnées du rectangle*
Point d'ancrage : (2,3)
Largeur : 4
Hauteur : 5

La méthode *toString* est appelée automatiquement en Java dès qu'un nom d'objet est rencontré au lieu d'une chaîne de caractères. C'est le cas notamment quand un objet est passé comme argument dans l'instruction *System.out.println*. Cette instruction attend normalement une chaîne de caractères. La méthode *toString* est alors appelée sur cet objet passé en argument.

Il en va de même dans l'instruction :

```
String message = "Coordonnées du rectangle \n" + premierRectangle;
```

L'opérateur de concaténation (+) est ici sensé concaténer la chaîne de caractères *"Coordonnées du rectangle \n"* avec une autre chaîne de caractères. Or, on lui demande ici de concaténer une chaîne de caractères et un objet appelé *premierRectangle*. La méthode *toString* est appelée ici aussi automatiquement par Java sur cet objet *premierRectangle*.

Exemple 2

```
public class EtudiantInformatique {  
    private String prenom;  
    private String nom;  
    private char genre;  
    private String section;  
    private int bloc;  
    private int totalEctsLangue;  
    private int nbEctsReussisLangue;  
    private int totalEctsMath;  
    private int nbEctsReussisMath;  
    private int totalEctsProgra;  
    int nbEctsReussisProgra;
```

```

...      // Constructeurs et méthodes

public String toString() {
    return "L'étudiant " + prenom + " " + nom + " inscrit au bloc " + bloc
        + " de la section "+ section;
}
}

```

```

public class Principal {
    public static void main(String [] args) {
        EtudiantInformatique alan ;
        alan = new EtudiantInformatique("Alan","Turing",'M',
            "Informatique de gestion",2,11,5,9,0,40,20);
        System.out.println(alan);
    }
}

```

⇒ Affiche à l'écran :

L'étudiant Alan Turing inscrit au bloc 2 de la section Informatique de gestion

```

String message = alan + "\na un taux de réussite de " +
    alan.tauxReussiteCoursSpecialite()+ "% en cours de spécialité";
System.out.println(message);

```

⇒ Affiche à l'écran :

*L'étudiant Alan Turing inscrit au bloc 2 de la section Informatique de gestion
a un taux de réussite de 50% en cours de spécialité*

Notons que si la méthode *toString* n'est pas définie dans une classe, Java propose sa propre description de tout objet de la classe. La chaîne de caractères proposée par Java n'est pas très conviviale ; elle est composée de :

nomPackage.nomClasse@hashCode².

Exemple : *packageRectangle.Rectangle@45ee12a7*

Il est dès lors utile de définir la méthode *toString* dans chacune des classes que l'on écrit.

² Représentation (non signée) en hexadécimal du hash code de l'objet.

En conclusion

① Séparation Vue – Modèle : on évitera d'utiliser *System.out.println* dans les méthodes des classes qu'on utilisera comme modèle pour créer des objets. On privilégiera des méthodes qui retournent des chaînes de caractères. La méthode *main* fera appel à ces méthodes et affichera à la console leur résultat.

② La méthode *toString* retourne une chaîne de caractères qui décrit tout objet de la classe. Elle est automatiquement appelée chaque fois que l'on écrit un nom d'objet là où Java attend une chaîne de caractères.

5 Surcharge (overloading) des constructeurs et méthodes

5.1 Surcharge des constructeurs

Il peut y avoir plusieurs constructeurs dans une classe. Or, un constructeur doit impérativement porter le même nom que la classe. Plusieurs constructeurs peuvent donc coexister au sein de la même classe tout en portant le même nom.

Ce qui les différencie, c'est leur signature.

La signature d'une méthode et/ou d'un constructeur, c'est la combinaison du nom, du nombre et des types d'arguments.

Par conséquent, ce qui différencie deux constructeurs de la même classe, c'est le nombre d'arguments et/ou les types d'arguments.

Lors de l'appel à un constructeur, Java fait la différence en analysant la signature du constructeur utilisé (nombre et types d'arguments).

Exemple 1

```
public class Vehicule {
    private String plaque;
    private String modele;
    private int anneeFabrication;
    private int jaugeCarburant;

    /* Constructeur avec autant d'arguments que de variables d'instance ⇒ permet de
       créer un objet et d'initialiser toutes ses variables d'instance en même temps */
    public Vehicule(String plaque, String modele, int anneeFabrication,
                    int jaugeCarburant) {
        this.plaque = plaque;
        this.modele = modele;
        this.anneeFabrication = anneeFabrication ;
        this.jaugeCarburant = jaugeCarburant;
    }
}
```



```

/* Constructeur avec une signature différente du premier constructeur car moins de
   Paramètres ⇒ Permet de créer des véhicules avec le reservoir vide */
public Vehicule(String plaque, String modele, int anneeFabrication) {
    ...                // Constructeur qui met la jauge à carburant à 0
}

/* Constructeur sans argument : permet de créer des véhicules sans initialiser ses
   variables d'instance */
public Vehicule() {
}
...
}

```

Afin de n'écrire qu'une seule fois les mêmes lignes de code (point de modification unique), il est possible dans un constructeur de faire appel à un autre constructeur de la même classe. La syntaxe est : **this(...)**.

```


public class Vehicule {
    private String plaque;
    private String modele;
    private int anneeFabrication;
    private int jaugeCarburant;

    public Vehicule(String plaque, String modele, int anneeFabrication,
                    int jaugeCarburant) {
        this.plaque = plaque;
        this.modele = modele;
        this.anneeFabrication = anneeFabrication ;
        this.jaugeCarburant = jaugeCarburant;
    }

    public Vehicule(String plaque, String modele, int anneeFabrication) {
        // Signature différente du premier constructeur car moins de paramètres
        this(plaque, modele, anneeFabrication, 0);
    }

    /* this(...) = référence vers un constructeur qui existe déjà.
       ⇒ Provoque l'appel en cascade au premier constructeur */
    ...
}

```



Il est donc possible dans un constructeur de faire appel à un autre constructeur via le mot réservé **this** suivi de **parenthèses**. À ne pas confondre avec

this.nomMethode (...), qui appelle la méthode *nomMethode (...)* sur l'objet courant.

Dans l'exemple ci-dessus, le second constructeur peut être utilisé pour créer des véhicules dont le réservoir de carburant est vide.

Le constructeur sans argument peut être utilisé pour créer des objets dont les variables d'instance seront initialisées avec les valeurs par défaut. Pour rappel, les nombres seront initialisés à 0, les booléens à false et les objets à (une référence) null. Le programmeur doit alors initialiser explicitement les variables d'instance de l'objet créé en appelant les différents setters.

Exemples

```
public class Principal {  
    public static void main(String [] args) {  
        Vehicule clio = new Vehicule("1-AZC-468","Clio",2016,30);  
        System.out.println("Le réservoir de la clio contient : " +  
                           clio.getJaugeCarburant() + " litres");  
    }  
}
```

⇒ Affiche à l'écran :

*Le réservoir de la clio contient : **30** litres*

```
Vehicule fiesta = new Vehicule("1-DEF-133","Fiesta",2017);  
System.out.println("Le réservoir de la fiesta contient : " +  
                   fiesta.getJaugeCarburant() + " litres");
```

⇒ Affiche à l'écran :

*Le réservoir de la fiesta contient : **0** litres*


```
Vehicule duster = new Vehicule();  
duster.setPlaque("1-DUS-288");  
duster.setModele("Duster");  
duster.setAnneeFabrication(2018);  
duster.setJaugeCarburant(50);  
System.out.println("Le réservoir du duster contient : " +  
                   duster.getJaugeCarburant() + " litres");
```

⇒ Affiche à l'écran :

*Le réservoir du duster contient : **50** litres*

Exemple 2

Utilisons des objets de type étudiant (à ne pas confondre avec la notion d'étudiant inscrit dans une section informatique : cf. classe *EtudiantInformatique*). Un étudiant ne contient que trois caractéristiques, à savoir, son nom (+ prénom), ainsi que la section et le bloc dans lesquels il est inscrit.



```
public class Etudiant {
    private String prenomNom;
    private String section;
    private int bloc;

    // Constructeur pour créer n'importe quel étudiant
    public Etudiant(String prenomNom, String section, int bloc) {
        this.prenomNom = prenomNom;
        this.section = section;
        this.bloc = bloc;
    }

    // Constructeur pour créer des étudiants en Informatique de gestion
    public Etudiant(String prenomNom, int bloc) {
        this(prenomNom, "Informatique de gestion", bloc);
    }

    // Constructeur pour créer des étudiants inscrits au bloc 1
    public Etudiant(String prenomNom, String section) {
        this(prenomNom, section, 1);
    }

    // Constructeur pour créer des étudiants inscrits en Informatique de gestion au bloc 1
    public Etudiant(String prenomNom) {
        this(prenomNom, 1);
        // ou this(prenomNom, "Informatique de gestion ");
    }

    public String toString() {
        return prenomNom + " est inscrit(e) au bloc " + bloc
            + " de la section " + section;
    }
}
```

Quatre constructeurs sont prévus dans cette classe. Le premier avec autant d'arguments qu'il y a de variables d'instance.

Le deuxième constructeur facilite la tâche du programmeur qui souhaite créer des étudiants inscrits en Informatique de gestion (deux arguments : le nom-prénom et le bloc).

Le troisième constructeur facilite la création d'étudiants inscrits en première année (deux arguments : le nom-prénom et la section).

Le dernier constructeur facilite la création d'étudiants inscrits en première année dans la section Informatique de gestion (un seul argument : le nom-prénom).

Le deuxième et le troisième constructeur ont, tous deux, deux arguments, mais ils diffèrent par leur type.

```
public class Principal {  
    public static void main(String [] args) {  
        Etudiant jules = new Etudiant("Jules Money","Comptabilité",3);  
        System.out.println(jules);  
    }  
}
```

⇒ Affiche à l'écran :

Jules Money est inscrit(e) au bloc 3 de la section Comptabilité

```
Etudiant alan = new Etudiant("Alan Turing",2);  
System.out.println(alan);
```

⇒ Affiche à l'écran :

Alan Turing est inscrit(e) au bloc 2 de la section Informatique de gestion

```
Etudiant juste = new Etudiant("Juste Leblanc","Droit");  
System.out.println(juste);
```

⇒ Affiche à l'écran :

Juste Leblanc est inscrit(e) au bloc 1 de la section Droit

```
Etudiant jenny = new Etudiant("Jenny Motion");  
System.out.println(jenny);
```

⇒ Affiche à l'écran :

Jenny Motion est inscrit(e) au bloc 1 de la section Informatique de gestion

À chaque appel de constructeur, le compilateur Java analyse la signature de celui-ci pour déterminer quel constructeur utiliser.

5.2 Surcharge des méthodes

On peut également surcharger des méthodes. Il suffit pour ce faire de donner le même nom à deux méthodes, à condition que leurs signatures diffèrent (types d'arguments et/ou nombre d'arguments).

Attention, le type du résultat n'intervient pas pour différencier deux méthodes de même signature. En effet, lors de l'appel de la méthode, Java ne peut identifier laquelle des deux méthodes choisir, si elles ont toutes deux la même signature mais des types de retour différents.

Exemple **incorrect** de surcharge de méthodes :

```
String methodeA(int x, float y, String z) {  
    ...  
}  
int methodeA(int a, float b, String c) {  
    ...  
}
```

Exemples **corrects**

Exemple 1 :

Imaginons une méthode permettant d'ajouter un certain nombre de litres de carburant dans le réservoir d'un véhicule et une autre méthode permettant de faire le plein, toutes les deux portant le même nom : *ajouterCarburant*.

Prévoyons pour ce faire une variable d'instance supplémentaire : *capaciteReservoir*.

Vehicule
plaque modele anneeFabrication capaciteReservoir jaugeCarburant
ajouterCarburant(int) ajouterCarburant() typeVehicule() toString()

```

public class Vehicule {
    private String plaque;
    private String modele;
    private int anneeFabrication;
    private int capaciteReservoir;
    private int jaugeCarburant;

    public Vehicule(String plaque, String modele, int anneeFabrication,
                    int capaciteReservoir, int jaugeCarburant) {
        this.plaque = plaque;
        this.modele = modele;
        this.anneeFabrication = anneeFabrication ;
        this.capaciteReservoir = capaciteReservoir;
        this.jaugeCarburant = jaugeCarburant;
    }
    public Vehicule(String plaque, String modele,
                    int anneeFabrication, int capaciteReservoir) {
        this(plaque, modele, anneeFabrication , capaciteReservoir, 0);
    }
    public Vehicule() {
    }

    public void ajouterCarburant(int nbLitresAjoutes) {
        jaugeCarburant += nbLitresAjoutes;
        if (jaugeCarburant > capaciteReservoir)
            jaugeCarburant = capaciteReservoir;
    }
    public void ajouterCarburant() {
        jaugeCarburant = capaciteReservoir;
    }

    public String typeVehicule() {
        return modele + " " + anneeFabrication;
    }
}

```

Surcharge de constructeurs

Surcharge de méthodes

La méthode *void **ajouterCarburant(int nbLitresAjoutes)*** permet d'ajouter *nbLitresAjoutes* de carburant dans le réservoir, tandis que la méthode *void **ajouterCarburant()*** permet d'enregistrer que le plein de carburant a été fait. Autrement dit, la méthode *ajouterCarburant ()* permet de mettre à jour la variable d'instance *jaugeCarburant* pour représenter l'état de la jauge à

carburant après passage à la pompe pour faire le plein. La jauge étant remplie, la variable d'instance *jaugeCarburant* contient désormais le nombre de litres de carburant égal à la valeur contenue dans la variable d'instance *capaciteReservoir*.

```
public class Principal {
    public static void main(String [] args) {
        Vehicule clio = new Vehicule("1-AZC-468","Clio",2018,50);
        // Création d'un véhicule avec capaciteReservoir = 50 et jaugeCarburant = 0
        clio.ajouterCarburant(30);    // Ajoute 30 à la variable jaugeCarburant
        System.out.println("Réserve de " + clio.getJaugeCarburant()
            + " litres");
    }
}
```

⇒ Affiche à l'écran : Réserve de **30** litres

```
        clio.ajouterCarburant();    // Faire le plein
        System.out.println("Réserve de " + clio.getJaugeCarburant()
            + " litres");
```

⇒ Affiche à l'écran : Réserve de **50** litres

Exemple 2

EtudiantInformatique
prenom nom genre section bloc totalEctsLangue nbEctsReussisLangue totalEctsMath nbEctsReussisMath totalEctsProgra nbEctsReussisProgra
nomUtilisateur() nbTotalEctsDansPAE() nbTotalEctsReussis() reussiteTotalePAE() reussitePartiellePAE() reussitePartiellePAE(int) tauxReussiteCoursGeneraux() tauxReussiteCoursSpecialite()

Les méthodes *reussitePartiellePAE* permettent soit de déterminer la réussite partielle à 45 Ects au bloc 1, soit de déterminer la réussite partielle à partir d'un seuil (fictif ?) donné en argument (exemple : 50 Ects).

```
public class EtudiantInformatique {
    private String prenom;
    private String nom;
    private char genre;
    private String section;
    private int bloc;
    private int totalEctsLangue;
    private int nbEctsReussisLangue;
    private int totalEctsMath;
    private int nbEctsReussisMath;
    private int totalEctsProgra;
    private int nbEctsReussisProgra;
    ...    // Constructeurs
    public int nbTotalEctsReussis() {
        ...
    }
    public boolean reussitePartiellePAE() {
        return !this.reussiteTotalePAE() && bloc == 1
               && this.nbTotalEctsReussis() >= 45;
    }
    public boolean reussitePartiellePAE(int seuil) {
        return this.nbTotalEctsReussis() >= seuil ;
    }
    ...
}
```

**Surcharge
de méthodes**

```
public class Principal {
    public static void main (String [] args) {
        EtudiantInformatique alan;
        alan = new EtudiantInformatique("Alan","Turing",'M',
                                         "Informatique de gestion",1,11,11,9,9,40,25);
        System.out.println(alan);
    }
}
```

⇒ Affiche à l'écran :

L'étudiant Alan Turing inscrit au bloc 1 de la section Informatique de gestion


```
System.out.println("Réussite partielle : " +
    alan.reussitePartiellePAE());
```

⇒ Affiche à l'écran :

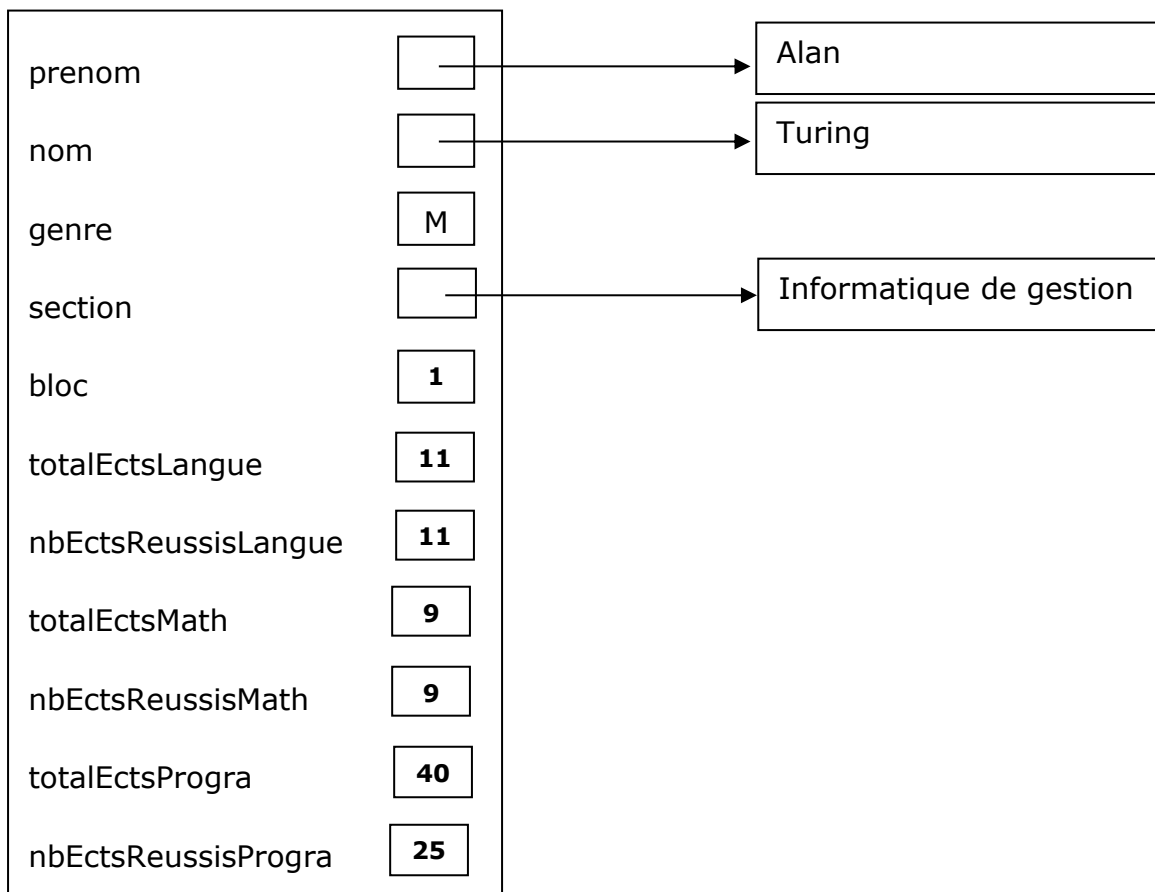
Réussite partielle : true

```
System.out.println("Réussite partielle : " +
    alan.reussitePartiellePAE(50));
```

⇒ Affiche à l'écran :

Réussite partielle : false

En mémoire, l'objet *alan* aura donc la structure suivante :



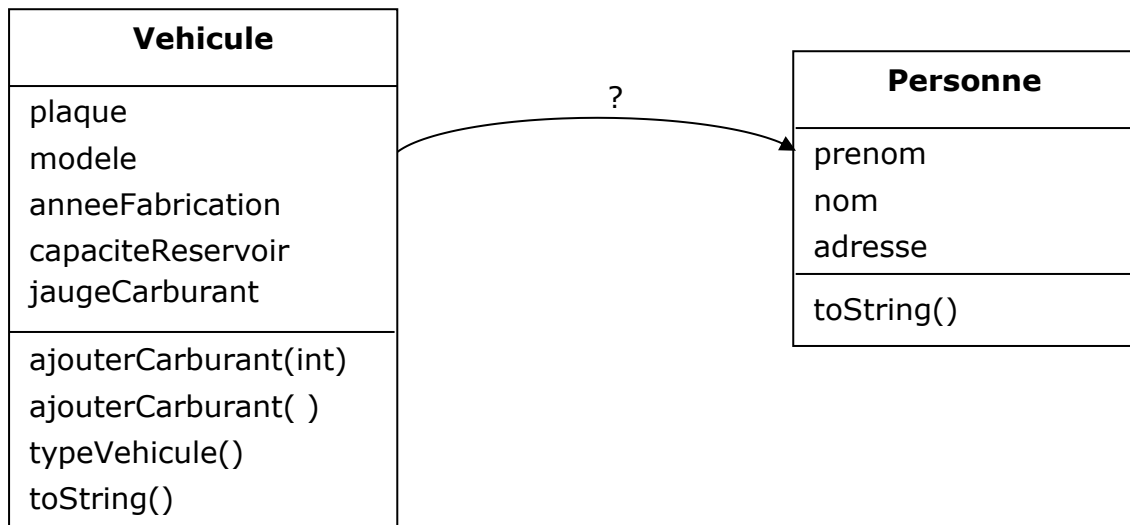
6 Liens entre objets

6.1 Lien entre deux objets (entre deux classes)

Exemple

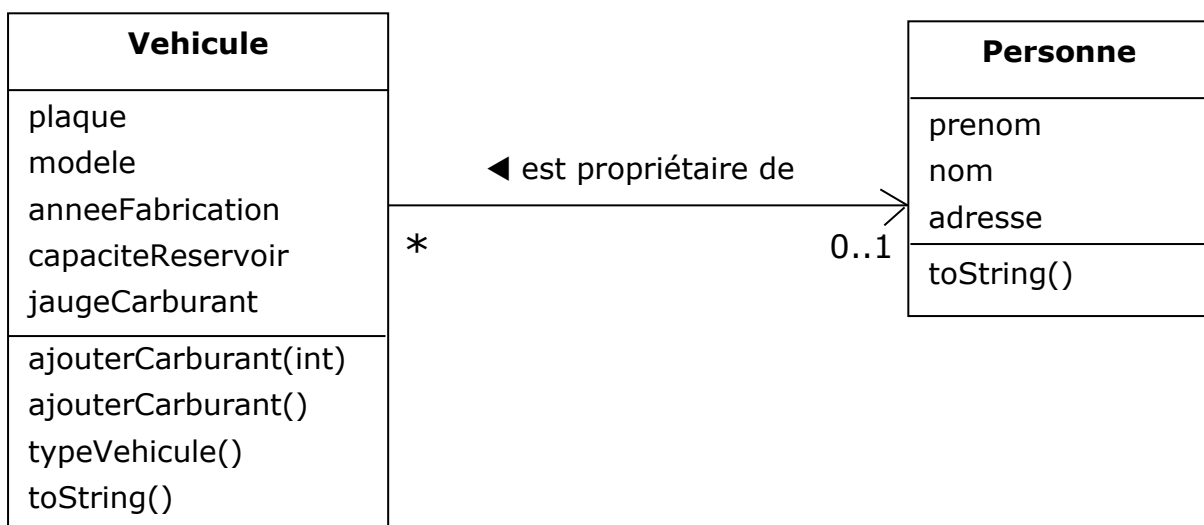
Partons de l'hypothèse que les classes *Vehicule* et *Personne* existent.

On désirerait associer un propriétaire à tout véhicule. Le propriétaire d'un véhicule doit être un objet de type *Personne*.



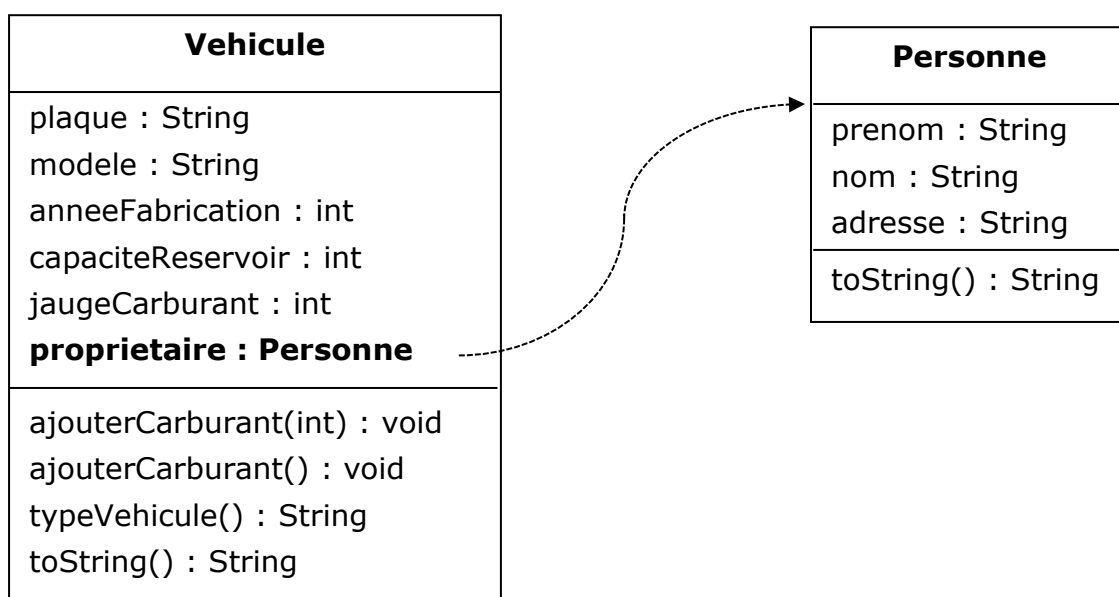
On établit une **association** entre les deux classes.

Diagramme de classes UML :



Selon ce diagramme de classes UML, un véhicule peut avoir un et un seul propriétaire et une personne peut être propriétaire de plusieurs véhicules. Un véhicule pourrait ne pas avoir de propriétaire. Seul un véhicule "voit" son propriétaire, s'il existe, mais un propriétaire ne "voit pas" ses véhicules (cf. la **flèche** du côté de la classe *Personne*). Autrement dit, à partir d'un objet de type *Vehicule*, on a accès, on peut retrouver l'objet de type *Personne* qui y serait associé, mais à partir d'un objet de type *Personne*, on n'a pas accès aux objets de type *Vehicule* associés.

Le lien entre un objet de type *Vehicule* et l'objet de type *Personne* qui en est le propriétaire se fait **via une variable d'instance de type *Personne* déclarée dans la classe *Vehicule***.



```

public class Personne {
    private String prenom;
    private String nom;
    private String adresse;

    // Constructeur
    public Personne(String prenom, String nom, String adresse {
        this.prenom = prenom;
        this.nom = nom;
        this.adresse = adresse;
    }

    ...          // Public getters : getPrenom(), getNom(), getAdresse()
    ...          // Public setters : setPrenom(...), setNom(...), setAdresse(...)
  
```

```

    public String toString() {
        return prenom + " " + nom + " domicilié au " + adresse;
    }
}

```

```

public class Vehicule {
    private String plaque;
    private String modele;
    private int anneeFabrication ;
    private int capaciteReservoir;
    private int jaugeCarburant;
    private Personne proprietaire;           // Variable d'instance de type Personne

    public Vehicule(String plaque, String modele, int anneeFabrication,
                    int capaciteReservoir, int jaugeCarburant,
                    Personne proprietaire) {
        this.plaque = plaque;
        this.modele = modele;
        this.anneeFabrication = anneeFabrication;
        this.capaciteReservoir= capaciteReservoir;
        this.jaugeCarburant = jaugeCarburant;
        this.proprietaire = proprietaire;
    }

    ...      /* Public getters : getPlaque(), getModele(), getAnneeFabrication(),
              getCapaciteReservoir(), getJaugeCarburant() */

    public Personne getProprietaire() {
        return proprietaire;
    }

    ...      /* Public setters : setPlaque(...), setModele(...), setAnneeFabrication(...),
              setCapaciteReservoir(...), setJaugeCarburant(...) */

    public void setProprietaire(Personne proprietaire) {
        this.proprietaire = proprietaire;
    }

    ...      // Autres méthodes
}

```

La nouvelle variable d'instance étant de type *Personne*, il faut bien entendu prévoir un argument de type *Personne* dans le constructeur.

De même, comme la variable d'instance *proprietaire* est déclarée *private*, il faut prévoir un getter public (dont le type de retour est *Personne*) ainsi qu'un setter public (dont le type d'argument est également *Personne*).

```
public class Principal {  
    public static void main(String [] args) {  
        Personne paul = new Personne("Paul","Petit","1, rue Ely, Namur");  
        Vehicule clio = new Vehicule("1-AZC-468","Clio",2018,50,20,paul);  
        System.out.println(clio.getPlaque());  
    }  
}
```

⇒ Affiche à l'écran : 1-AZC-468

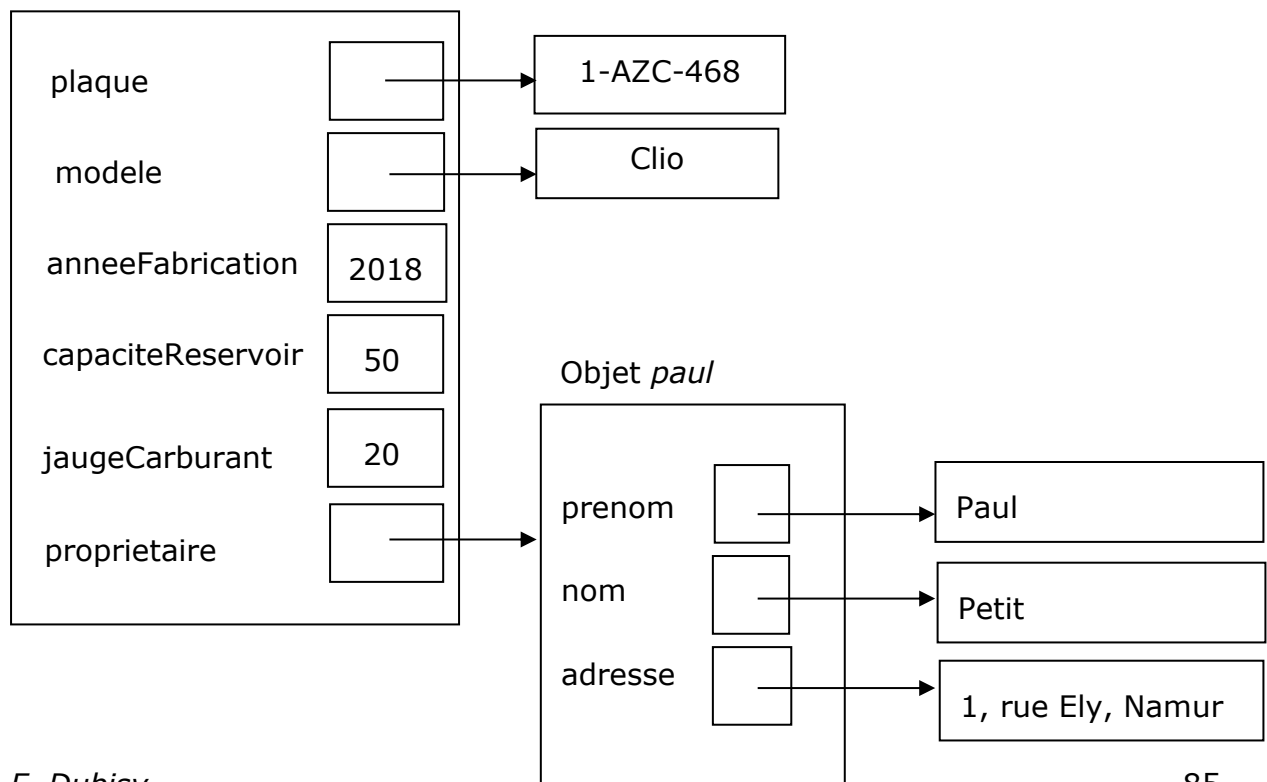
```
System.out.println("L'adresse du propriétaire de la Clio : "  
    + clio.getProprietaire().getAdresse());
```

1 objet de type Personne

⇒ Affiche à l'écran :

L'adresse du propriétaire de la Clio : 1, rue Ely, Namur

En mémoire, l'objet *clio* a la structure suivante :



En effet, la variable d'instance *proprietaire* du véhicule *clio* est une référence vers un autre objet (de type *Personne*). Ce dernier contient lui-même des variables d'instance qui sont des références vers des objets (de type *String*) se trouvant ailleurs en mémoire.

⇒ *clio.getProprietaire()* fait référence à l'objet de type *Personne* référencé par la variable d'instance *proprietaire* de l'objet *clio*.

Autrement dit, *clio.getProprietaire()* référence l'objet *paul*.

⇒ *clio.getProprietaire.getAdresse()* fait référence à l'objet de type *String* référencé par la variable d'instance *adresse* de l'objet *paul*.

6.2 Appel implicite à la méthode *toString* d'un objet référencé par une variable d'instance

```
public class Vehicule {
    private String plaque;
    private String modele;
    private int anneeFabrication ;
    private int capaciteReservoir;
    private int jaugeCarburant;
    private Personne proprietaire;          // Variable d'instance de type Personne
    ...    // Constructeurs, public getters/setters, autres méthodes

    public String typeVehicule() {
        return modele + " " + anneeFabrication;
    }

    public String toString() {
        return "La " + this.typeVehicule() + " immatriculée " + plaque +
            "\nappartient à " + proprietaire;
    }
    /* Proprietaire est un objet de type Personne,
    ⇒ appel au toString() de la classe Personne */
}
```

La méthode *toString* de la classe *Vehicule* fait appel à la variable d'instance *proprietaire* qui est un objet. Pour rappel, chaque fois qu'une référence à un objet apparaît dans une instruction "là où on attendrait" une chaîne de caractères, Java fait appel implicitement à la méthode *toString* sur cet objet. Par conséquent, la méthode *toString* de la classe *Vehicule* appelle implicitement la méthode *toString* sur l'objet *proprietaire*.

```
public class Principal {
    public static void main(String [] args) {
        Personne paul = new Personne("Paul","Petit","1, rue Ely, Namur");
        Vehicule clio = new Vehicule("1-AZC-468","Clio",2018,50,20, paul);
        // 1 objet de type Personne
        System.out.println(clio.getProprietaire());
        // Appel au toString sur la variable d'instance proprietaire de l'objet clio
    }
}
```

⇒ Affiche à l'écran :

Paul Petit domicilié au 1, rue Ely, Namur

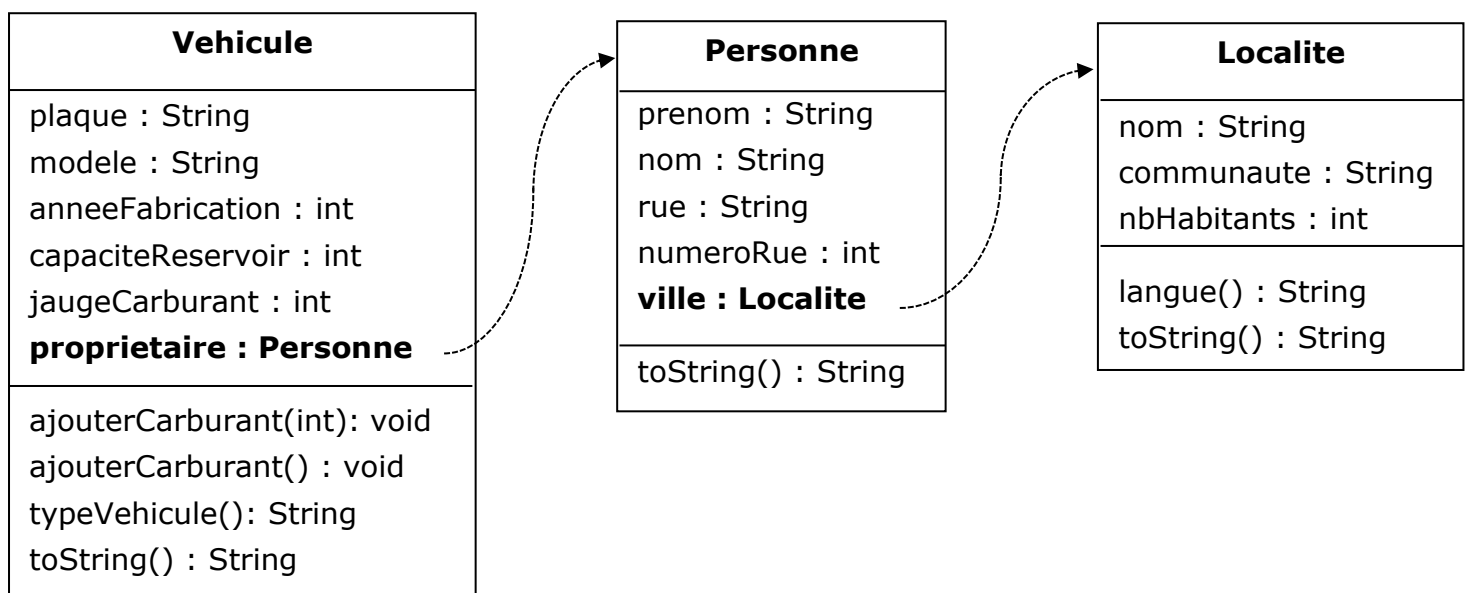
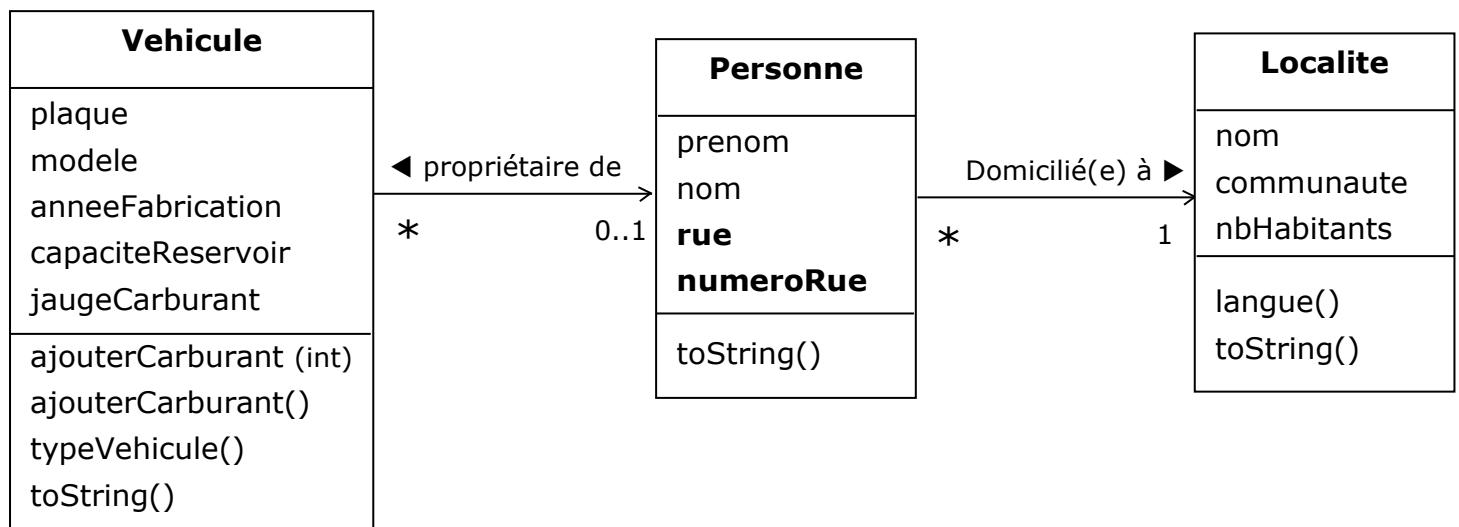
```
|| System.out.println(clio);
```

⇒ Affiche à l'écran :

*La Clio 2018 immatriculée 1-AZC-468
appartient à Paul Petit domicilié au 1, rue Ely, Namur*

6.3 Plus de deux classes reliées

Exemple 1



```

public class Localite {
    private String nom;
    private String commune;
    private int nbHabitants;
    public Localite(String nom, String commune, int nombreHabitants) {
        this.nom = nom;
        this.commune = commune;
        nbHabitants = nombreHabitants;
    }
}
  
```

```

    public String langue() {
        if (communaute.equals("flamande")) {
            return "néerlandais";
        }
        else {
            if(communaute.equals("française")) {
                return "français";
            } else {
                return "allemand";
            }
        }
    }
    public String toString() {
        return nom + " (" + nbHabitants + " habitants)";
    }
}

```

Notons que la méthode *langue* ne gère pas les cas d'erreurs.

Pour comparer le contenu de deux objets de type *String*, il faut utiliser la méthode ***equals*** de la classe *String* : **objet1.equals (objet2)**.

Cette méthode est appelée ici sur un objet de type *String*. Elle prend un argument qui est la référence vers le second objet à comparer. La méthode *equals* retourne un booléen (*true* si les deux valeurs sont égales, *false* sinon).

Exemple

```

String nom1 = " Jules ";
String nom2 = " Julos " ;
if (nom1.equals(nom2)) ...

```

N.B. Il serait illogique d'écrire : `if (nom1 == nom2)` : on comparerait alors des références vers des objets, autrement dit, des adresses en mémoire centrale.

Pour comparer le contenu d'un objet de type *String* à une valeur, on peut utiliser la méthode *equals(...)* comme suit : **objet.equals(valeur)**.

L'argument *valeur* est la valeur à laquelle il faut comparer la chaîne de caractères référencée par l'objet *objet*.

Exemple

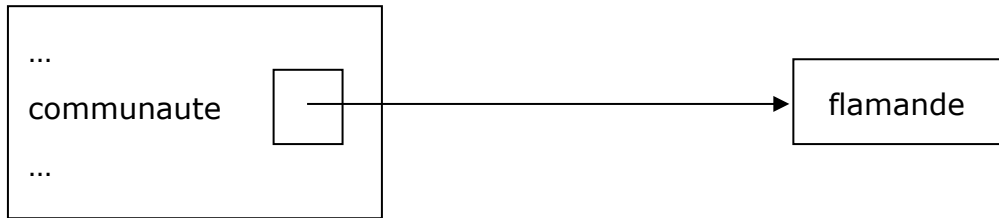
```

communaute.equals("flamande")

```

⇒ Compare la chaîne de caractères référencée par la variable d'instance *communaute* à la valeur *flamande*.

En mémoire



```
public class Personne {
    private String prenom;
    private String nom;
    private String rue;
    private int numeroRue;
    private Localite ville;

    // Constructeur
    public Personne(String prenom, String nom, String rue, int numeroRue,
                    Localite ville) {
        this.prenom = prenom;
        this.nom = nom;
        this.rue = rue;
        this.numeroRue = numeroRue;
        this.ville = ville;
    }
    ...    // Public getters : getPrenom(), getNom(), getRue(), getNumeroRue()

    public Localite getVille() {
        return ville ;
    }
    ...    // Public setters : setPrenom(...), setNom(...), setRue(...), setNumeroRue(...)

    public void setVille(Localite ville) {
        this.ville = ville;
    }
}
```

```

    public String toString() {
        return prenom + " " + nom + " domicilié au " + numeroRue
            + ", rue " + rue + " à " + ville;
    }
}

```

Appel au *toString* de *Localite*

```

public class Vehicule {
    private String plaque;
    private String modele;
    private int anneeFabrication ;
    private int capaciteReservoir;
    private int jaugeCarburant;
    private Personne proprietaire;

    public Vehicule(String plaque, String modele, int anneeFabrication ,
        int capaciteReservoir, int jaugeCarburant,
        Personne proprietaire) {
        this.plaque = plaque;
        this.modele = modele;
        this.anneeFabrication = anneeFabrication ;
        this.capaciteReservoir= capaciteReservoir;
        this.jaugeCarburant = jaugeCarburant;
        this.proprietaire = proprietaire;
    }

    ... /* Public getters : getPlaque(), getModele(), getAnneeFabrication(),
        getCapaciteReservoir(), getJaugeCarburant() */

    public Personne getProprietaire() {
        return proprietaire ;
    }

    ... /* Public setters : setPlaque(...), setModele(...), setAnneeFabrication(...)
        setCapaciteReservoir(...), setJaugeCarburant(...) */

    public void setProprietaire(Personne proprietaire) {
        this.proprietaire = proprietaire;
    }

    ... // Autres méthodes

    public String toString() {
        return "La " + this.typeVehicule() + " immatriculée " + plaque +
            "\nappartient à " + proprietaire;
    }
}

```

Appel au *toString* de *Personne*

```

public class Principal {
    public static void main(String [] args) {
        Localite namur = new Localite("Namur","française",15000);
        Personne paul = new Personne("Paul","Petit","Ely", 1, namur);
        Vehicule clio = new Vehicule("1-AZC-468","Clio",2018,50,20, paul);
        System.out.println(clio);
    }
}

```

⇒ Affiche à l'écran :

La Clio 2018 immatriculée 1-AZC-468

appartient à Paul Petit domicilié au 1, rue Ely à Namur (15000 habitants)

1 objet de type *Personne* ⇒ *toString* de *Personne*

```

System.out.println(clio.getProprietaire());

```

⇒ Affiche à l'écran :

Paul Petit domicilié au 1, rue Ely à Namur (15000 habitants)

1 objet de type *Localite* ⇒ *toString* de *Localite*

```

System.out.println(clio.getProprietaire().getVille());

```

⇒ Affiche à l'écran :

Namur (15000 habitants)

```

System.out.println(clio.getProprietaire().getVille().getNom());

```

⇒ Affiche à l'écran :

Namur

```

System.out.println(clio.getProprietaire().getVille().langue());

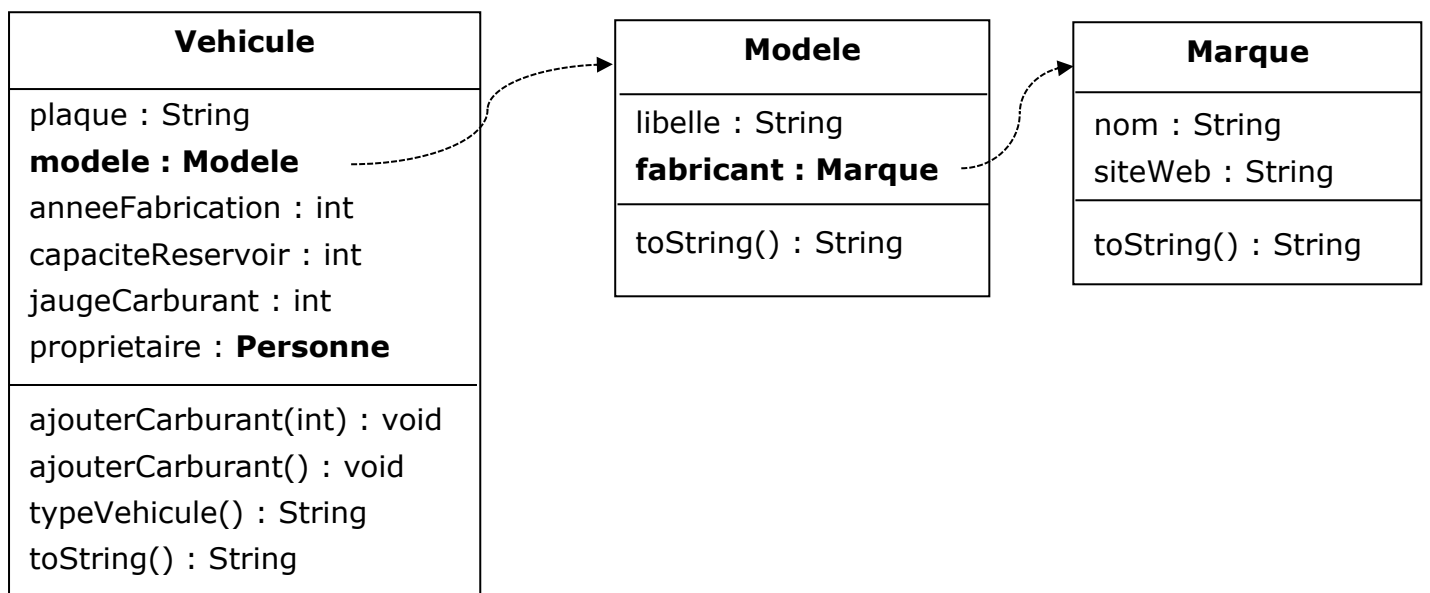
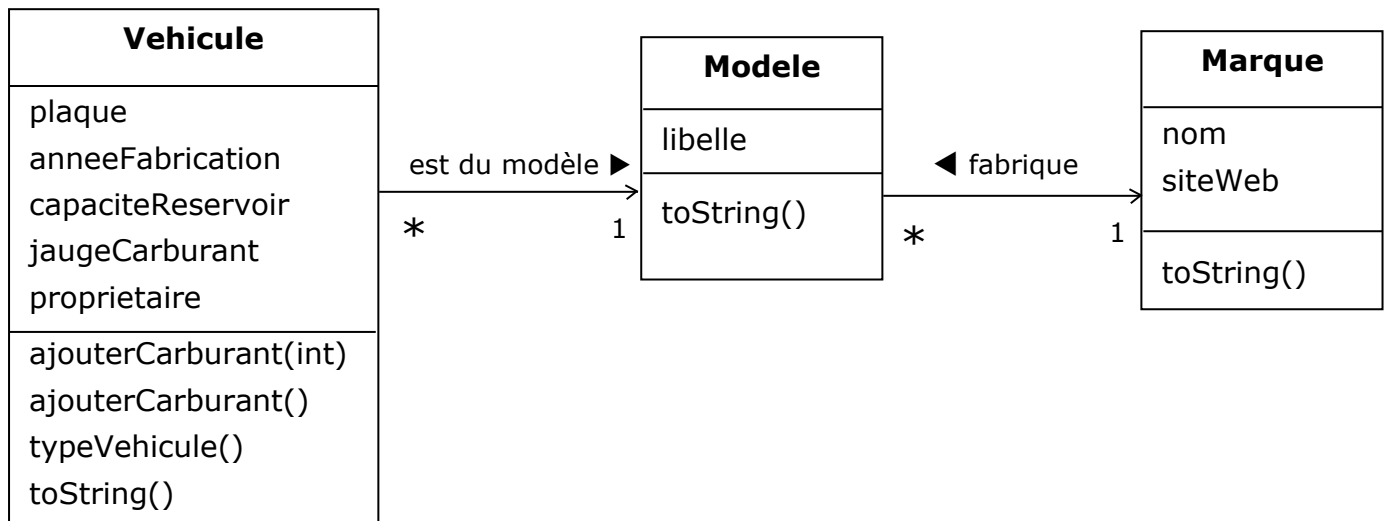
```

⇒ Affiche à l'écran :

français

Exemple 2

F. Dubisy



```

public class Marque {
    private String nom;
    private String siteWeb;

    public Marque(String nom, String siteWeb) {
        this.nom = nom;
        this.siteWeb = siteWeb;
    }

    public String toString() {
        return nom;
    }

    ... // Getters et setters
}

```

```

public class Modele {
    private String libelle;
    private Marque fabricant;

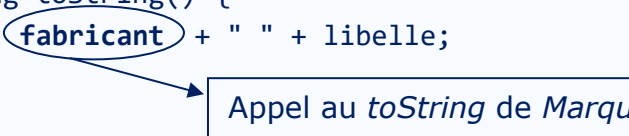
    public Modele(String libelle, Marque fabricant) {
        this.libelle = libelle;
        this.fabricant = fabricant;
    }
    ... // public getLibelle()

    public Marque getFabricant() {
        return fabricant;
    }
    ... // public setLibelle(...)

    public void setFabricant(Marque fabricant) {
        this.fabricant = fabricant;
    }

    public String toString() {
        return fabricant + " " + libelle;
    }
}

```



Appel au *toString* de *Marque*

```

public class Vehicule {
    private String plaque ;
    private Modele modele; // Variable d'instance de type Modele
    private int anneeFabrication ;
    private int capaciteReservoir ;
    private int jaugeCarburant;
    private Personne proprietaire; // Variable d'instance de type Personne

    public Vehicule(String plaque, Modele modele, int anneeFabrication ,
        int capaciteReservoir, int jaugeCarburant,
        Personne proprietaire) {
        this.plaque = plaque;
        this.modele = modele;
        this.anneeFabrication = anneeFabrication ;
        this.capaciteReservoir= capaciteReservoir;
        this.jaugeCarburant = jaugeCarburant;
        this.proprietaire = proprietaire;
    }
}

```

```

public Vehicule(String plaque, Modele modele, int annee,
                int capaciteMax, Personne proprietaire) {
    this(plaque, modele, annee, capaciteMax, 0, proprietaire);
}

... /* Public getters : getPlaque(), getAnneeFabrication(),
    getCapaciteReservoir(), getJaugeCarburant(), getProprietaire() */

public Modele getModele() {
    return modele ;
}

... /* Public setters : setPlaque(...), setAnneeFabrication(...)
    setCapaciteReservoir(...), setJaugeCarburant(...), setProprietaire(...) */

public void setModele(Modele modele) {
    this.modele = modele;
}

... // Autres méthodes

public String toString() {
    return "La " + modele + " immatriculée " + plaque
        + "\nappartient à " + proprietaire ;
}
}

```

Appel au *toString* de *Modele* et de *Personne*

```

public class Principal {
    public static void main(String [] args) {
        Localite namur = new Localite("Namur","française",15000);
        Personne paul = new Personne("Paul","Petit","Ely", 1, Namur);
        Marque renault = new Marque("Renault","https://fr.renault.be");
        Modele clio = new Modele("Clio", renault);
        Vehicule clioDePaul = new Vehicule("1-AZC-468",
                                           clio, 2018, 50, 20,
                                           paul);
        System.out.println(clioDePaul);
    }
}

```

⇒ Affiche à l'écran :

La Renault Clio immatriculée 1-AZC-468

appartient à Paul Petit domicilié au 1, rue Ely à Namur (15000 habitants)

1 objet de type *Modele* ⇒ *toString* de *Modele*

```
|| System.out.println(clioDePaul.getModele());
```

⇒ Affiche à l'écran :

Renault Clio

1 objet de type *Marque* ⇒ *toString* de *Marque*

```
|| System.out.println(clioDePaul.getModele().getFabricant());
```

⇒ Affiche à l'écran :

Renault

1 objet de type *Marque*

```
|| System.out.println(clioDePaul.getModele().getFabricant().getSiteWeb());
```

⇒ Affiche à l'écran :

https://fr.renault.be

7 Héritage

Afin de réduire la complexité d'un programme, il est intéressant d'appliquer le principe de généralisation, en remplaçant plusieurs entités qui partagent des fonctions similaires par une construction unique. Autrement dit, il est intéressant de créer une classe qui encapsule les fonctionnalités partagées par plusieurs classes et de construire de nouvelles classes à partir de cette classe mère en ajoutant des fonctionnalités spécifiques.

C'est ainsi que l'on peut construire des hiérarchies de classes héritant les unes des autres.

De plus, un des objectifs de la programmation est de diminuer le temps de programmation en réutilisant le plus possible des composants existants. Un des avantages de la programmation O.O., c'est que les composants réutilisés (classes) peuvent être adaptés s'il y a lieu.

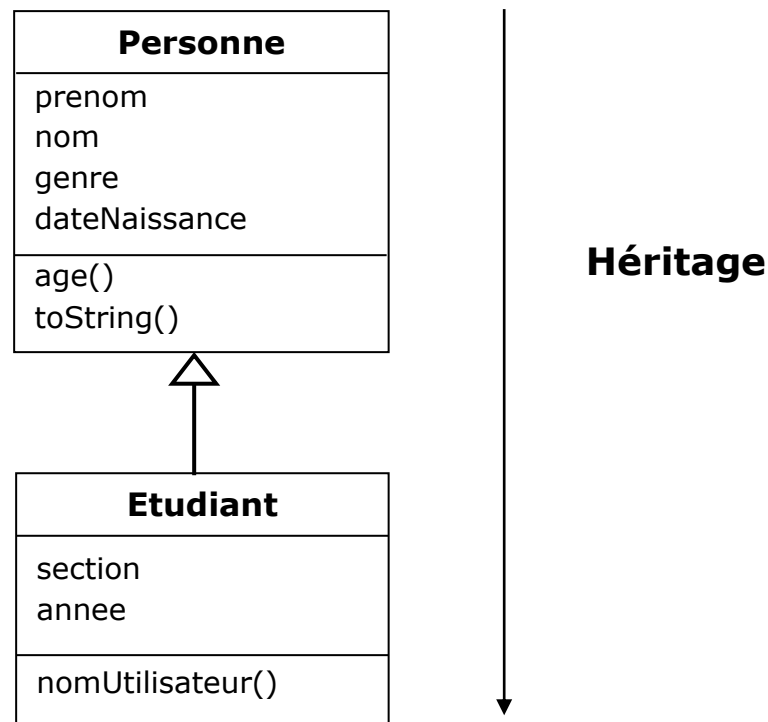
7.1 Sous-classe (déclaration et constructeur)

Partons des hypothèses suivantes :

- La classe *Personne* est disponible (créée par un autre programmeur ou par nous-mêmes pour une autre application)
- Nous devons gérer des étudiants.

La classe *Personne* permet de manipuler des notions comme le nom, le prénom, le genre (*M*, *m* ou *F*, *f*), la date de naissance et l'âge d'une personne. Ces notions restent d'actualité pour un étudiant, mais un étudiant présente des caractéristiques supplémentaires qui lui sont propres, comme sa section (*Informatique de gestion*, *Comptabilité*, *Droit...*), son bloc (1,2,3) et le nom d'utilisateur (login name) qu'il reçoit (par exemple, *TuringAlanIn2*). Pour pouvoir récupérer les caractéristiques de la classe *Personne* tout en en ajoutant d'autres, il suffit de créer une nouvelle classe *Etudiant* en précisant qu'elle hérite des caractéristiques de la classe *Personne* et y placer les caractéristiques supplémentaires spécifiques aux étudiants. On parle alors de super-classe ou classe parent : la classe *Personne*, et de sous-classe ou classe enfant : la classe *Etudiant*.

Diagramme de classes UML avec héritage



Le lien **d'héritage** entre la classe *Etudiant* et la classe *Personne* signifie que toute occurrence de la classe *Etudiant* hérite des caractéristiques de la classe *Personne* (variables d'instance et méthodes) en plus des caractéristiques qui lui sont propres. Autrement dit, tout objet de type *Etudiant* contiendra les variables d'instance héritées de la classe *Personne*, à savoir, *prenom*, *nom*, *genre* et *dateNaissance* en plus des variables d'instance propres à la classe *Etudiant*, à savoir, *section* et *bloc*.

Nous allons utiliser la classe *LocalDate* pour gérer les dates. Cette classe nécessite un import : `import java.time.LocalDate`.

Dans la classe *LocalDate*, il existe une méthode pour créer un objet de type date à partir d'une année, un mois et un jour dans le mois. Cette méthode est la méthode *of*. Cette méthode est particulière car elle est déclarée *static* ; il s'agit d'une méthode dite méthode de classe. Les méthodes de classe seront vues en détail dans le chapitre 8. En deux mots, une méthode de classe ne s'appelle pas sur un objet mais sur une classe.

Par exemple, pour générer un objet de type *LocalDate* correspondant au 15 janvier 2000, il faut écrire : `LocalDate.of(2000,1,15)`.

Dans le constructeur de la classe *Personne*, nous pourrions nous contenter de recevoir comme argument un objet de type *LocalDate*. Pour la facilité d'utilisation de la classe *Personne* par d'autres programmeurs, il semble intéressant de proposer un constructeur dans la classe *Personne* qui demande l'année, le mois et le jour de naissance. Le constructeur se chargera de créer un objet de type *LocalDate* à partir de ces trois arguments et de l'affecter à la variable d'instance *dateNaissance*.

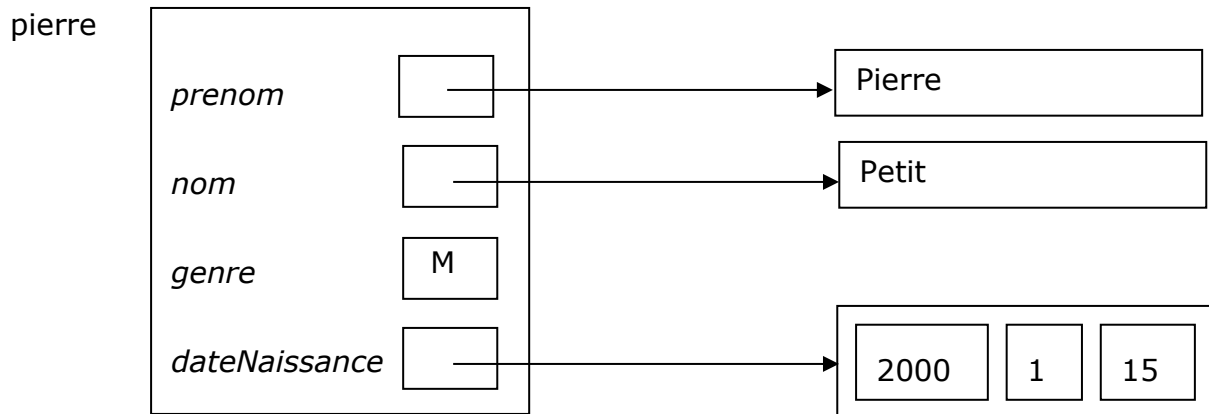
```
import java.time.LocalDate;
public class Personne {
    private String prenom ;
    private String nom;
    private char genre;
    private LocalDate dateNaissance;

    public Personne(String prenom, String nom, char genre,
        int jourNaissance, int moisNaissance, int anneeNaissance) {
        this.prenom = prenom;
        this.nom = nom;
        setGenre(genre);
        this.dateNaissance = LocalDate.of(anneeNaissance, moisNaissance,
            jourNaissance);
    }
    ...    // Public getters et setters
    public int age() {
        ...
    }

    public String toString() {
        return "La personne " + prenom + " " + nom;
    }
}
```

```
public class Principal {
    public static void main(String [] args) {
        Personne pierre = new Personne("Pierre","Petit",'M',15,1,2000);
    }
}
```

En mémoire :



La variable d'instance *dateNaissance* est une référence vers un objet (de type *LocalDate*) se trouvant ailleurs en mémoire. Il en va de même pour les variables de type String *prenom* et *nom* ; elles contiennent des références vers des objets ailleurs en mémoire.

Pour déclarer qu'une classe est une sous-classe d'une autre, on ajoute la clause **extends** suivi du nom de la super-classe. La déclaration "*class Etudiant extends Personne*" signifie que la classe *Etudiant* est une sous-classe de la classe *Personne*.

```
public class Etudiant extends Personne {
    private String section;
    private int bloc;

    public Etudiant( String prenom, String nom, char genre,
                    int jourNaissance, int moisNaissance, int anneeNaissance,
                    String section, int bloc ) {
        super(prenom, nom, genre,
              jourNaissance, moisNaissance, anneeNaissance);
        this.section = section;
        this.bloc = bloc;
    }
    ... // Public getters / setters et autres méthodes
}
```

Caractéristiques de la **personne**

Caractéristiques propres à l'**étudiant**

La classe *Etudiant* propose un constructeur à 8 arguments permettant d'initialiser ses 6 variables d'instance : *prenom*, *nom*, *genre*, *dateNaissance*, *section* et *bloc*.

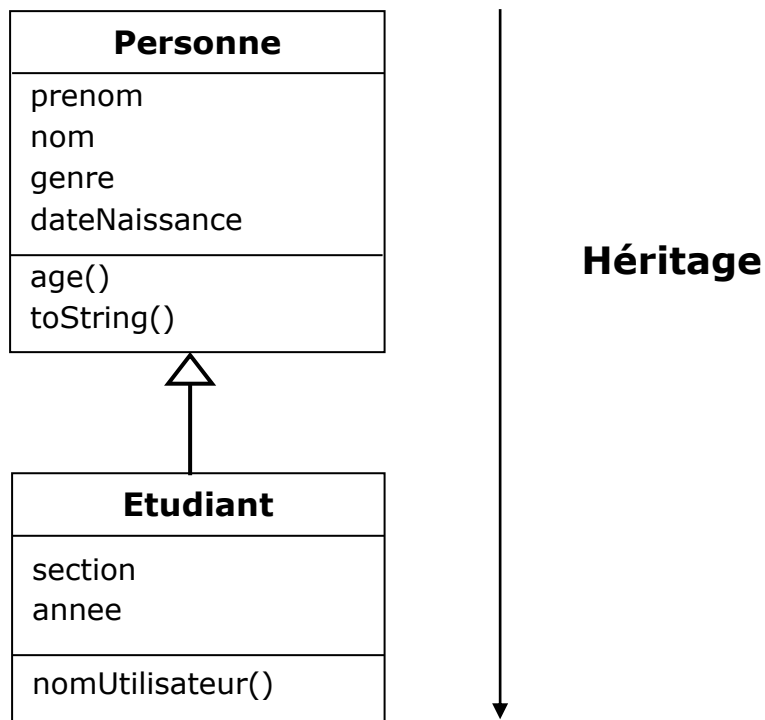
Il est possible de faire appel, au sein du constructeur de la sous-classe, au constructeur de la super-classe, ce qui permet de faire l'économie de lignes de

code. Cet appel se fait en utilisant le mot réservé **super** suivi entre parenthèses des arguments à passer au constructeur de la super-classe.

L'instruction "**super**(*prenom, nom, genre, jourNaissance, moisNaissance, anneeNaissance*)" signifie que l'on appelle le constructeur de la super-classe avec ces arguments. Cette instruction aura pour effet d'initialiser les variables d'instance correspondantes.

Attention, l'instruction `super(...)` doit toujours être placée en première position dans le constructeur de la sous-classe !

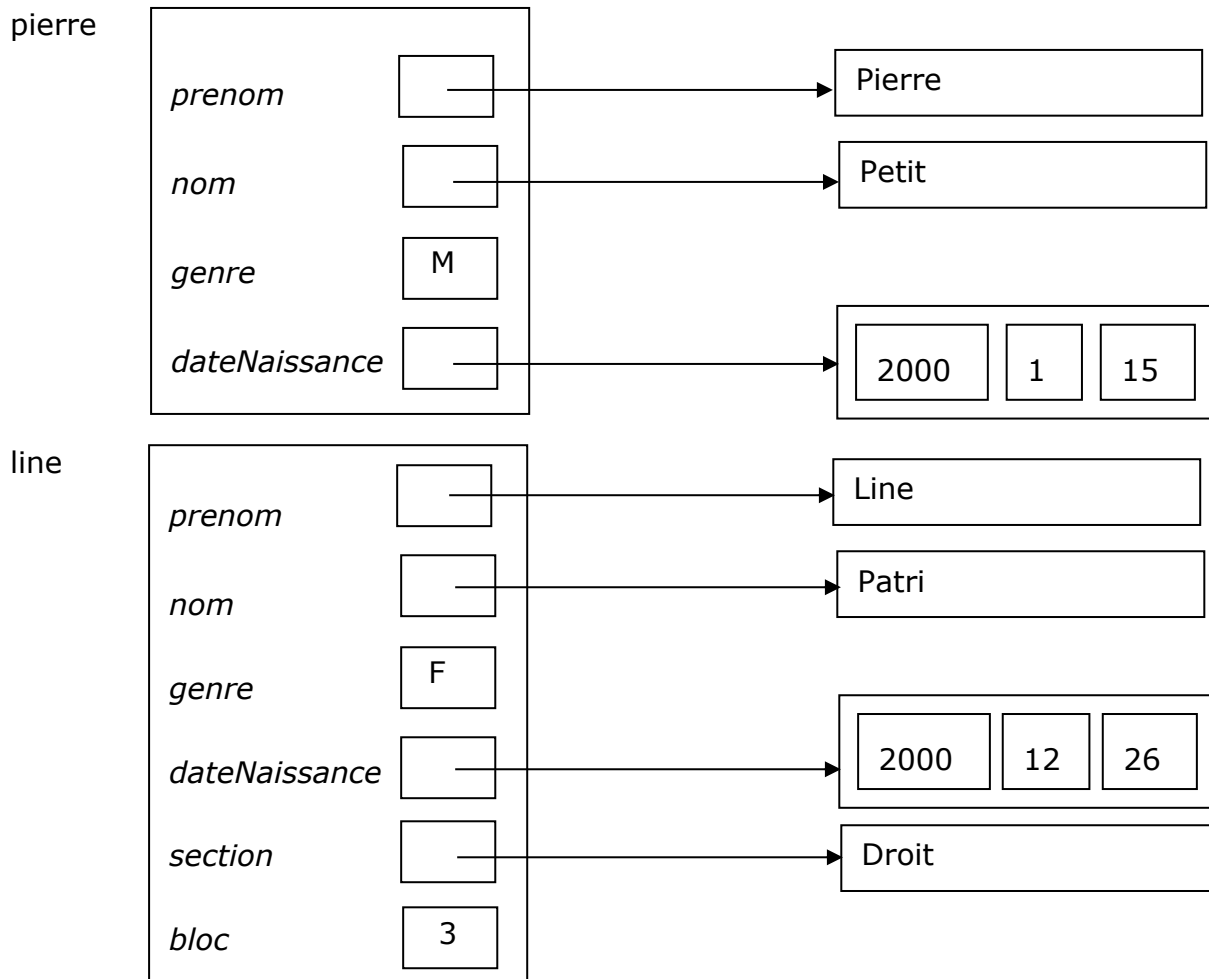
7.2 Héritage des variables d'instance et des méthodes



Comme dit précédemment, la classe *Etudiant* hérite de la classe *Personne*, ce qui signifie que toute occurrence de la classe *Etudiant* hérite des variables d'instance de la classe *Personne*, mais aussi de toute méthode de la classe *Personne*. Autrement dit, toute méthode de la classe *Personne* pourra être appelée sur tout objet de type *Etudiant* (par exemple, la méthode `age()`). Tout objet de type *Etudiant* aura donc en mémoire des variables *prenom*, *nom*, *genre* et *dateNaissance*.

```
public class Principal {
    public static void main(String [] args) {
        Personne pierre = new Personne("Pierre","Petit",'M',15,1,2000);
        Etudiant line = new Etudiant("Line","Patri",'F',26,12,2000,"Droit",3);
    }
}
```

En mémoire :



```
System.out.println(line.age()); // OK héritage de la méthode public age  
System.out.println(line);      // Appel implicite à toString
```

⇒ *toString* hérité de **Personne**

⇒ Affiche à l'écran :

La personne Line Patri

7.3 Accès aux variables d'instance privées héritées

Les variables d'instance *prenom*, *nom*, *genre* et *dateNaissance* d'un étudiant sont héritées de la classe *Personne*. Cela signifie qu'en mémoire tout objet de type *Etudiant* aura bien une place réservée pour les variables héritées *prenom*, *nom*, *genre* et *dateNaissance*. Cependant, **une sous-classe ne peut pas accéder directement aux variables d'instance, méthodes et constructeurs déclarés *private*** dans la super-classe.

Pour accéder aux variables d'instance privées héritées de la classe *Personne*, il faut utiliser les getters déclarés *public* dans la classe *Personne* : pour accéder aux variables d'instance privées *prenom*, *nom*, *genre* et *dateNaissance* héritées de la classe *Personne* il faut utiliser respectivement les getters publics *getPrenom*, *getNom*, *getGenre* et *getDateNaissance*.

```
public class Principal {
    public static void main(String [] args) {
        Etudiant line = new Etudiant("Line","Patri",'F',26,12,2000,"Droit",3);
        System.out.println(line.nom); /* Variable d'instance existante en mémoire
                                         Mais accès direct en lecture pas OK car variable d'instance privée
                                         ⇒ Utiliser le getter */

        System.out.println(line.getNom()); // OK car getter publique

        if (line.genre == 'F') /* Variable d'instance existante en mémoire
                               Mais accès direct en lecture pas OK car variable d'instance privée
                               ⇒ Utiliser le getter */

        if (line.getGenre() == 'F') { // OK getter publique
            System.out.println("Sois la bienvenue");
        }
        else {
            System.out.println("Sois le bienvenu");
        }
    }
}
```

```

public class Etudiant extends Personne {
    private String section;
    private int bloc;

    public Etudiant(String prenom, String nom, char genre,
        int jourNaissance, int moisNaissance, int anneeNaissance,
        String section, int bloc) {
        super(prenom, nom, genre,
            jourNaissance, moisNaissance, anneeNaissance);
        this.section = section;
        this.bloc = bloc;
    }

    ...    // Public getters / setters

    public String nomUtilisateur() {
        // Les deux premières lettres ↩
        return nom + prenom + section.substring(0,2) + bloc;
        // Pas ok car variables d'instance héritées mais private
        return getNom() + getPrenom() + section.substring(0,2) + bloc;
    }
}

```

De même, un constructeur de la classe *Etudiant* ne pourrait pas faire appel à un constructeur privé de la classe *Personne* (via *super (...)*).

Puisque la sous-classe hérite des variables d'instance et des méthodes de la super-classe, un objet de la sous-classe peut être affecté à un objet de la super-classe.

Exemple

```

Personne marie = new Etudiant("Marie", "Flore", 'F', 5, 8, 2001,
    "Marketing", 1);

System.out.println(marie.age());

// Appel possible des méthodes de Personne

```

En conclusion

① Pour réutiliser une classe existante et l'adapter, il suffit de créer une nouvelle classe en la déclarant sous-classe de la classe réutilisée. La classe réutilisée jouera le rôle de super-classe. La déclaration de la sous-classe est :

class SousClasse extends SuperClasse

② Une sous-classe hérite des variables d'instance de la super-classe. Toute occurrence de la sous-classe possède en mémoire une copie des variables d'instance héritées.

③ Une sous-classe hérite des méthodes de la super-classe. Toute méthode de la super-classe peut être appelée sur toute occurrence de la sous-classe.

④ La sous-classe peut contenir ses propres variables d'instance.

⑤ La sous-classe peut contenir ses propres méthodes.

⑥ Un objet d'une sous-classe peut être affecté à un objet de type d'une super-classe.

⑦ Une variable d'instance, constructeur ou méthode déclaré **private** dans une super-classe n'est **pas accessible** au sein de la sous-classe.

⑧ Pour accéder en lecture à une variable d'instance privée héritée, il faut utiliser le getter public de la super-classe.

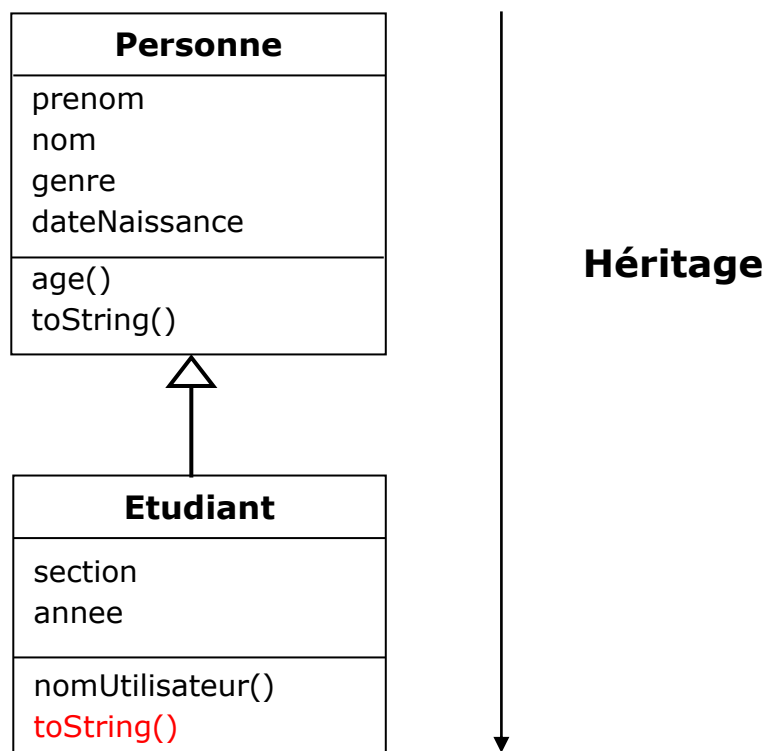
7.4 Redéfinition (overwriting/overriding) de méthode

Si l'on veut récupérer une classe existante et l'adapter à l'application courante, il suffit donc de créer une nouvelle classe qui est une sous-classe de la classe existante et d'y ajouter les caractéristiques qui lui sont propres. Une première façon d'adapter une classe existante est donc d'**ajouter de nouvelles caractéristiques** (variables d'instance et/ou méthodes).

Il est aussi possible d'**adapter des méthodes héritées** d'une super-classe. Pour adapter une méthode héritée qui ne conviendrait pas parfaitement dans la sous-classe, il suffit de réécrire cette méthode dans la sous-classe en prenant bien soin de garder **la même signature** que la méthode héritée. Quand une sous-classe contient une méthode de même signature qu'une méthode de la super-classe, on dit que la sous-classe **redéfinit** la méthode.

Il y a deux façons de redéfinir / d'adapter une méthode héritée :

- Soit en la **remplaçant** complètement (en réécrivant complètement son code) ;
- Soit en la **récupérant** et en l'**étendant** (en ajoutant du code au code existant).



Dans l'exemple ci-dessous, la méthode *toString* de la classe *Etudiant* redéfinit (écrase) complètement la méthode *toString* héritée de la classe *Personne*. Ultérieurement, nous verrons comment récupérer le code d'une méthode héritée et l'étendre (cf. méthode *toString* de la classe *EtudiantInformatique*).

L'annotation *@Override* symbolise cette redéfinition d'une méthode héritée.

```
public class Etudiant extends Personne {
    private String section;
    private int bloc;

    ...    // Constructeurs, getters, setters

    public String nomUtilisateur() {
        return getNom() + getPrenom() + section.substring(0,2) + bloc;
    }

    @Override
    public String toString() {
        return "L'étudiant " + nomUtilisateur() + " est inscrit au bloc "
            + bloc + " en " +section;
    }
}
```

```
public class Principal {
    public static void main(String [] args) {
        Personne pierre = new Personne("Pierre","Petit",'M',15,1,2000);
        System.out.println(pierre);
    }
}
```

⇒ Affiche à l'écran :

La personne Pierre Petit

```
Etudiant luc = new Etudiant("Luc","Right",'F',26,12,2000,"Droit",3);
System.out.println(luc);
```

⇒ Affiche à l'écran :

L'étudiant RightLucDr3 est inscrit au bloc 3 en Droit

Comment Java résout-il le conflit quand une méthode appelée sur un objet existe dans la classe correspondant à l'objet ainsi que dans la super-classe ?

On parle de liaison dynamique (**Dynamic Binding**). En Java, la recherche de la méthode débute toujours dans **la classe la plus spécifique**, c'est-à-dire dans la sous-classe. En l'occurrence, quand la méthode *toString* est appelée sur un objet de type *Etudiant*, Java cherche d'abord dans la classe *Etudiant* si la méthode *toString* s'y trouve. Comme elle s'y trouve, c'est cette méthode-là qui est exécutée. Si la méthode *toString* n'avait pas été redéfinie dans la classe *Etudiant*, Java aurait exécuté la méthode *toString* héritée de la classe *Personne*.

Si une méthode recherchée ne se trouve ni dans la sous-classe, ni dans aucune des super-classes, une **erreur** est détectée **à la compilation**.

N.B. Le cas de la méthode *toString* est un peu particulier. En effet, **toute classe** écrite par un programmeur **est implicitement une sous-classe de la super-classe *Object***. Or, **dans la super-classe *Object*, se trouve définie la méthode *toString***. Ce qui explique que si aucune méthode *toString* n'est définie dans une classe, aucune erreur ne sera détectée à la compilation si on appelle implicitement la méthode *toString* sur un objet. En effet, c'est la méthode *toString* de la classe *Object* qui sera exécutée. Notons que la chaîne de caractères ainsi produite est peu conviviale. Il est donc recommandé de redéfinir la méthode *toString* héritée de la classe *Object*.

7.5 Hiérarchie d'héritage

Le mécanisme d'héritage est applicable à plus de deux classes. On peut ainsi construire des **hiérarchies de classes**.

Le schéma ci-après propose une hiérarchie d'héritage composée de 5 classes :

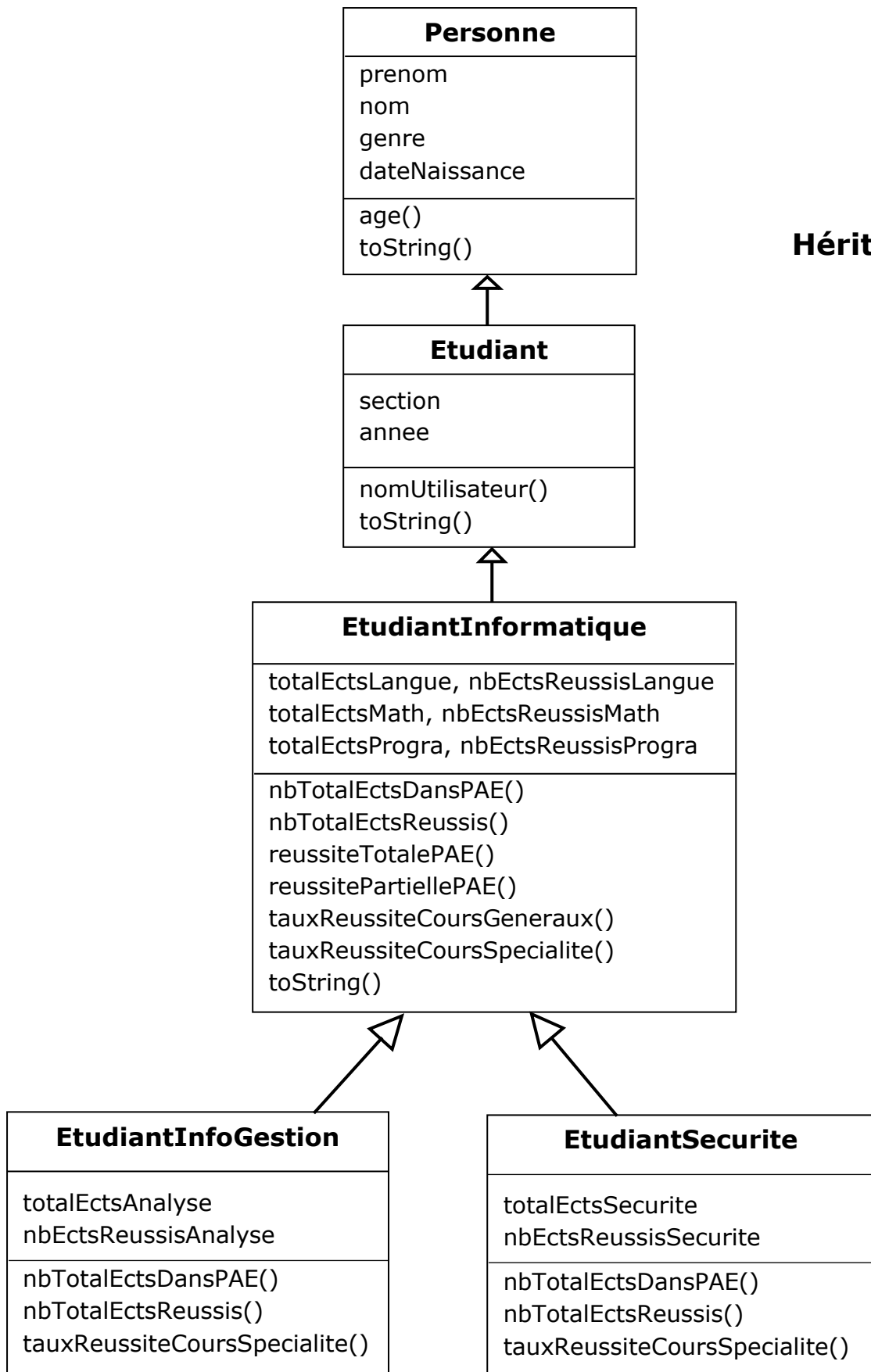
- Les classes *Personne* et *Etudiant* déjà rencontrées.
- La classe *EtudiantInformatique* permettant de gérer des étudiants inscrits en informatique : les nombres de crédits Ects du PAE en langues, mathématiques et programmation, ainsi que les nombres de ces crédits Ects réussis sont enregistrés sous forme de variables d'instance.

Les calculs du nombre total d'Ects dans le PAE, du nombre total d'Ects réussis, de la réussite totale ou partielle du PAE ainsi que les taux de réussite des cours généraux et de spécialité sont proposés sous forme de méthodes.

- La classe *EtudiantInfoGestion* permettant de gérer des étudiants inscrits en informatique de gestion : le nombre de crédits Ects en Analyse dans le PAE ainsi que le nombre d'Ects réussis en Analyse sont enregistrés sous forme de variables d'instance. Les méthodes de calcul du nombre total d'Ects dans le PAE, du nombre total d'Ects réussis et du taux de réussite des cours de spécialité doivent être **redéfinies pour tenir compte de la formation en Analyse**.
- La classe *EtudiantSecu* permettant de gérer des étudiants inscrits en Sécurité des systèmes : le nombre de crédits Ects en Sécurité dans le PAE ainsi que le nombre d'Ects réussis en Sécurité sont enregistrés sous forme de variables d'instance. Les méthodes de calcul du nombre total d'Ects dans le PAE, du nombre total d'Ects réussis et du taux de réussite des cours de spécialité doivent être **redéfinies pour tenir compte de la formation en Sécurité**.

Le mécanisme d'héritage est implémenté 4 fois :

- Entre la classe *Etudiant* et la classe *Personne*,
- Entre la classe *EtudiantInformatique* et *Etudiant*,
- Entre la classe *EtudiantInfoGestion* et *EtudiantInformatique* et
- Entre la classe *EtudiantSecu* et *EtudiantInformatique*.



Héritage

Le code des classes *Personne* et *Etudiant* a déjà été analysé. Le code des trois autres classes est proposé ci-après.

```
public class EtudiantInformatique extends Etudiant {

    private int totalEctsLangue;
    private int nbEctsReussisLangue;
    private int totalEctsMath;
    private int nbEctsReussisMath;
    private int totalEctsProgra;
    private int nbEctsReussisProgra;

    public EtudiantInformatique(String prenom, String nom, char genre,
        int jourNaissance, int moisNaissance, int anneeNaissance,
        String section, int bloc, int totalEctsLangue,
        int nbEctsReussisLangue, int totalEctsMath,
        int nbEctsReussisMath, int totalEctsProgra, int nbEctsReussisProgra) {
        super(prenom, nom, genre, jourNaissance, moisNaissance,
            anneeNaissance, section, bloc);
        this.totalEctsLangue = totalEctsLangue;
        this.nbEctsReussisLangue = nbEctsReussisLangue;
        this.totalEctsMath = totalEctsMath;
        this.nbEctsReussisMath = nbEctsReussisMath;
        this.totalEctsProgra = totalEctsProgra;
        this.nbEctsReussisProgra = nbEctsReussisProgra;
    }

    ...    // Public getters et setters

    public int nbTotalEctsDansPAE() {
        return totalEctsLangue + totalEctsMath + totalEctsProgra;
    }

    public int nbTotalEctsReussis() {
        return nbEctsReussisLangue + nbEctsReussisMath +
            nbEctsReussisProgra;
    }

    public boolean reussiteTotalePAE() {
        return this.nbTotalEctsReussis() == this.nbTotalEctsDansPAE();
    }

    public boolean reussitePartiellePAE() {
        return !this.reussiteTotalePAE() && getBloc() == 1
            && this.nbTotalEctsReussis() >= 45;
    }
}
```

```

public double tauxReussiteCoursGeneraux() {
    return ((nbEctsReussisLangue + nbEctsReussisMath) /
            (double) (totalEctsLangue + totalEctsMath)) * 100;
}

public double tauxReussiteCoursSpecialite() {
    return (nbEctsReussisProgra / (double) totalEctsProgra) * 100;
}

@Override
public String toString() {
    return super.toString() +
           (reussiteTotalePAE()?"\na réussi l'entièreté de son PAE"
            : "\nn'a pas réussi l'entièreté de son PAE")
           + "\navec " + tauxReussiteCoursSpecialite()
           + " de taux de réussite dans les cours de spécialité";
}
}

```

Appel à la méthode héritée

Nous avons vu précédemment comment la méthode *toString* de la classe *Etudiant* avait redéfini la méthode *toString* héritée de la classe *Personne* : le code de la méthode héritée avait été purement et simplement **été écrasé et remplacé** par le code de la nouvelle méthode *toString* de la classe *Etudiant*.

Ici aussi, la méthode *toString* de la classe *EtudiantInformatique* redéfinit la méthode *toString* héritée de la classe *Etudiant*. Mais, le code de la méthode *toString* de la classe *EtudiantInformatique* ne remplace pas purement et simplement le code de la méthode *toString* héritée. Il **récupère et étend** le code de la méthode *toString* héritée de la classe *Etudiant*.

L'instruction : ***super.toString()*** + ...

a pour effet d'appeler la méthode *toString* de la super-classe et donc de récupérer la chaîne de caractères construite par la méthode héritée, en vue d'y ajouter une autre chaîne de caractères.

```

public class Principal {
    public static void main(String [] args) {
        EtudiantInformatique marc =
            new EtudiantInformatique("Marc","Infol",'M',18,8,1998,
                                     "Technologie de l'informatique",2,11,11,9,9,40,20);
        System.out.println(marc);
    }
}

```

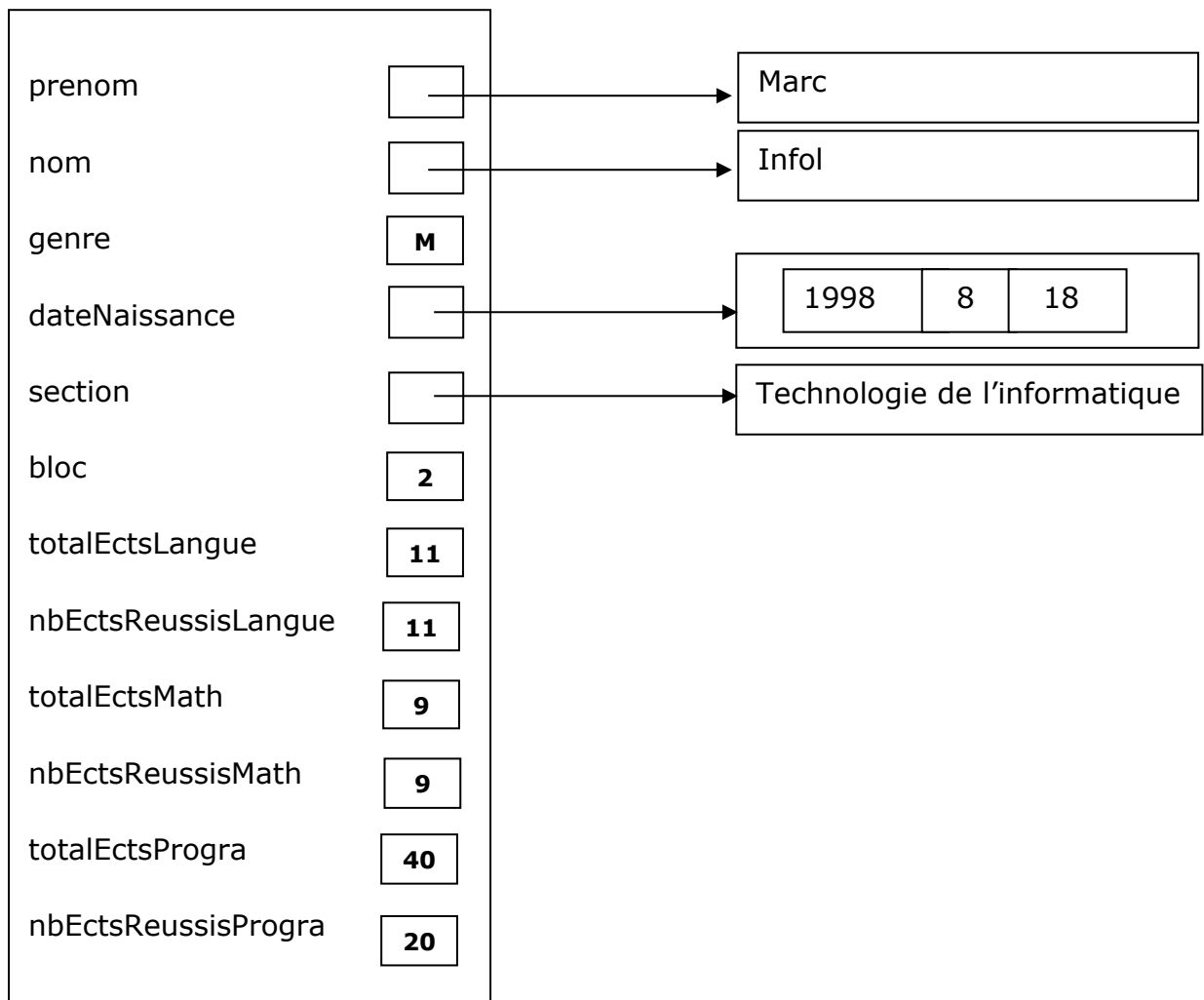
⇒ Affiche à l'écran :

*L'étudiant InfolMarcTe2 est inscrit au bloc 2 en Technologie de l'informatique
n'a pas réussi l'entièreté de son PAE
avec 50.0 de taux de réussite dans les cours de spécialité*

La première ligne de l'affichage provient de la méthode *toString* de la classe *Etudiant*, tandis que les deux dernières lignes proviennent de la méthode *toString* de la classe *EtudiantInformatique*.

En mémoire :

Marc



```

public class EtudiantInfoGestion extends EtudiantInformatique {
    private int totalEctsAnalyse ;
    private int nbEctsReussisAnalyse;

    public EtudiantInfoGestion(String prenom, String nom, char genre,
        int jourNaissance, int moisNaissance, int anneeNaissance, int bloc,
        int totalEctsLangue, int nbEctsReussisLangue, int totalEctsMath,
        int nbEctsReussisMath, int totalEctsProgra, int nbEctsReussisProgra
        int totalEctsAnalyse, int nbEctsReussisAnalyse) {
        super(prenom, nom, genre, jourNaissance, moisNaissance,
            anneeNaissance, "Informatique de gestion", bloc,
            totalEctsLangue, nbEctsReussisLangue, totalEctsMath,
            nbEctsReussisMath, totalEctsProgra, nbEctsReussisProgra);
        this.totalEctsAnalyse = totalEctsAnalyse;
        this.nbEctsReussisAnalyse = nbEctsReussisAnalyse;
    }

    ... // Public getters et setters

    @Override
    public int nbTotalEctsDansPAE() {
        return super.nbTotalEctsDansPAE() + totalEctsAnalyse;
    }

    @Override
    public int nbTotalEctsReussis() {
        return super.nbTotalEctsReussis() + nbEctsReussisAnalyse;
    }

    @Override
    public double tauxReussiteCoursSpecialite() {
        return ( (getNbEctsReussisProgra()+ nbEctsReussisAnalyse) /
            (double) (getTotalEctsProgra() + totalEctsAnalyse) ) * 100;
    }
}

```

Notons que la variable d'instance *section* de l'étudiant existe quand même en mémoire (avec la valeur "*Informatique de gestion*"), même si elle n'apparaît pas dans les arguments du constructeur.

La classe *EtudiantInfoGestion* présente aussi un exemple de redéfinition de méthode qui remplace totalement le code de la méthode héritée (cf. méthode *tauxReussiteCoursSpecialite*) et deux exemples de redéfinition de méthode qui

recupèrent et étendent le code de la méthode héritée (cf. méthodes *nbTotalEctsDansPAE* et *nbTotalEctsReussis*).

```
public class EtudiantSecurite extends EtudiantInformatique {
    private int totalEctsSecurite;
    private int nbEctsReussisSecurite;

    public EtudiantSecurite(String prenom, String nom, char genre,
        int jourNaissance, int moisNaissance, int anneeNaissance, int bloc,
        int totalEctsLangue, int nbEctsReussisLangue, int totalEctsMath,
        int nbEctsReussisMath, int totalEctsProgra, int nbEctsReussisProgra,
        int totalEctsSecurite, int nbEctsReussisSecurite) {
        super(prenom, nom, genre, jourNaissance, moisNaissance,
            anneeNaissance, "Sécurité des systèmes", bloc,
            totalEctsLangue, nbEctsReussisLangue, totalEctsMath,
            nbEctsReussisMath, totalEctsProgra, nbEctsReussisProgra);
        this.totalEctsSecurite = totalEctsSecurite;
        this.nbEctsReussisSecurite = nbEctsReussisSecurite;
    }
    ... // Public getters et setters

    @Override
    public int nbTotalEctsDansPAE() {
        return super.nbTotalEctsDansPAE() + totalEctsSecurite;
    }

    @Override
    public int nbTotalEctsReussis() {
        return super.nbTotalEctsReussis() + nbEctsReussisSecurite;
    }

    @Override
    public double tauxReussiteCoursSpecialite() {
        return ( (getNbEctsReussisProgra()+ nbEctsReussisSecurite) /
            (double) (getTotalEctsProgra() + totalEctsSecurite) ) * 100;
    }
}
```

En conclusion

- ① Une sous-classe peut adapter une méthode héritée qui ne conviendrait pas exactement. On parle alors de **redéfinition de méthode**. Pour redéfinir une méthode héritée, il suffit de prévoir dans la sous-classe, une méthode possédant la **même signature** que la méthode héritée à redéfinir.
- ② Une méthode peut être redéfinie de deux façons :
 - Soit en **remplaçant** purement et simplement le code de la méthode héritée ;
 - Soit en **recupérant** et en **étendant** le code de la méthode héritée : l'instruction pour appeler la méthode héritée est :
super.methodeHeritee (...) ...
- ③ Une méthode héritée redéfinie sera annotée **@Override** par le compilateur.

7.6 Polymorphisme

Étant donné qu'une méthode héritée de la super-classe peut être redéfinie dans une sous-classe, la même méthode (même signature) peut être présente dans plusieurs classes de la même hiérarchie. Comment Java choisit-il la méthode à exécuter ?

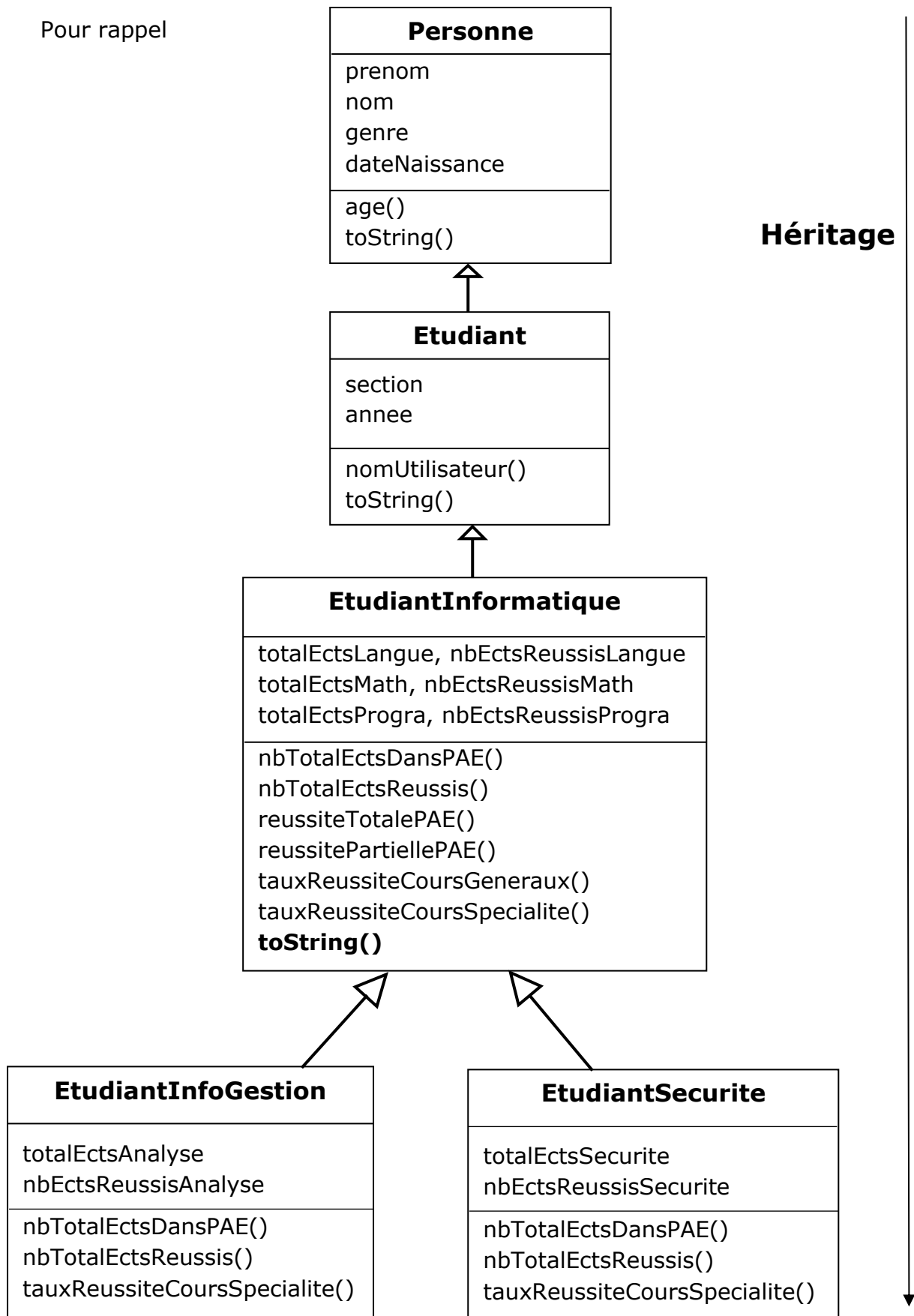
Ce n'est parfois qu'à l'exécution (et donc pas toujours à la compilation) que Java sait quelle méthode exécuter en fonction du type de l'objet sur lequel on a appelé la méthode.

Le principe est d'appeler la méthode la plus spécifique, c'est-à-dire la méthode qui se trouve le plus bas possible dans la hiérarchie d'héritage.

Quand une méthode *methodeX* est appelée sur un objet **o**, Java recherche la classe correspondant à l'objet **o**, soit la classe *ClassA*. Java vérifie si la méthode appelée (*methodeX*) est définie dans la classe *ClassA*. Si oui, cette méthode est exécutée. Sinon, Java remonte la hiérarchie d'héritage à partir de la classe *ClassA*. La méthode *methodeX* est recherchée dans la super-classe de la classe *ClassA* (soit *ClassB*, la super-classe de la classe *ClassA*). Si la méthode se trouve dans la classe *ClassB*, elle est exécutée. Sinon, Java continue de remonter la hiérarchie d'héritage et recherche la méthode *methodeX* dans la super-classe de la classe *ClassB*, et ainsi de suite jusqu'à trouver la méthode *methodeX*. Si la méthode recherchée n'est trouvée dans aucune des super-classes de la hiérarchie, une erreur est détectée dès la compilation.

Dans la hiérarchie ci-dessous, les classes *EtudiantInfoGestion* et *EtudiantSecu* n'ont pas de méthode *toString*. Une méthode *toString* est donc recherchée dans une des super-classes. Une méthode *toString* existe dans la classe parent *EtudiantInformatique*. C'est donc cette méthode qui sera exécutée pour présenter décrire les objets de type *EtudiantInfoGestion* et *EtudiantSecu*.

Pour rappel




```

public class Principal {
    public static void main(String [] args) {
        EtudiantInfoGestion alan =
            new EtudiantInfoGestion("Alan","Turing",'M',15,6,2002,2,
                                    11,11,9,9,40,40,5,5);

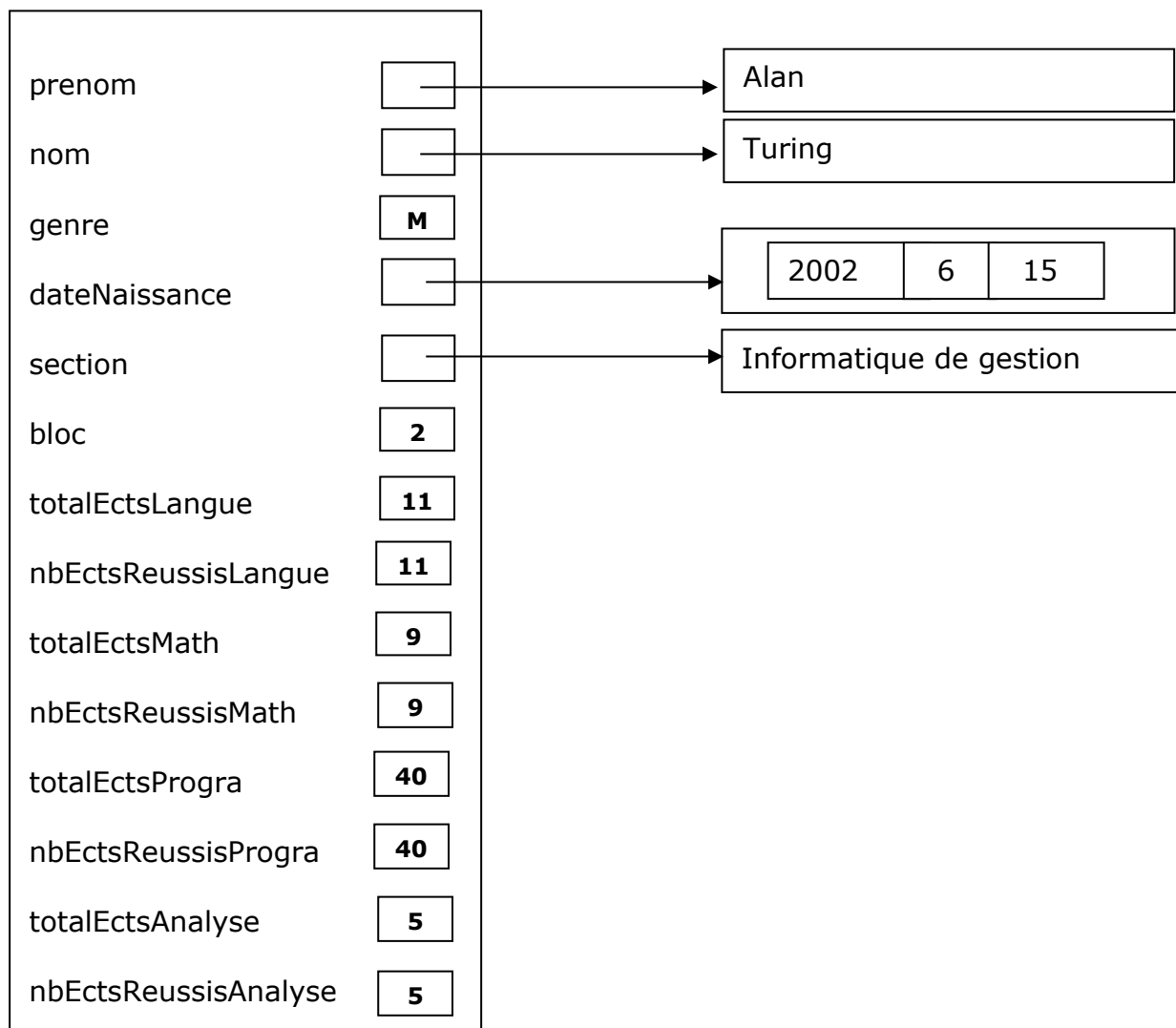
        System.out.println(alan);
    }
}

```

⇒ Affiche à l'écran :

*L'étudiant TuringAlanIn2 est inscrit au bloc 2 en Informatique de gestion
a réussi l'entièreté de son PAE
avec 100.0 de taux de réussite dans les cours de spécialité*

En mémoire : *alan*



La dernière instruction fait appel implicitement à la méthode *toString*. Or, la méthode *toString* ne se trouve pas redéfinie dans la classe *EtudiantInfoGestion*. Java remonte donc la hiérarchie d'héritage à la recherche de cette méthode, et ce, en partant de la classe correspondant à l'objet *alan*. La super-classe de la classe *EtudiantInfoGestion* est la classe *EtudiantInformatique*. La méthode *toString* est redéfinie dans cette classe. C'est donc ce code-là qui sera exécuté :

```
@Override
public String toString() {
    return super.toString() +
        (reussiteTotalePAE())?"\na réussi l'entièreté de son PAE"
        :"\nn'a pas réussi l'entièreté de son PAE")
    + "\navec " + tauxReussiteCoursSpecialite()
    + " de taux de réussite dans les cours de spécialité";
}
```

La première instruction (***super.toString()***) fait appel à la méthode *toString* de la super-classe *Etudiant* :

```
@Override
public String toString() {
    return "L'étudiant " + nomUtilisateur() + " est inscrit au bloc "
        + bloc + " en " +section;
}
```

La chaîne de caractères retournée par cet appel est :

L'étudiant TuringAlanIn2 est inscrit au bloc 2 en Informatique de gestion

Ensuite, la méthode *reussiteTotalePAE* est appelée sur l'objet courant *alan*. Java tente de nouveau de rechercher cette méthode à partir de la classe correspondant à *alan*, c'est-à-dire dans la classe *EtudiantInfoGestion*. La méthode *reussiteTotalePAE* ne s'y trouve pas. Java remonte donc la hiérarchie d'héritage à la recherche de cette méthode. La méthode *reussiteTotalePAE* se trouve dans la super-classe *EtudiantInformatique*.

C'est donc ce code-là qui sera exécuté :

```
public boolean reussiteTotalePAE() {
    return this.nbTotalEctsReussis() == this.nbTotalEctsDansPAE();
}
```

Les méthodes *nbTotalEctsReussis* et *nbTotalEctsDansPAE* doivent à leur tour être exécutées. Notons que la méthode *nbTotalEctsReussis* et *nbTotalEctsDansPA* sont

définies dans la classe *EtudiantInformatique*. Mais, comme convenu, Java recherche ces méthodes en partant d'abord de la classe correspondant à l'objet *alan*. La recherche débute donc dans la classe *EtudiantInfoGestion*. Or, les méthodes *nbTotalEctsReussis* et *nbTotalEctsDansPAE* s'y trouvent redéfinies. Ce sont donc ces codes-là qui seront exécutés et en aucun cas, les méthodes *nbTotalEctsReussis* et *nbTotalEctsDansPAE* qui se trouvent dans la classe *EtudiantInformatique*, **même si c'est la méthode *reussiteTotalePAE* de cette classe-là qui a été exécutée.**

Codes des méthodes *nbTotalEctsReussi* et *nbTotalEctsDansPAE* de la classe *EtudiantInfoGestion* qui seront exécutés :

```
@Override
public int nbTotalEctsDansPAE() {
    return super.nbTotalEctsDansPAE() + totalEctsAnalyse;
}

@Override
public int nbTotalEctsReussis() {
    return super.nbTotalEctsReussis() + nbEctsReussisAnalyse;
}
```

Le raisonnement est le même pour la méthode *tauxReussiteCoursSpecialite*. C'est la méthode de la classe *EtudiantInfoGestion* qui sera exécutée et non celle de la classe *EtudiantInformatique* :

```
@Override
public double tauxReussiteCoursSpecialite() {
    return ((getNbEctsReussisProgra()+ nbEctsReussisAnalyse) /
            (double) (getTotalEctsProgra() + totalEctsAnalyse)) * 100;
}
```

Dans tous les cas, c'est donc **le code de la méthode la plus spécifique qui sera exécuté**. Autrement dit, c'est le code de **la méthode** qui sera définie **dans la classe se trouvant le plus bas possible dans la hiérarchie d'héritage** qui sera exécuté.

```

public class Principal {
    public static void main(String [] args) {
        EtudiantSecurite jane =
            new EtudiantSecurite("Jane","Hacker",'F',22,5,2000,2,
                                11,11,9,0,35,5,5,5);

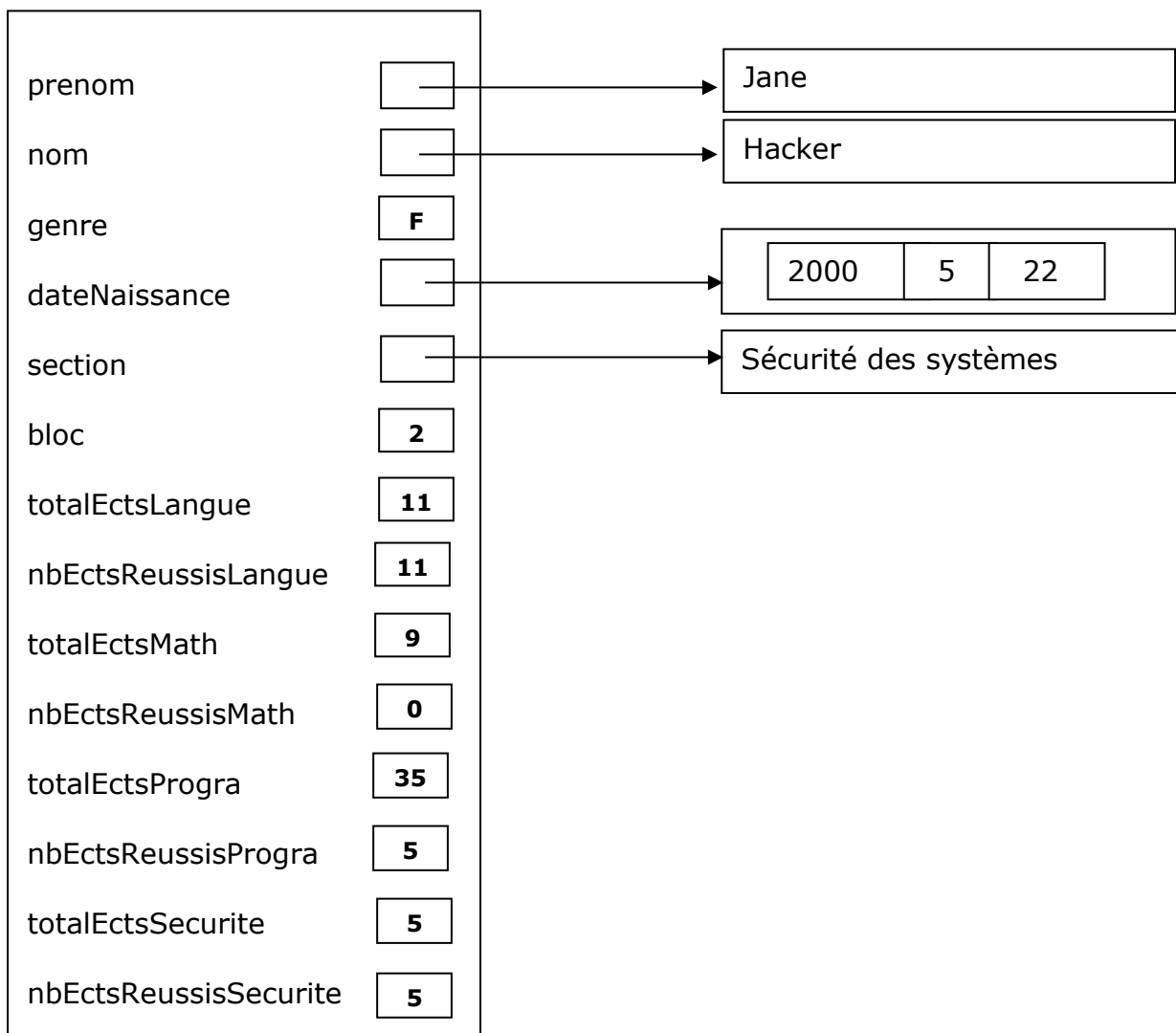
        System.out.println(jane);
    }
}

```

⇒ Affiche à l'écran :

*L'étudiant HackerJaneSe2 est inscrit au bloc 2 en Sécurité des systèmes
n'a pas réussi l'entièreté de son PAE
avec 25.0 de taux de réussite dans les cours de spécialité*

En mémoire : jane



Le polymorphisme a été illustré dans l'exemple proposé.

Ce n'est qu'à l'exécution que Java peut déterminer quelles méthodes doivent être exécutées en fonction du type de l'objet sur lequel les méthodes sont appelées.

En conclusion

① Quand il y a appel d'une méthode sur un objet, Java recherche cette méthode d'abord dans la classe correspondant à l'objet. Si elle ne s'y trouve pas, Java remonte la hiérarchie d'héritage et exécute la première méthode trouvée. Si la méthode a été redéfinie dans plusieurs classes, ce principe assure que c'est la méthode **la plus spécifique** qui sera exécutée (c'est-à-dire celle qui se trouve le plus bas possible dans la hiérarchie d'héritage).

② On parle de polymorphisme lorsque ce n'est qu'**à l'exécution** qu'on sait déterminer **quelle méthode exécuter** si celle-ci (même signature) existe dans plusieurs classes. Le choix de la méthode à exécuter est déterminé en fonction du type de l'objet sur lequel la méthode est appelée. Il est alors **impossible de compiler** un tel code.

③ Une classe ne peut être sous-classe que d'une seule super-classe. **L'héritage multiple est interdit** en Java : une classe ne peut pas hériter de plusieurs super-classes.

7.7 Protection de type *protected*

Les trois types de protection déjà rencontrés sont :

- ***private***
- ***package*** (aucun mot clé)
- ***public***

Il existe un quatrième type de protection : ***protected***

Ce dernier type de protection se situe entre les protections *package* et *public*. En effet, la protection *protected* est plus permissive que la protection de type *package* et moins permissive que la protection *public*.

La protection de type *protected* peut s'appliquer aux **variables d'instance**, aux **méthodes** (et donc constructeurs), mais en aucun cas à une classe. Par conséquent, les seules protections permises dans une déclaration de classe sont : *public* et aucune protection (c'est-à-dire protection de type *package*).

Exemples

```
|| public ClassA { ... }
```

ou

```
|| ClassA { ... }      ⇒ Protection de type package
```

La protection de type *protected* associée à une variable d'instance ou une méthode (ou constructeur) signifie que cette variable d'instance ou cette méthode reste **accessible au sein du même package** (car inclut la protection de type *package*) mais également **par toutes les sous-classes même** celles qui seraient éventuellement créées **dans d'autres packages**. Autrement dit, les variables d'instance et méthodes déclarées *protected* dans une super-classe **sont accessibles dans toutes les sous-classes** créées dans le même package ou **dans un autre package**.

Illustrons ce principe de la façon suivante : reprenons les classes ***Personne***, ***Etudiant*** et ***EtudiantInformatique*** déjà étudiées et plaçons les dans un ***packageA***. Partons du principe que le concepteur de ces classes transmette son *packageA* à d'autres programmeurs, mais qu'il souhaite permettre l'accès aux méthodes et constructeurs que contiennent les classes du *packageA* qu'à la seule condition que les programmeurs créent des sous-classes des classes *Personne*, *Etudiant* ou *EtudiantInformatique*.

Déclarons donc **protected** toutes les variables d'instance, les constructeurs et les méthodes des classes du *packageA*, à l'exception, bien entendu, de la méthode *toString* dont la déclaration est immuable (**public** *String toString ()*).

Déclarons **public** ces trois classes pour qu'elles puissent être utilisées dans un autre package.

Pour rappel, une classe ne peut être déclarée *protected* ; nous n'avons le choix qu'entre *public* ou la protection de type *package*.

```
package packageA;
public class Personne {
    protected String prenom;
    protected String nom;
    protected char genre;
    protected LocalDate dateNaissance;
    protected Personne(...) {...}
    protected int age() { ...}
    public String toString() {...}
}
```

⇨ Variables d'instance

⇨ Constructeur

⇨ Méthodes

```
package packageA;
public class Etudiant extends Personne {
    protected String section;
    protected int annee;
    protected Etudiant(...) {...}
    protected String nomUtilisateur() { ... }
    public String toString() {...}
}
```

⇨ Variables d'instance

⇨ Constructeur

⇨ Méthodes

```
package packageA;
public class EtudiantInformatique extends Etudiant {
    protected int totalEctsLangue;
    protected int nbEctsReussisLangue;
    protected int totalEctsMath;
    protected int nbEctsReussisMath;
    protected int totalEctsProgra;
    protected int nbEctsReussisProgra;
}
```

⇨ Variables d'instance

```

protected EtudiantInformatique(...) {...}           ⇨ Constructeur
protected int nbTotalEctsDansPAE() {...}             ⇨ Méthodes
protected int nbTotalEctsReussis() {...}
protected boolean reussiteTotalePAE() { ...}
protected boolean reussitePartiellePAE() {...}
protected double tauxReussiteCoursGeneraux() {...}
protected double tauxReussiteCoursSpecialite() {...}
public String toString() {...}
}

```

Soit un second programmeur qui importe le *packageA* dans des classes de son programme afin de les utiliser (cf. l'instruction : ***import packageA.* ;***).

Ce programmeur crée la classe *EtudiantTechnoInfo* pour gérer des étudiants inscrits en Technologie de l'informatique. Ces étudiants doivent suivre une unité d'enseignement en réseaux (cours de spécialité).

Cette classe peut être déclarée **sous-classe** de la classe *EtudiantInformatique*, car cette dernière qui se trouve pourtant dans un autre package a été déclarée **public** et toutes les classes du *packageA* ont été importées.

```

package packageB;
import packageA.*;
public class EtudiantTechnoInfo extends EtudiantInformatique {
    private int totalEctsReseaux;
    private int nbEctsReussisReseaux;

    public EtudiantTechnoInfo(...) {
        super(...);
        ...
    }

    @Override
    protected int nbTotalEctsDansPAE() {
        return super.nbTotalEctsDansPAE() + totalEctsReseaux;
    }

    @Override
    protected int nbTotalEctsReussis() {
        return super.nbTotalEctsReussis() + nbEctsReussisReseaux;
    }
}

```



```

@Override
protected double tauxReussiteCoursSpecialite() {
    return ((nbEctsReussisProgra + nbEctsReussisReseaux) /
            (double) (totalEctsProgra + totalEctsReseaux)) * 100;
}
}

```

Le constructeur de la classe *EtudiantTechnoInfo* peut faire appel via l'instruction ***super (...)*** au constructeur hérité, car le constructeur de la super-classe a été déclaré *protected* : il peut donc être appelé par une sous-classe se trouvant dans un autre package.

Les méthodes *nbTotalEctsDansPAE*, *nbTotalEctsReussis* et *tauxReussiteCoursSpecialite* doivent être déclarées *protected* ou *public*. En effet, **toute méthode héritée peut être redéfinie à condition de l'être avec la même protection ou une protection plus permissive**. Ces méthodes doivent donc être redéfinies avec la protection *protected* ou *public* mais en aucun cas sans protection ce qui équivaldrait à la protection de type *package* (aucun mot clé). En effet, la protection de type *package* est plus restrictive que la protection *protected*.

Les variables d'instance *nbEctsReussisProgra* et *totalEctsProgra* sont accessibles car déclarées *protected* dans la super-classe, et donc accessibles par les sous-classes même situées dans un autre package.

Les instructions *super.nbTotalEctsDansPAE()* et *super.nbTotalEctsReussis()* peuvent être exécutées. Il s'agit d'appel aux méthodes *nbTotalEctsDansPAE* et *nbTotalEctsReussis* héritées. Or, celles-ci sont déclarées *protected* dans la super-classe. Elles sont donc accessibles.

```

package packageB;
import packageA.*;
public class Principal {
    public static void main(String [] args) {
        Etudiant line;
        line = new Etudiant("Line","Patri",'F',26,12,2000,"Droit",3); // ☹️
        EtudiantTechnoInfo alan ;
        alan = new EtudiantTechnoInfo("Alan","Turing",'M',26,11,2000,2,
                                      11,11,9,9,40,40,5,5);
    }
}

```

```

System.out.println(alan);
System.out.println(alan.age()); // ☹️
System.out.println(alan.tauxReussiteCoursSpecialite());
}
}

```

Un objet de type *Etudiant* peut être déclaré dans la classe *Principal*, car la classe *Etudiant* a été déclarée *public* dans le package importé.

Par contre, l'objet déclaré de type *Etudiant* ne peut être créé en mémoire et y être initialisé en appelant le constructeur de la classe *Etudiant*, car ce constructeur est déclaré *protected* et **la classe *Principal* n'est ni dans le même package, ni une sous-classe de la classe *Etudiant*.** Cela provoque donc une erreur à la compilation.

Un objet de type *EtudiantTechnoInfo* peut être déclaré car la classe *EtudiantTechnoInfo* a été déclarée avec la protection de type *public*. Cet objet peut être créé en mémoire et initialisé en appelant le constructeur de la classe *EtudiantTechnoInfo*, car celui-ci est déclaré de type *public*.

La méthode *toString* peut être appelée sur l'objet de type *EtudiantTechnoInfo* car elle est déclarée *public*.

Par contre, la méthode *age* ne peut être appelée sur un objet de type *EtudiantTechnoInfo*, car elle est déclarée *protected* dans la classe *Personne* et **la classe *Principal* n'est ni une sous-classe située dans la hiérarchie d'héritage de la classe *Personne*, ni une classe dans le même package que la classe *Personne*.**

La méthode *tauxReussiteCoursSpecialite* peut, elle, être appelée sur un objet de type *EtudiantTechnoInfo*, car elle est déclarée *protected* dans la classe *EtudiantTechnoInfo* et **la classe *Principal* se trouve dans le même package que la classe *EtudiantTechnoInfo*.**

En conclusion

- ① Les quatre types de protection possibles sont (de la plus restrictive à la plus permissive) : ***private***, ***package*** (aucun mot clé), ***protected*** et ***public***.
- ② Ces quatre types de protection peuvent être associés aux variables d'instance, constructeurs et méthodes.
- ③ Une variable d'instance, méthode ou constructeur déclaré ***protected*** est accessible **au sein du même package** et par **toutes les sous-classes** (se trouvant dans le même package ou non).
- ④ Les classes ne peuvent être déclarées qu'avec la protection ***public*** ou la protection de type package (aucun mot-clé).
- ⑤ Les méthodes héritées qui sont **redéfinies** doivent l'être avec une **protection égale à ou plus permissive que** la protection de la méthode héritée.
- ⑥ La méthode *toString* a une déclaration immuable :
public String toString ()....

8 Les mots-clés *static* et *final*

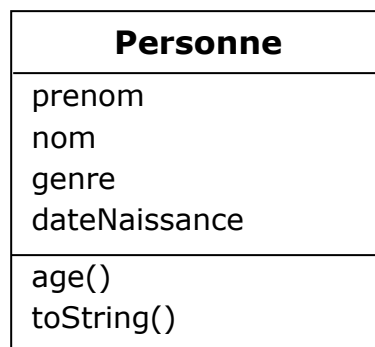
Variable et méthode de classe (*static*)

8.1 Variable de classe

Pour rappel, chaque objet d'une classe possède en mémoire un espace qui lui est propre pour ses variables d'instance.

Exemple

Soit la classe *Personne*

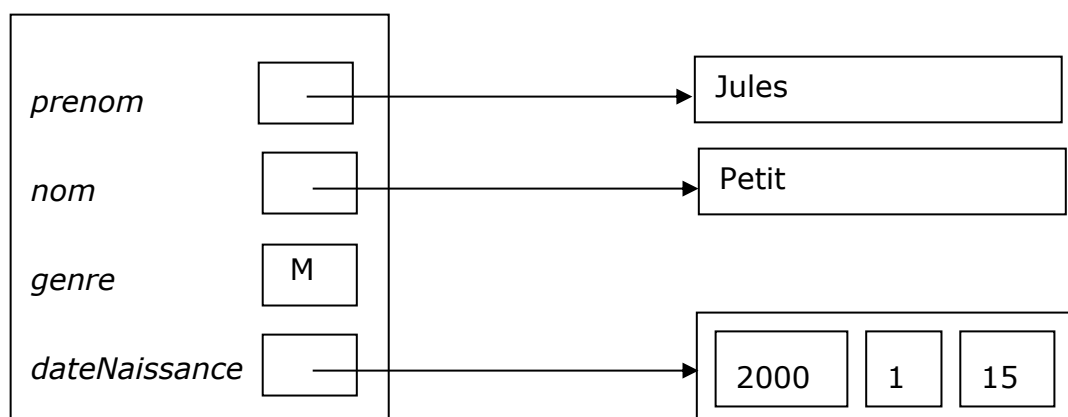


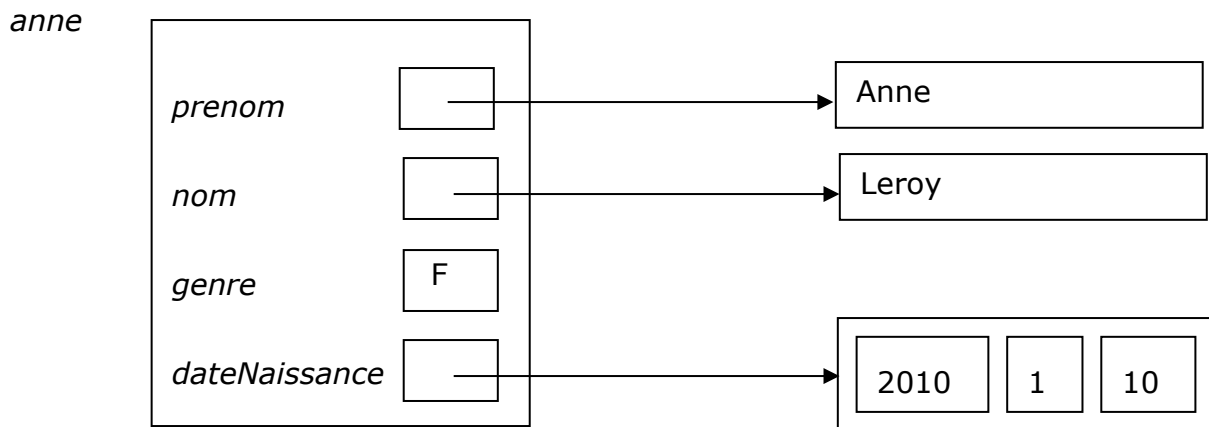
Soient les instructions de création des objets *jules* et *anne* de type *Personne* :

```
Personne jules = new Personne("Jules", "Petit", 'M', 15, 1, 2000);  
Personne anne = new Personne("Anne", "Leroy", 'F', 10, 1, 2010);
```

En mémoire, les objets *jules* et *anne* ont la structure suivante :

jules





Il est possible cependant de déclarer dans une classe des propriétés qui ne sont pas des caractéristiques propres à chaque objet de la classe comme le sont les variables d'instance, mais des propriétés qui sont des **caractéristiques de la classe**. On parle alors de variables de classe. Une variable de classe est une propriété de la classe, quel que soit le nombre d'objets créés. Il y a alors **un seul espace alloué en mémoire pour une variable de classe**, et ce, quel que soit le nombre d'objets de la classe créés : 0, 1 ou plusieurs.

Un objet d'une classe, quant à lui, ne possèdera pas en mémoire de copie qui lui sera propre des variables de la classe.

Exemple

Reprenons la classe *Personne*.

Une caractéristique de la classe *Personne* est par exemple :

- *Le nombre de femmes* créées par le programme principal (c'est-à-dire le nombre d'objets de type *Personne* qui sont du *genre* féminin)

ou encore,

- *La moyenne d'âge* des personnes (c'est-à-dire des objets de type *Personne*) créées par le programme principal.

Il faudrait prévoir alors trois variables de classe, à savoir,

- La variable **nbFemmes** pour comptabiliser les objets de type *Personne* qui sont du *genre* féminin créés par le programme principal
- Les variables **nbPersonnes** et **totalAges** pour comptabiliser le nombre de personnes créées et totaliser les âges de ces personnes, en vue de calculer la moyenne d'âge : ($moyenneAge = totalAges / nbPersonnes$).

Une variable de classe est déclarée en Java via le mot réservé **static**.

Personne
<ul style="list-style-type: none"> - prenom - nom - genre - dateNaissance - static nbFemmes - static nbPersonnes - static totalAges
<ul style="list-style-type: none"> + age() + toString()

Le principe de l'**Information Hiding** reste de mise pour les variables de classe. Il est donc recommandé de déclarer les variables de classe avec la protection **private**, comme on le fait pour les variables d'instance.

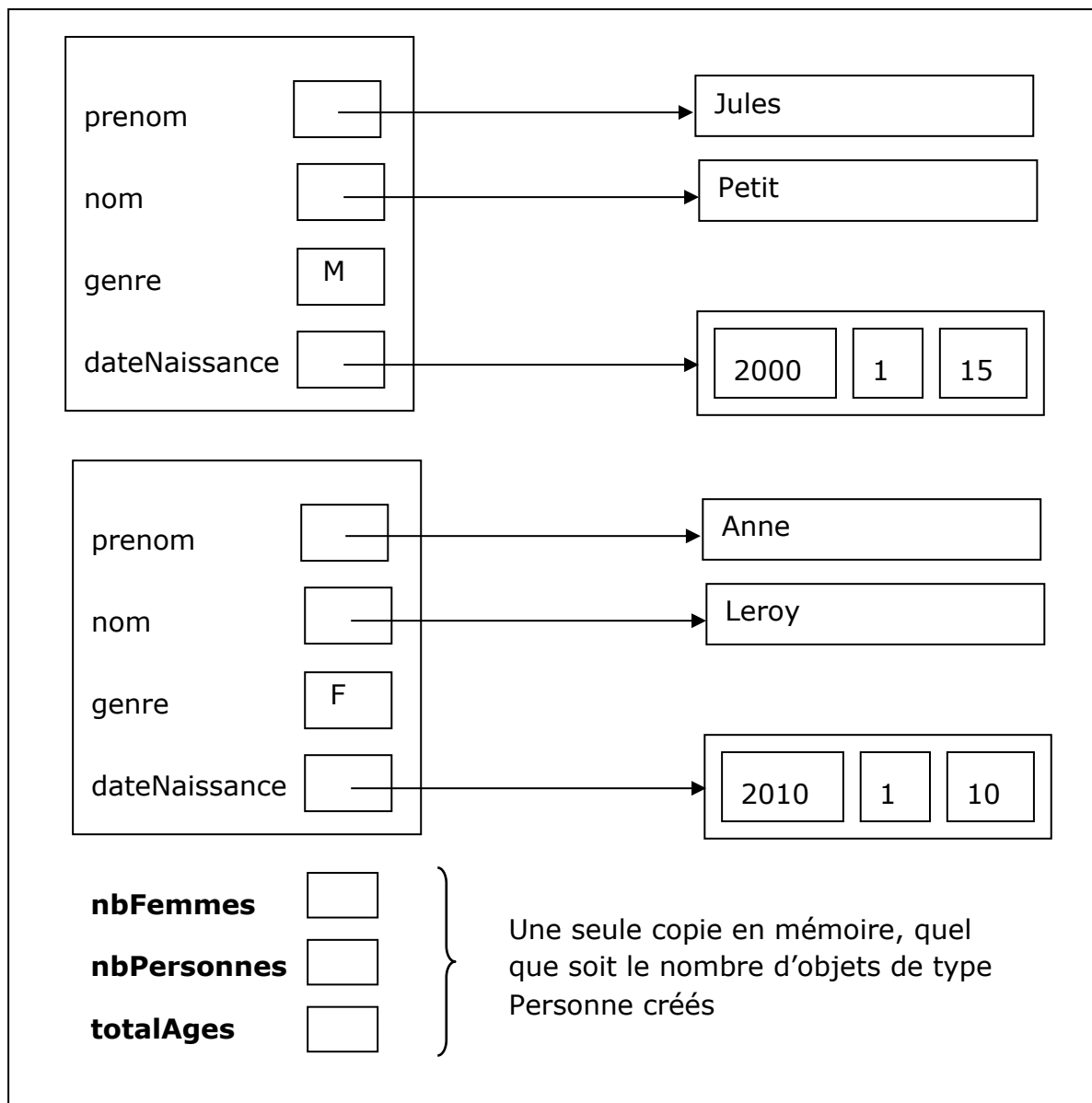
La déclaration des variables de classe commence donc par les mots réservés :

private static

```
public class Personne {
    private String prenom;
    private String nom;
    private char genre;
    private LocalDate dateNaissance;
    private static int nbFemmes = 0 ;
    private static int nbPersonnes = 0;
    private static int totalAges = 0;
    ...
}
```

```
public class Principal {
    public static void main(String [] args) {
        Personne jules = new Personne("Jules", "Petit", 'M', 15, 1, 2000);
        Personne anne = new Personne("Anne", "Leroy", 'F', 10, 1, 2010);
    }
}
```

En mémoire



N.B. Si aucune initialisation explicite n'est faite, les variables de classes seront initialisées par défaut (booléen = false, nombre = 0 et objet = référence null).

Où mettre à jour les variables de classe ?

C'est-à-dire où augmenter le nombre de personnes et de femmes créées et où ajouter l'âge de la personne créée au total des âges ?

Le plus adéquat est de mettre à jour ces variables à chaque fois que l'on crée un nouvel objet. Or, la méthode qui est appelée automatiquement chaque fois que l'on crée un nouvel objet, c'est le constructeur. Il suffit donc d'ajouter dans tout constructeur les instructions de mise à jour des variables de classe.

Comme il s'agit de caractéristique de classe, on y accède en la préfixant du nom de la classe : ***NomClasse.variableDeClasse***

```
public class Personne {
    private String prenom;
    private String nom;
    private char genre;
    private LocalDate dateNaissance;
    private static int nbFemmes = 0;
    private static int nbPersonnes = 0;
    private static int totalAges = 0;

    public Personne(String prenom, String nom, char genre,
        int jourNaissance, int moisNaissance, int anneeNaissance) {
        this.prenom = prenom;
        this.nom = nom;
        this.genre = genre;
        this.dateNaissance = LocalDate.of(anneeNaissance, moisNaissance,
            jourNaissance);

        Personne.nbPersonnes ++;
        if (genre == 'F') {
            Personne.nbFemmes ++;
        }
        Personne.totalAges += age();
    }

    public int age() {
        ...
    }
    ...
}
```


Notons qu'il n'est pas obligatoire de préfixer les variables de classe du nom de la classe lorsqu'on les utilise au sein de cette même classe.

8.2 Méthode de classe

Les caractéristiques de classe peuvent aussi bien être des variables (**variables de classe**) que des méthodes (**méthodes dites de classe**). On déclare ces méthodes **static** et on y fait appel via le nom de la classe :

NomClasse.methodeDeClasse(...)

Exemple

On pourrait prévoir une méthode de classe (donc déclarée **static**) qui calculerait la moyenne d'âge de tous les objets de type *Personne* créés. Cette méthode, appelée **moyenneAge**, ne prendrait aucun argument : elle se baserait tout naturellement sur les valeurs stockées dans les variables de classe **nbPersonnes** et **totalAges**.

```
public static double moyenneAge() {
    if (Personne.nbPersonnes != 0) {
        return Personne.totalAges / (double) Personne.nbPersonnes;
    }
    else {
        return 0;
    }
}
```

N.B. Les variables *totalAges* et *nbPersonnes* étant de type *int*, la division sera de type entier. Il faut donc caster en *double* une des deux variables afin d'obtenir un réel.

```
public class Principal {
    public static void main(String [] args) {
        Personne jules = new Personne("Jules", "Petit", 'M', 15, 1, 2000);
        Personne anne = new Personne("Anne", "Leroy", 'F', 10, 1, 2010);
        System.out.println(Personne.moyenneAge());
    }
}
```

N.B. Notons qu'il est cependant aussi permis syntaxiquement d'accéder à une variable de classe ou une méthode de classe via le nom d'un objet. Comme cette

façon de procéder n'est ni naturelle ni logique, nous ne l'utiliserons pas dans la suite du cours.

Exemple à éviter :

```
public class Principal {
    public static void main(String [] args) {
        Personne jules = new Personne("Jules", "Petit", 'M', 15, 1, 2000);
        System.out.println(jules.moyenneAge());
    }
}
```

8.2.1 Variable de classe et le principe de l'Information Hiding

Le principe de l'**Information Hiding** a été appliqué aux variables de classe : elles ont donc été déclarées avec la protection *private*.

Par conséquent, si le concepteur de la classe désire permettre l'accès en lecture ou en écriture aux variables de classe en dehors de la classe, il doit prévoir des **getters** et des **setters** déclarés avec la protection **public**. Comme ces méthodes permettent l'accès à des caractéristiques de classe, il s'agit de méthodes de classe : elles seront donc déclarées **static**.

La déclaration des getters et setters d'accès aux variables de classe (avec la protection **public**) commence donc par les mots réservés : **public static ...**

Exemple

```
public class Personne {
    private String prenom;
    private String nom;
    private char genre;
    private LocalDate dateNaissance;
    private static int nbFemmes = 0;
    private static int nbPersonnes = 0;
    private static int totalAges = 0;

    ... // Constructeur, getters, setters et autres méthodes

    public static int getNbFemmes() {
        return Personne.nbFemmes;
    }

    public static void setNbFemmes(int nbFemmes) {
        Personne.nbFemmes = nbFemmes;
    }
}
```

```

public static int getNbPersonnes() {
    return Personne.nbPersonnes;
}
public static void setNbPersonnes(int nbPersonnes) {
    Personne.nbPersonnes = nbPersonnes;
}
public static int getTotalAges() {
    return Personne.totalAges;
}
public static void setTotalAges(int totalAges) {
    Personne.totalAges = totalAges;
}
public static double moyenneAge() {
    if (Personne.nbPersonnes != 0) {
        return Personne.totalAges / (double) Personne.nbPersonnes;
    }
    else {
        return 0;
    }
}
}

```

```

public class Principal {
    public static void main(String [] args) {
        Personne jules = new Personne("Jules", "Petit", 'M', 15, 1, 2000);
        Personne anne = new Personne("Anne", "Leroy", 'F', 10, 1, 2010);
        System.out.println("Nombre de personnes : "
            + Personne.getNbPersonnes());
    }
}

```

⇒ Affiche à l'écran :

Nombre de personnes : 2

```

System.out.println("Nombre de femmes : "
    + Personne.getNbFemmes());

```

⇒ Affiche à l'écran :

Nombre de femmes : 1

8.2.2 Gestion des dates

Il existe des méthodes de classes dans des classes existantes.

Reprenons à titre d'exemple la classe *LocalDate*. Cette classe nécessite un import : *import java.time.LocalDate*.

Dans la classe *LocalDate*, il existe une méthode pour créer un objet de type date à partir d'une année, un mois et un jour dans le mois. Cette méthode est la méthode *of* dont la définition dans la documentation de la classe *LocalDate* est :

```
|| public static LocalDate of(int year,int month,int dayOfMonth)
```

Cette méthode déclarée *static* est donc une méthode de classe. Il **faut préfixer son appel du nom de la classe** *LocalDate*.

Par exemple, si on veut créer un objet de type *LocalDate* correspondant à la date de parution d'un livre paru le 15 janvier 2000, il faut écrire :

```
|| LocalDate dateParution = LocalDate.of(2000,1,15);
```

De même, il existe dans la classe *LocalDate* une méthode de classe permettant de créer une date correspondant à la date système (date courante). Cette méthode est la méthode *now*. La définition de cette méthode dans la documentation est :

```
|| public static LocalDate now()
```

Exemple :

```
|| LocalDate aujourd'hui = LocalDate.now();
```

D'autre part, la classe *LocalDate* contient des méthodes permettant de retrouver l'année, le numéro du mois ou le numéro du jour dans le mois d'un objet de type *LocalDate*. Ces méthodes ne sont pas des méthodes de classe ; on doit donc les appeler sur des objets. Les définitions de ces méthodes sont respectivement :

```
|| public int getYear()  
|| public int getMonthValue()  
|| public int getDayOfMonth()
```

Exemple :

```
|| LocalDate dateParution = LocalDate.of(2000,1,15);  
|| System.out.println("Année de parution : " + dateParution.getYear());  
|| System.out.println("Mois de parution : " + dateParution.getMonthValue());  
|| System.out.println("Jour de parution : " + dateParution.getDayOfMonth());
```

Une autre classe intéressante dans la gestion des dates est la classe *Period* qui permet de gérer des intervalles entre deux dates. Cette classe nécessite un import : *import java.time.Period*.

Dans la classe *Period*, il existe une méthode qui calcule la période entre deux dates données en argument (de type *LocalDate*) et qui retourne un objet de type *Period*. Cette méthode est la méthode *between* dont la définition dans la documentation de la classe *Period* est :

```
public static Period between (LocalDate startDateInclusive,  
                             LocalDate endDateExclusive)
```

Cette méthode déclarée *static* est donc une méthode de classe. Il faut préfixer son appel du nom de la classe *Period*.

Par exemple, si on veut retrouver la période écoulée entre la date de parution d'un livre et la date système, il faut écrire :

```
LocalDate dateParution = LocalDate.of(2000,1,15);  
LocalDate aujourd'hui = LocalDate.now();  
Period periodeParution = Period.between(dateParution, aujourd'hui);
```

À noter qu'il existe dans la classe *Period*, comme dans la classe *LocalDate*, une méthode qui permet de retrouver le nombre d'années d'un objet de type *Period*. La définition de cette méthode est :

```
public int getYears()
```

Par exemple, si on veut retrouver le nombre d'années écoulées depuis la parution d'un livre, il faut écrire :

```
LocalDate dateParution = LocalDate.of(2000,1,15);  
LocalDate aujourd'hui = LocalDate.now();  
Period periodeParution = Period.between(dateParution, aujourd'hui);  
int nbAnneesParution = periodeParution.getYears();
```

8.3 Bloc d'instructions déclaré *static*

Il est possible de déclarer un bloc d'instructions *static*. Ce qui signifie que ces instructions ne seront exécutées qu'une seule fois lors de la première utilisation de la classe.

On peut donc utiliser un tel bloc par exemple pour initialiser des variables de classe.

Exemple

```
public class Personne {  
    private String prenom;  
    private String nom;  
    private char genre;  
    private LocalDate dateNaissance;  
    private static int nbFemmes;  
    private static int nbPersonnes;  
    private static int totalAges;  
  
    static {  
        Personne.nbFemmes = 0;  
        Personne.nbPersonnes = 0;  
        Personne.totalAges = 0;  
    }  
    ... // Constructeur, getters, setters et autres méthodes  
}
```

Conclusion

- ① Chaque objet d'une classe possède en mémoire un espace qui lui est propre pour ses **variables d'instance**. Il s'agit de caractéristiques de chaque objet de la classe.
- ② Par opposition aux variables d'instance, les variables dites **variables de classe** sont des caractéristiques de la classe. Il y a **un seul espace en mémoire** alloué à une variable de classe, et ce, quel que soit le nombre d'objets de la classe qui ont été créés : 0, 1 ou plusieurs.
- ③ La déclaration d'une variable de classe contient le mot réservé : **static**.
- ④ On accède aux variables de classe via le nom de la classe :
NomClasse.variableDeClasse
- ⑤ De même qu'il existe des caractéristiques de classe qui sont des variables de classe, il existe des **méthodes de classe**. La signature d'une méthode de classe contient le mot réservé **static**.
- ⑥ On appelle une méthode de classe via le nom de la classe :
NomClasse.methodeDeClasse(...)
- ⑦ Le principe de l'**Information Hiding** est applicable aux variables de classe : on les déclare donc avec la protection **private**. La déclaration des variables de classe débute donc par **private static**. Si le concepteur de la classe désire permettre l'accès en lecture ou en écriture aux variables de classe en dehors du package, il faut prévoir des **getters** et **setters** déclarés avec la protection **public**. Comme il s'agit de méthodes permettant l'accès à des caractéristiques de classe, il s'agit de **méthodes de classe**. Elles sont donc déclarées **static**. La déclaration d'un tel getter ou setter débute donc par **public static**.
- ⑧ Les variables de classes peuvent être initialisées soit implicitement (valeurs par défaut), soit explicitement dans leur déclaration, soit via un bloc d'instructions déclaré **static** (qui ne sera exécuté qu'une seule fois à la première utilisation de la classe).

8.4 Classes *String* et classes de type *Wrapper*

Comme nous l'avons déjà vu, la classe *String* peut être vue comme une "boîte à outils" de manipulation de chaînes de caractères, comme par exemple :

- retourner la longueur d'une chaîne de caractères (*length*) ;
- extraire un caractère (*charAt*) ;
- extraire une sous-chaîne de caractères (*substring*) ;
- comparer deux chaînes de caractères (*compareTo*) ;
- vérifier le contenu d'une chaîne de caractères (*equals*) ;
- transformer une chaîne de caractères en majuscules ou minuscules (*toUpperCase* ou *toLowerCase*).

Exemples

```
String texte = "Bonjour";
String message = "Bienvenue";
int longueurTexte = texte.length();
char troisiemeLettreTexte = texte.charAt(2);    // Premier caractère ⇒ 0
int resultatComparaison = texte.compareTo(message);    // 0 si égaux
if (texte.equals("Au revoir")) {
    ...
}
```

Le même type de "boîte à outils" existe pour chacun des types primitifs. On appelle ces classes des *Wrappers* en anglais (conteneurs en français).

- Type primitif **byte** ⇒ classe **Byte**
- Type primitif **short** ⇒ classe **Short**
- Type primitif **int** ⇒ classe **Integer**
- Type primitif **long** ⇒ classe **Long**
- Type primitif **float** ⇒ classe **Float**
- Type primitif **double** ⇒ classe **Double**
- Type primitif **char** ⇒ classe **Character**
- Type primitif **boolean** ⇒ classe **Boolean**

Pour utiliser correctement ces "boîtes à outils", il faut bien analyser la déclaration des méthodes disponibles ; en effet, certaines méthodes sont déclarées *static*.

Exemples

Dans la classe **Integer** :

public int intValue()

↳ Retourne la valeur de type int (type primitif) d'un objet de type Integer

Exemple d'utilisation :

```
Integer objetEntier = new Integer(8);  
int valeurDeTypeInt = objetEntier.intValue();
```

public static int parseInt(String s) throws NumberFormatException

↳ Parse une chaîne de caractères
et essaie de retourner l'entier de type int (type primitif) correspondant

Exemples d'utilisation :

```
try {  
    int entier = Integer.parseInt("12");  
    System.out.println("valeur de l'entier : " + entier);  
}  
catch (NumberFormatException exception) {  
    System.out.println("Impossible de générer un entier à partir de  
        cette chaîne de caractères");  
}
```

⇒ Affiche : valeur de l'entier : 12

```
try {  
    int entier = Integer.parseInt("Bonjour");  
    System.out.println("valeur de l'entier : " + entier);  
}  
catch (NumberFormatException exception) {  
    System.out.println("Impossible de générer un entier à partir de  
        cette chaîne de caractères");  
}
```

⇒ Affiche :

Impossible de générer un entier à partir de cette chaîne de caractères

Le mot réservé *final*

8.5 Constante : variable déclarée *final*

Les variables d'instance classiques telles que nous les utilisons jusqu'à présent contiennent des valeurs qui peuvent être modifiées :

- Soit directement si leur protection le permet
 - Exemple : `facture.montant = 95.5`
- Soit indirectement si elles sont déclarées avec la protection **private** ; on peut alors y accéder si des **setters** déclarés avec la protection **public** sont disponibles
 - Exemple : `facture.setMontant(95.5)`

On peut cependant déclarer des variables d'instance qui, une fois initialisées (par exemple, via le constructeur) ne sont plus modifiables, et ce, quelle que soit leur protection. Il suffit de les déclarer avec le mot réservé **final**.

Puisque la valeur de telles variables ne peut plus être modifiée, il s'agit de **constantes**.

Par convention, le nom de toute constante en Java est écrit entièrement en **MAJUSCULES**.

En outre, puisque la valeur d'une constante ne peut être modifiée, il n'y a aucun problème de sécurité; on déclare donc souvent une constante avec la protection **public**.

Exemple

```
public class Emprunteur {  
    public final double MONTANT_EMPRUNT_MAXIMUM;  
    ...  
    public Emprunteur(double maximum, ...) {  
        MONTANT_EMPRUNT_MAXIMUM = maximum;  
        ...  
    }  
}
```

Une fois qu'une variable déclarée **final** a été initialisée, sa valeur ne peut plus être modifiée. Toute tentative pour modifier une telle constante provoquera une erreur à la compilation.

On accède à une constante publique comme on accède à une variable d'instance qui serait déclarée publique :

nomObjet.CONSTANTE

Exemple

```
public class Principal {
    public static void main(String [] args) {
        Emprunteur pierre = new Emprunteur(1000.0, ...);
        System.out.println("Montant maximum de l'emprunt pour Pierre : " +
            pierre.MONTANT_EMPRUNT_MAXIMUM);
        pierre.MONTANT_EMPRUNT_MAXIMUM = 2000.0;
    }
}
```

Si une constante doit être déclarée au niveau d'une classe, il s'agit alors d'une **constante de classe**. La déclaration d'une constante de classe contient donc les mots réservés ***public final static***.

On accède à une constante de classe via le nom de la classe :

NomClasse.CONSTANTE_DE_CLASSE

Exemple

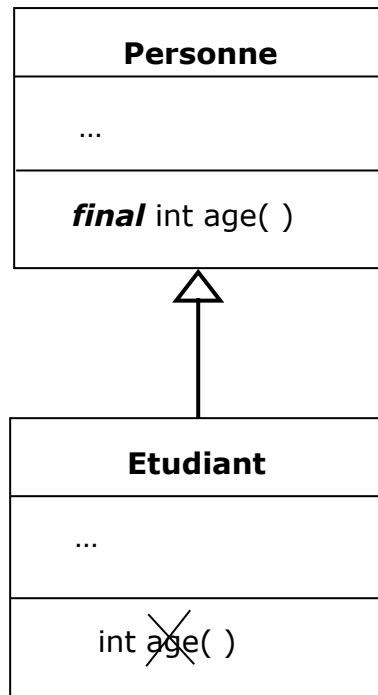
```
public class Emprunteur {
    public final double MONTANT_EMPRUNT_MAXIMUM;
    public final static int AGE_MINIMUM = 18;
    ...
}
```

```
public class Principal {
    public static void main(String[] args) {
        System.out.println(Emprunteur.AGE_MINIMUM);
    }
}
```

8.6 Méthode déclarée *final*

Une **méthode déclarée *final*** ne peut être redéfinie dans une sous-classe.

Exemple

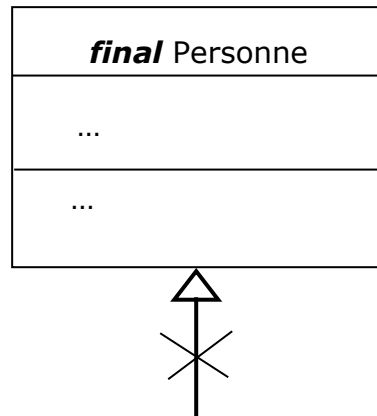


Dans l'exemple proposé ci-dessus, la méthode *age* est déclarée **final** dans la classe *Personne*. Elle ne peut donc *plus* être **redéfinie** dans *aucune* sous-classe de la classe *Personne*, et donc en aucun cas dans la classe *Etudiant*.

8.7 Classe déclarée *final*

Une **classe déclarée *final*** ne peut avoir de sous-classe. On ne peut donc pas hériter d'une classe déclarée *final*. Il s'agit de classes qui "*terminent*" leur hiérarchie d'héritage, c'est-à-dire qui se trouvent le plus bas possible dans leur hiérarchie.

Exemple



Dans l'exemple proposé ci-dessus, la classe *Personne* ne peut avoir de sous-classe.

Conclusion

① Une **variable d'instance** déclarée avec le mot réservé ***final*** ne peut plus voir sa valeur modifiée, une fois qu'elle a été initialisée. Il s'agit donc d'une **constante**. Par convention, le nom d'une constante s'écrit entièrement en **majuscules**. On accède à une variable d'instance constante via le nom d'un objet :

nomObjet.CONSTANTE

② Une **variable de classe** déclarée ***final*** est une **constante de classe**. On accède à une constante de classe via le nom de la classe :

NomClasse.CONSTANTEdeCLASSE

③ Une **méthode** déclarée ***final*** ne peut pas être redéfinie dans une sous-classe.

④ Une **classe** déclarée ***final*** ne peut pas avoir de sous-classe.

9 Classes abstraites et interfaces

9.1 Classe abstraite (*abstract*)

Une **méthode** est déclarée abstraite si elle ne contient **pas d'implémentation**, c'est-à-dire, si aucun code n'est associé à la signature de la méthode.

Une **classe** qui **contient au moins une méthode abstraite doit être déclarée abstraite**.

Une classe abstraite **ne peut avoir d'occurrences**. Autrement dit, on ne peut pas créer d'objets à partir d'une classe abstraite, et ce, même si cette classe contient un constructeur.

Une classe abstraite n'a donc d'intérêt que si on crée, à partir d'elle, des sous-classes (non abstraites), **sous-classes qui implémenteront les méthodes abstraites** dont elles auront hérité. On pourra alors créer des occurrences de ces sous-classes, sur lesquelles on pourra appeler n'importe quelle méthode, puisque toutes auront une implémentation.

Une sous-classe peut faire appel au constructeur de sa super-classe abstraite (via *super (...)*).

Une sous-classe d'une classe abstraite qui ne fournirait pas d'implémentation pour chacune des méthodes abstraites héritées doit elle-même être déclarée abstraite. On ne peut alors créer des occurrences d'une telle sous-classe.

Une **méthode** est déclarée **abstraite** si elle est contenue le mot réservé ***abstract*** dans sa déclaration. Une méthode abstraite ne contient aucune implémentation, aucun code.

La déclaration d'une méthode abstraite se termine par **;**.

Attention, terminer la déclaration d'une méthode par **{ }** n'en fait pas une méthode abstraite, mais une méthode à laquelle correspond une implémentation, même s'il s'agit d'une implémentation vide.

Déclaration **correcte** d'une méthode abstraite :

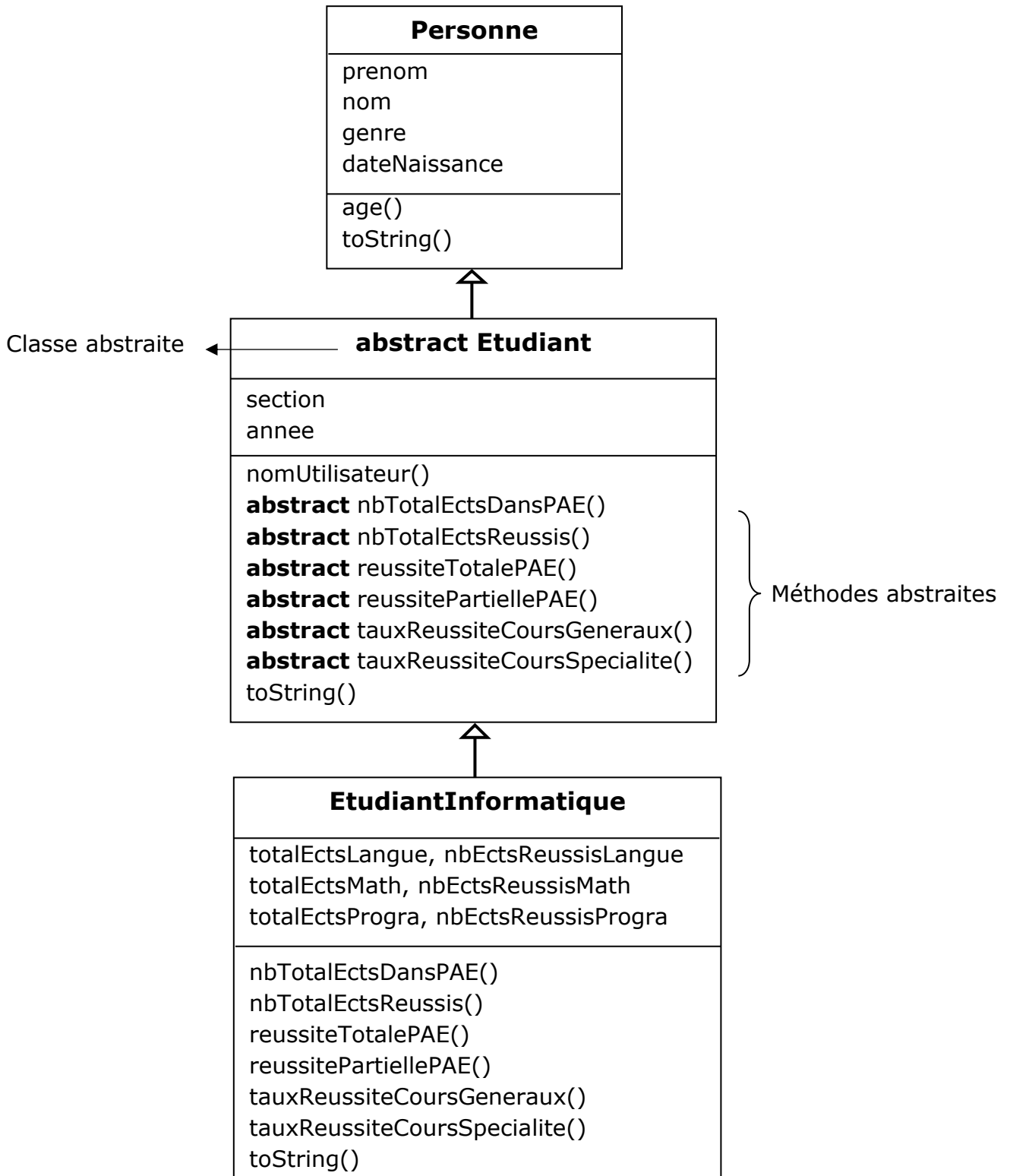
abstract *typeRetour methodeX(...)* **;**

Déclaration **incorrecte** d'une méthode abstraite :

typeRetour methodeX(...) **{ }**

Une *classe* abstraite doit contenir le mot réservé ***abstract*** dans sa déclaration.

Exemple 1



Dans l'exemple, la classe *Etudiant* est abstraite car elle contient 6 méthodes qui sont abstraites, à savoir, les méthodes *nbTotalEctsDansPAE*, *nbTotalEctsReussis*, *reussiteTotalePAE*, *reussitePartiellePAE*, *tauxReussiteCoursGeneraux* et *tauxReussiteCoursSpecialite*. En effet, il est impossible de leur donner une implémentation. Il faudrait, pour ce faire, connaître le PAE et les nombres de Ects réussis par l'étudiant pour calculer sa réussite. Or, il n'y a aucune variable d'instance dans la classe *Etudiant* reprenant ces informations. Par contre, la classe *EtudiantInformatique*, qui est une sous-classe de la classe *Etudiant*, contient les variables d'instance *totalEctsLangue*, *nbEctsReussisLangue*, *totalEctsMath*, *nbEctsReussisMath*, *totalEctsProgra* et *nbEctsReussisProgra*. Ce qui permet de donner une implémentation aux méthodes *nbTotalEctsDansPAE*, *nbTotalEctsReussis*, *reussiteTotalePAE*, *reussitePartiellePAE*, *tauxReussiteCoursGeneraux* et *tauxReussiteCoursSpecialite*. La classe *EtudiantInformatique* étant une sous-classe d'*Etudiant*, elle hérite de ces méthodes. Elle peut donc les **implémenter**. Puisque ces 6 méthodes abstraites héritées de la classe *Etudiant* contiennent une implémentation, la classe *EtudiantInformatique* n'est pas une classe abstraite. **On peut donc créer des occurrences de type *EtudiantInformatique*, alors qu'on ne peut pas créer d'occurrences de la classe abstraite *Etudiant*.**

Cependant, même si on ne peut créer des occurrences de la classe *Etudiant*, **on peut prévoir un constructeur** dans cette classe. Ce constructeur peut être appelé par le constructeur d'une sous-classe. C'est le cas du constructeur de la classe *EtudiantInformatique* qui appelle le constructeur de la classe *Etudiant* via l'instruction *super (...)*.

```
public class Personne {
    private String prenom;
    private String nom;
    private char genre;
    private LocalDate dateNaissance;
    public Personne(String prenom, String nom, char genre,
                    int jourNaissance, int moisNaissance, int anneeNaissance) {
        ...
    }
    ...
}
```



```

public abstract class Etudiant extends Personne {
    private String section;
    private int bloc;
    public Etudiant(String prenom, String nom, char genre,
                    int jourNaissance, int moisNaissance, int anneeNaissance,
                    String section, int bloc) {
        super(prenom, nom, genre, jourNaissance, moisNaissance,
              anneeNaissance);
        this.section = section;
        this.bloc = bloc;
    }
    ...    // Getters et setters
    public String nomUtilisateur() {
        return getNom() + getPrenom() + section.substring(0,2) + bloc;
    }
    public abstract int nbTotalEctsDansPAE();
    public abstract int nbTotalEctsReussis();
    public abstract boolean reussiteTotalePAE();
    public abstract boolean reussitePartiellePAE();
    public abstract double tauxReussiteCoursGeneraux();
    public abstract double tauxReussiteCoursSpecialite();

    @Override
    public String toString() {
        return "L'étudiant " + nomUtilisateur() + " est inscrit au bloc "
              + bloc + " en " + section;
    }
}

```

Constructeur prévu même si on ne peut pas créer d'occurrence de la classe *Etudiant*

Méthodes **abstraites** :
Pas d'implémentation :
Déclaration terminée par ;

```

public class EtudiantInformatique extends Etudiant {
    private int totalEctsLangue;
    private int nbEctsReussisLangue;
    private int totalEctsMath;
    private int nbEctsReussisMath;
    private int totalEctsProgra;
    private int nbEctsReussisProgra;
}

```

Appel au
constructeur de
la super-classe
abstraite

```
public EtudiantInformatique(String prenom, String nom, char genre,
    int jourNaissance, int moisNaissance, int anneeNaissance,
    String section, int bloc,
    int totalEctsLangue, int nbEctsReussisLangue, int totalEctsMath,
    int nbEctsReussisMath, int totalEctsProgra, int nbEctsReussisProgra) {
    super(prenom, nom, genre, jourNaissance, moisNaissance,
        anneeNaissance, section, bloc);
    this.totalEctsLangue = totalEctsLangue;
    this.nbEctsReussisLangue = nbEctsReussisLangue;
    this.totalEctsMath = totalEctsMath;
    this.nbEctsReussisMath = nbEctsReussisMath;
    this.totalEctsProgra = totalEctsProgra;
    this.nbEctsReussisProgra = nbEctsReussisProgra;
}
... // Public getters et setters
public int nbTotalEctsDansPAE() {
    return totalEctsLangue + totalEctsMath + totalEctsProgra;
}
public int nbTotalEctsReussis() {
    return nbEctsReussisLangue + nbEctsReussisMath + nbEctsReussisProgra;
}
public boolean reussiteTotalePAE() {
    return this.nbTotalEctsReussis() == this.nbTotalEctsDansPAE();
}
public boolean reussitePartiellePAE() {
    return !this.reussiteTotalePAE() && getBloc() == 1
        && this.nbTotalEctsReussis() >= 45;
}
public double tauxReussiteCoursGeneraux() {
    return ((nbEctsReussisLangue + nbEctsReussisMath) /
        (double) (totalEctsLangue + totalEctsMath)) * 100;
}
public double tauxReussiteCoursSpecialite() {
    return (nbEctsReussisProgra / (double) totalEctsProgra) * 100;
}
@Override
public String toString() {
    ...
}
}
```

```
public class Principal {
    public static void main(String [] args) {
        Etudiant line = new Etudiant("Line", "Patri", 'F', 26, 12, 2000, "Droit", 3);
    }
}
```

- ↳ Cette instruction provoque une erreur à la compilation, car la classe *Etudiant* est abstraite, et ce, même si le constructeur existe

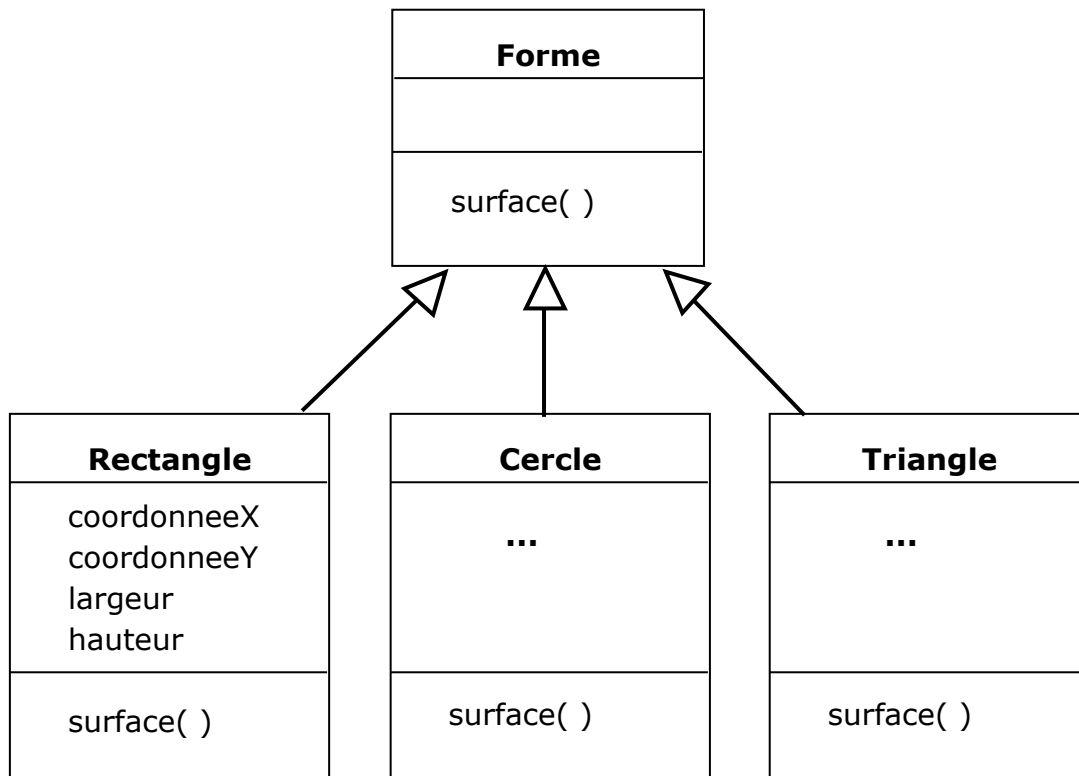
```
EtudiantInformatique marc =
    new EtudiantInformatique("Marc", "Inf1", 'M', 26, 11, 2000,
        "Technologie de l'informatique", 2, 11, 11, 9, 9, 40, 20);
```

- ↳ Cette instruction ne provoque pas d'erreur à la compilation, même si ce constructeur fait appel au constructeur de la super-classe abstraite

Exemple 2

On pourrait imaginer qu'un programmeur 1 mette au point un programme qui gère et manipule des formes, à condition de disposer de la méthode qui retourne la surface de ces formes.

Le programmeur 1 crée alors une classe abstraite *Forme* qui contient la méthode abstraite *surface* et signale à tout programmeur 2 qui désire réutiliser (bénéficier de son) programme qu'il suffit de créer des sous-classes de la classe abstraite *Forme* qui redéfinissent la méthode abstraite *surface*.



```
public abstract class Forme {  
    public abstract int surface();  
}
```

```
public class Rectangle extends Forme {  
    private int coordonneeX;  
    private int coordonneeY;  
    private int largeur;  
    private int hauteur;  
  
    public Rectangle(int coordonneeX, int coordonneeY, int largeur,  
                    int hauteur) {  
        this.coordonneeX = coordonneeX;  
        this.coordonneeY = coordonneeY;  
        this.largeur = largeur;  
        this.hauteur = hauteur;  
    }  
  
    public int surface() {  
        return largeur * hauteur;  
    }  
}
```

Conclusion

① Une **méthode** est **abstraite** si elle ne contient **pas d'implémentation**. La déclaration d'une méthode abstraite doit contenir le mot réservé ***abstract*** et se terminer par **;**

abstract typeRetour methodeX (...) **;**

② Une **classe** qui contient **au moins une méthode abstraite** doit être déclarée **abstraite**. Une classe abstraite doit contenir le mot réservé ***abstract*** dans sa déclaration.

③ On ne peut **pas créer d'occurrence d'une classe abstraite**, même si celle-ci contient un constructeur.

④ On peut créer des **sous-classes** d'une classe abstraite. Une sous-classe d'une classe abstraite doit, si elle n'est pas elle-même déclarée abstraite, implémenter **toutes** les méthodes abstraites dont elle hérite.

9.2 Interface

Une interface peut être assimilée à une sorte de **classe entièrement abstraite** : **toutes ses méthodes sont abstraites**. Une interface ne peut contenir de variables d'instance ni de constructeur. Elle peut cependant contenir des **constantes** (*final static*). **Toutes les méthodes** d'une interface sont non seulement implicitement abstraites mais également implicitement déclarées avec la protection **public**. Une interface contient donc, outre des constantes, un ensemble de déclarations de méthodes.

La déclaration d'une interface commence par le mot réservé *interface* au lieu du mot réservé *class* :

```
interface NomInterface
{ ...
}
```

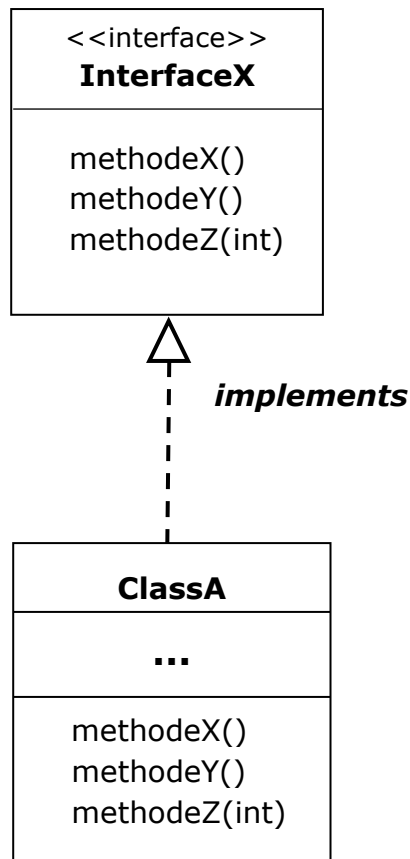
Puisque toutes les méthodes d'une interface sont implicitement abstraites et publiques, les **mots réservés *abstract* et *public* sont facultatifs** dans la déclaration des méthodes d'une interface.

Une interface n'a d'utilité que si des **classes s'engagent à redéfinir les méthodes de l'interface** et donc à leur donner une implémentation. On peut donc voir ce mécanisme comme une sorte de contrat soumis par l'interface à toute classe qui souhaite l'implémenter.

Une classe qui s'engage à implémenter les méthodes dont les déclarations sont reprises dans une interface contient la clause ***implements*** dans sa déclaration suivie du nom de l'interface :

```
class NomClasse implements NomInterface
{...
}
```

Notons que si une classe contient la clause *implements* dans sa déclaration, elle doit normalement implémenter **toutes les méthodes reprises dans l'interface**. Si au moins une de ces méthodes n'est pas implémentée, il s'agit d'une erreur détectée à la compilation, à moins qu'on ne décide volontairement d'en faire une classe **abstraite**, auquel cas on doit la déclarer *abstract*.



```

public interface InterfaceX {
    void methodeX();
    int methodeY();
    void methodeZ(int a);
}
  
```

Toutes les méthodes sont implicitement **public** et **abstract**

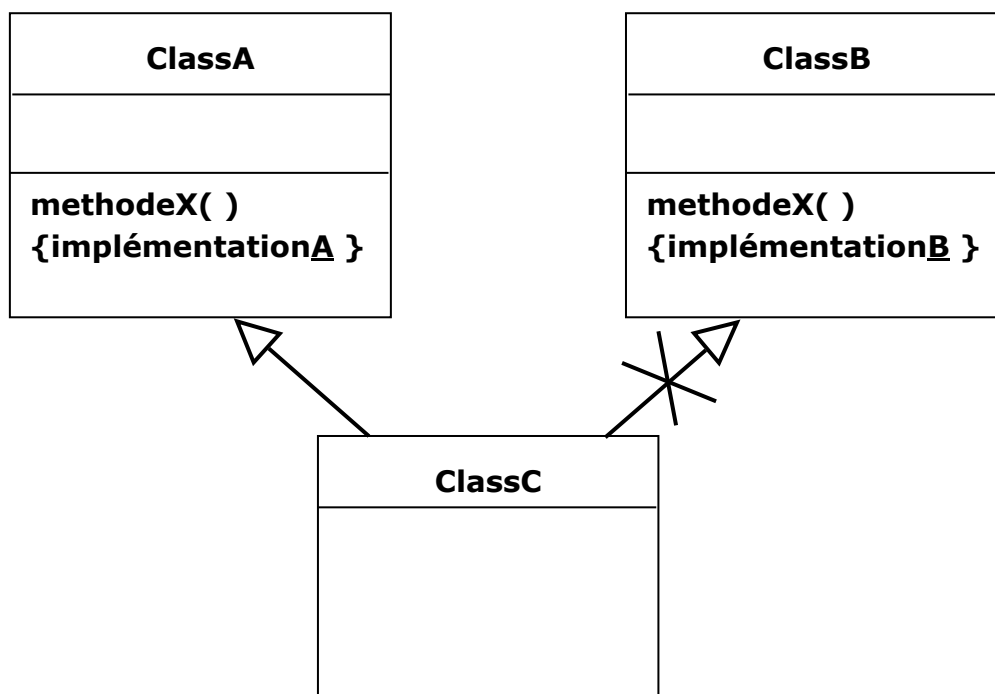
```

public class ClassA implements InterfaceX {
    public void methodeX() {
        ...
    }
    public int methodeY() {
        ...
    }
    public void methodeZ(int a) {
        ...
    }
}
  
```

Code correspondant à l'implémentation des méthodes de l'interface

N.B. Comme les méthodes des interfaces sont implicitement déclarées publiques (*public*), toute classe qui implémente une interface en redéfinissant ses méthodes doit les déclarer avec la protection **public**. En effet, une méthode héritée ne peut pas être redéfinie avec une protection moins permissive.

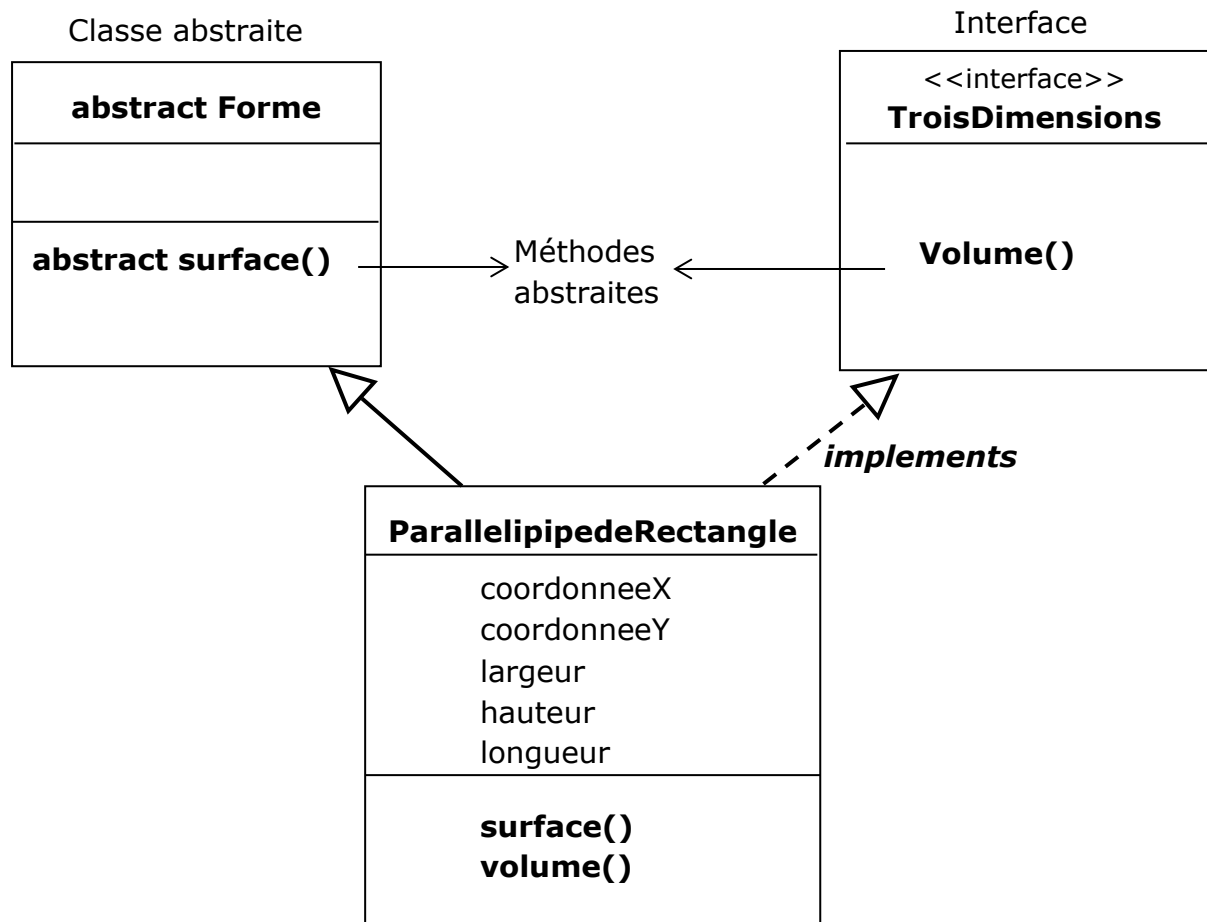
Il n'y a **pas d'héritage multiple permis en Java**. Une classe ne peut hériter de plusieurs super-classes à la fois. Cette restriction se justifie par le **risque de conflit** qu'un héritage multiple pourrait engendrer. En effet, que faire si deux méthodes avec la même signature se trouvent dans deux des super-classes ? Laquelle des deux implémentations exécuter lorsque cette méthode est appelée sur une occurrence de la sous-classe ?



Dans cette hiérarchie, sachant que la classe *ClassC* est une sous-classe à la fois de la classe *ClassA* et de la classe *ClassB*, quel code exécuter si la méthode *methodeX* est appelée sur une occurrence de la classe *ClassC* : *implémentationA* ou *implémentationB* ?

Par contre, **une classe Java peut implémenter plus d'une interface**. De plus, une classe Java peut à la fois être **sous-classe** d'une super-classe tout en implémentant **plusieurs interfaces**.

Exemple



```
public abstract class Forme {  
    public abstract int surface();  
}
```

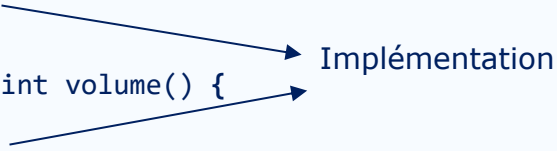
```
public interface TroisDimensions {  
    int volume();  
}
```

```
public class ParallelipedeRectangle extends Forme implements TroisDimensions {  
    private int coordonneeX;  
    private int coordonneeY;  
    private int largeur;
```

```

private int hauteur;
private int longueur;
...
public int surface() {
    ...
}
public int volume() {
    ...
}
}

```



Notons que dans l'exemple précédent, on aurait pu déclarer *Forme* comme une interface, puisque cette classe abstraite ne contient qu'une unique méthode qui est elle-même abstraite.

```

public interface Forme {
    int surface();
}

```

```

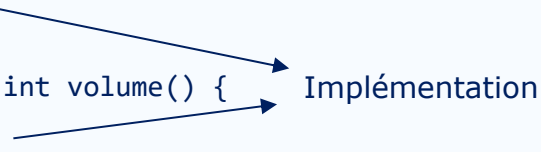
public interface TroisDimensions {
    int volume();
}

```

```

public class ParallelipipedeRectangle implements Forme, TroisDimensions {
    private int coordonneeX;
    private int coordonneeY;
    private int largeur;
    private int hauteur;
    private int longueur;
    ...
    public int surface() {
        ...
    }
    public int volume() {
        ...
    }
}

```



Conclusion

① La déclaration d'une interface commence par le mot réservé **interface**. Une interface est un ensemble de déclaration de **constantes** et/ou de **méthodes abstraites**.

② Toutes les méthodes d'une interface sont implicitement déclarées **public** et **abstract**.

③ Une classe qui implémente une interface s'engage à fournir une implémentation pour **toutes** les méthodes de l'interface. Une telle classe contient dans sa déclaration la clause **implements** suivi du nom de l'interface :

class NomClasse implements NomInterface

④ Toute classe qui implémente une interface doit implémenter **chacune** des méthodes de l'interface en les déclarant avec la protection **public**.

⑤ Une classe qui contient une clause *implements* dans sa déclaration mais **n'implémente pas toutes les méthodes reprises dans l'interface**, n'a de sens que si l'on veut en faire une **classe abstraite** (doit alors contenir le mot *abstract* dans sa déclaration).

⑥ Il n'y a **pas d'héritage multiple** permis en Java. Par contre, une classe peut implémenter **plusieurs interfaces**. Une classe peut aussi être sous-classe d'une super-classe tout en implémentant une ou plusieurs interface(s).

10 Les tableaux

Un tableau est une suite d'éléments de même type.

Dans un premier temps nous aborderons les tableaux contenant des éléments de type primitif. Ensuite, nous verrons les tableaux contenant des objets.

10.1 Tableaux d'éléments de type primitif

Pour rappel, les types primitifs sont : **byte**, **int**, **long**, **float**, **double**, **char** et **boolean**.

Pour utiliser un tableau, le programmeur doit :

- Déclarer un tableau en précisant le **type** de ses éléments et la **longueur** du tableau (c'est-à-dire le nombre maximum d'éléments que pourra contenir ce tableau) ;
- Prévoir des **méthodes d'accès** qui ne sont pas les getters/setters classiques, mais des méthodes d'accès adaptées aux tableaux :
 - **Getters**
Exemples : pour lire le $i^{\text{ème}}$ élément, pour connaître le nombre réel d'éléments dans le tableau ...
 - **Setters**
Exemples : pour modifier le $i^{\text{ème}}$ élément, pour ajouter un nouvel élément en fin de tableau ...

Déclaration de tableau

Pour préciser qu'une variable est un tableau, on place les caractères **[]** à côté du **type** ou du nom de la variable

Exemple :

```
int [] cotesInterro; // ou int cotesInterro[]
```

⇒ Déclare un tableau appelé *cotesInterro* qui pourra contenir des éléments de type entier.

D'autre part, une simple déclaration de variable ne suffit pas. Il faut encore réserver de la place en mémoire pour ce tableau en précisant, entre autres, sa longueur. Pour ce faire, on utilise l'instruction : **new type [longueur]**.

Exemples

```
cotesInterro = new int [10];
```

⇒ Réserve la place en mémoire pour 10 éléments de type entier qui seront **tous initialisés à 0.**

```
int longueur = 5;  
cotesInterro = new int[longueur];
```

⇒ Réserve la place en mémoire pour 5 éléments de type entier qui seront **tous initialisés à 0.**

Accès aux éléments du tableau

Chaque élément du tableau possède un indice. Le **premier** élément du tableau a l'**indice 0**.

Pour accéder à un élément du tableau, on utilise la notation :

nomTableau [indice]

Exemples

```
cotesInterro[0]          ⇒ Désigne le premier élément du tableau  
cotesInterro[9]         ⇒ Désigne le 10ème élément du tableau
```

Exemple de classe possédant une variable d'instance de type tableau

Soit la classe *EvaluationContinue* qui enregistre les résultats obtenus aux interrogations lors d'évaluations continues dans une activité d'apprentissage pour un étudiant donné. Les interrogations sont cotées sur 20. Partons du principe que l'on n'organisera pas plus de 10 interrogations par évaluation continue. Un tableau de type **int** et de longueur **10** est donc prévu comme variable d'instance dans la classe *EvaluationContinue*. Ce tableau est appelé *cotesInterro*. Le nombre maximum d'éléments dans le tableau est mémorisé via une constante. La déclaration et l'initialisation de ce tableau est donc :

```
public static final int NOMBRE_MAXIMUM_INTERROS = 10;  
...  
int [] cotesInterro;  
...  
cotesInterro = new int[NOMBRE_MAXIMUM_INTERROS];
```

Soient, par exemple, trois cotes d'interrogation stockées dans ce tableau :

Tableau **coteInterros**

18	14	16	0	0	0	0	0	0	0
----	----	----	---	---	---	---	---	---	---



Nombre d'interros dans le tableau : 3

Pour rappel, toute variable d'instance doit être déclarée *private*. Le tableau est donc déclaré *private*.

Cela nécessite la création de certains getters et setters adaptés aux tableaux.

- **Getters**

getInterro(i) : pour lire la i^{ème} interro ;

nbInterros() : pour compter le nombre réel de cotes d'interro dans le tableau ;

- **Setters**

setInterro(cote,i) : pour modifier la cote de la i^{ème} interro ;

ajouterInterro(cote) : pour ajouter une nouvelle interro en fin de tableau (seulement s'il y a encore de la place dans le tableau !!!).

EvaluationContinue
prenomNomEtudiant : String libelleActiviteApprentissage : String coteInterros : int []
getInterro (int) : int nbInterros () : int setInterro (int, int) : void ajouterInterro (int) : void

```

public class EvaluationContinue {
    public static final int NOMBRE_MAXIMUM_INTERROS = 10;
    private String prenomNomEtudiant;
    private String libelleActiviteApprentissage;
    private int [] cotesInterro;

    public EvaluationContinue(String prenomNomEtudiant,
                               String libelleActiviteApprentissage) {
        this.prenomNomEtudiant = prenomNomEtudiant;
        this.libelleActiviteApprentissage = libelleActiviteApprentissage;
        // Réservation de place mémoire pour le tableau dans le constructeur
        cotesInterro = new int[NOMBRE_MAXIMUM_INTERROS];
        // Initialisation des valeurs dans le tableau
        for (int i = 0; i < NOMBRE_MAXIMUM_INTERROS; i++)
            cotesInterro[i] = -1;
    }

    ... // Getters/setters pour prenomNomEtudiant et libelleActiviteApprentissage

    // Getters liés à la gestion des interros

    public int getInterro(int position) {
        return cotesInterro[position-1];
    }

    public int nbInterros() {
        int nbInterros = 0;
        while (nbInterros < NOMBRE_MAXIMUM_INTERROS
               && cotesInterro[nbInterros] != -1)
            nbInterros++;
        return nbInterros;
    }

    // Setters liés à la gestion des interros

    public void setInterro(int resultat, int position) {
        if (position <= nbInterros()) // si l'interro existe
            cotesInterro[position-1] = resultat;
    }

    public void ajouterInterro(int resultat) {
        int nbInterros = nbInterros();      ⇐ Optimisation de code !
        if (nbInterros < NOMBRE_MAXIMUM_INTERROS)
            cotesInterro[nbInterros] = resultat;
    }
}

```

La méthode *getInterro* (*int position*) retourne la cote de l'interrogation enregistrée à une position donnée dans le tableau (cf. l'argument *position*). Notons que l'argument *position* indique la position de l'interrogation dans le tableau **en partant de 1**. Ainsi, par exemple, si l'on veut obtenir la cote de la deuxième interrogation du tableau, l'argument *position* vaut **2**, ce qui correspond à l'indice **1** dans le tableau. Il faut donc décrémenter de 1 la valeur de la variable *position* avant d'accéder au tableau.

La méthode *nbInterros* retourne le nombre réel d'interrogations dans le tableau. Il faut donc boucler sur les éléments du tableau jusqu'à arriver à la première cote d'interro égale à 0.

La méthode *setInterro* (*int resultat*, *int position*) modifie la cote de l'interrogation située à une position donnée dans le tableau (cf. l'argument *position*), à condition que la position donnée corresponde bien à un élément existant du tableau (la position donnée doit être \leq au nombre d'éléments réels dans le tableau). Rappelons que l'argument *position* indique la position de l'interrogation dans le tableau **en partant de 1** (\Rightarrow l'indice est égal à la position-1).

La méthode *ajouterInterro* (*int resultat*) ajoute une nouvelle interrogation à la première position libre dans le tableau. La première position libre dans le tableau est calculée par la méthode *nbInterros()*. En effet, s'il y a par exemple 3 interrogations dans le tableau, l'appel à la méthode *nbInterros()* retourne 3. La première cellule libre dans le tableau se trouve à la 4^{ème} position, autrement dit à l'indice 3.

Attention toutefois à optimiser le code : il faut éviter d'appeler plusieurs fois la méthode *nbInterros()* au sein de la méthode *ajouterInterro* ; il ne faut appeler qu'une seule fois la méthode *nbInterros()* et placer son résultat dans une variable locale qu'on utilisera plusieurs fois.

De plus, il faut vérifier le non-débordement du tableau : c'est-à-dire ne pas ajouter un nouvel élément dans le tableau s'il est déjà plein. Si le tableau est plein, l'élément ne sera pas ajouté.


```

public class Principal {
    public static void main(String[] args) {
        EvaluationContinue evaluation =
            new EvaluationContinue("Alan Turing","Java");

        // Ajout de trois cotes d'interrogation dans le tableau interros
        evaluation.ajouterInterro(18);                (1)
        evaluation.ajouterInterro(14);
        evaluation.ajouterInterro(16);

        // Modification du résultat de la deuxième interro
        evaluation.setInterro(17,2);

        // Boucle d'affichage des (résultats des) interros
        for (int i = 1; i <= evaluation.nbInterros(); i++ )
            System.out.println(evaluation.getInterro(i));    (2)
    }
}

```

A noter dans la boucle d'affichage, que pour afficher l'élément en position *i*, on fait appel à la méthode publique *getInterro(i)*.

En effet, écrire ***evaluation.cotesInterro[i-1]*** serait une erreur car le tableau est déclaré ***private***.

Autre erreur fréquente : utiliser [] au lieu de () et inversement.

Par exemple, dans l'instruction (1), il s'agit de parenthèses et non de crochets :

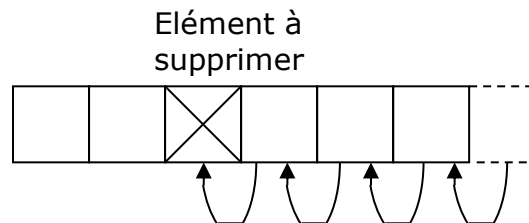
<code>evaluation.ajouterInterro(18);</code>	correct
<code>evaluation.ajouterInterro[18];</code>	incorrect

De même dans l'instruction (2) :

<code>evaluation.getInterro(i)</code>	correct
<code>evaluation.getInterro[i]</code>	incorrect

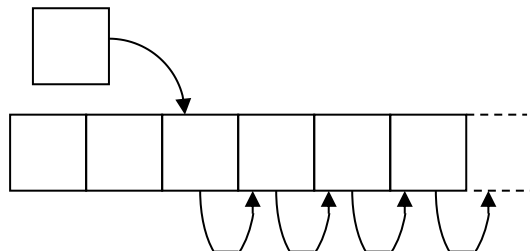
N.B. On pourrait prévoir encore d'autres méthodes d'accès au tableau, comme par exemple :

- Une méthode permettant **de supprimer une interrogation** qui se trouve à une position donnée. Dans ce cas, il faudrait déplacer vers la gauche les éléments qui se trouvent à droite de l'élément à supprimer, afin d'éviter les cellules vides dans le tableau.



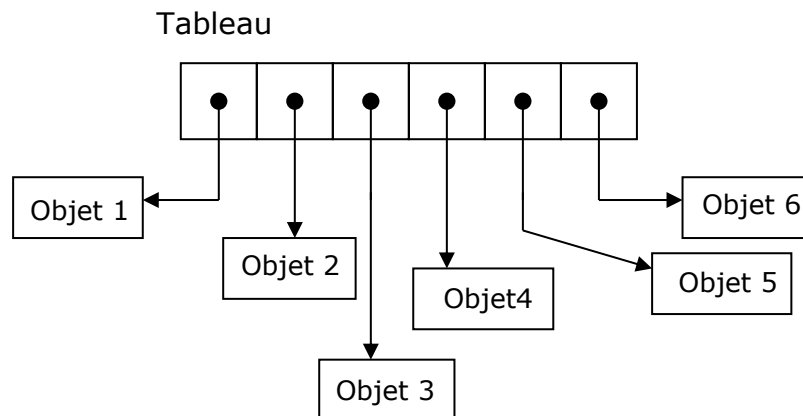
- Une méthode permettant **d'insérer une nouvelle interrogation** à une position donnée. Dans ce cas, il faudrait d'abord déplacer vers la droite certains éléments du tableau avant d'y insérer le nouvel élément, afin de n'écraser aucun élément déjà présent dans le tableau.

Nouvel élément à insérer



10.2 Tableaux d'objets

Le principe est le même que pour les tableaux d'éléments de type primitif, si ce n'est pour la déclaration du tableau et la réservation de la place en mémoire pour ce tableau. Les éléments du tableau ne sont plus de type primitif (int, char, float, ...) mais sont des **références vers des objets** d'une classe. Un tableau d'objets est un tableau dont chaque cellule est une référence vers un objet stocké ailleurs en mémoire :



Pour utiliser un tableau, le programmeur doit donc :

- Déclarer un tableau en précisant la **classe** à laquelle appartiennent ses éléments ainsi que la **longueur** du tableau ;
- Prévoir des **méthodes d'accès** adaptées aux tableaux :
 - **Getters**
Exemples : pour lire le $i^{\text{ème}}$ élément, pour connaître le nombre réel d'éléments dans le tableau ...
 - **Setters**
Exemples : pour modifier le $i^{\text{ème}}$ élément, pour ajouter un nouvel élément en fin de tableau ...

Exemple

Supposons que l'on mémorise la liste des stagiaires dont est responsable chaque professeur. Les stagiaires sont des étudiants. Partons du principe qu'un professeur ne peut être responsable de plus de 5 stagiaires. Il faut donc prévoir une variable d'instance dans la classe *Professeur* qui est un **tableau d'objets de type *Etudiant***. Appelons cette variable *stagiaires*.

L'instruction

***Etudiant* [] stagiaires = new *Etudiant* [NOMBRE_MAXIMUM_STAGIAIRES];**

a pour effet de créer en mémoire un tableau contenant autant d'éléments que renseigné par la constante ***NOMBRE_MAXIMUM_STAGIAIRES***, chacun des éléments étant une référence nulle : au départ, ce tableau ne contient aucune référence vers un étudiant. Par la suite, n'importe quelle cellule de ce tableau pourra contenir une référence vers un objet de type *Etudiant*.

Professeur
prenomNom : String stagiaires : Etudiant []
nbStagiaires() : int getStagiaire(int) : Etudiant setStagiaire(Etudiant , int) : void ajouterStagiaire(Etudiant) : void

```
public class Professeur {  
    public static final int NOMBRE_MAXIMUM_STAGIAIRES = 5;  
    private String prenomNom;  
    private Etudiant[] stagiaires;  
  
    // Constructeur  
    public Professeur(String prenomNom) {  
        this.prenomNom = prenomNom;  
        stagiaires = new Etudiant[NOMBRE_MAXIMUM_STAGIAIRES];  
    }  
  
    // Getters liés à la gestion des stagiaires  
  
    public int nbStagiaires() {  
        int nbStagiaires = 0;  
        while (nbStagiaires < NOMBRE_MAXIMUM_STAGIAIRES  
            && stagiaires[nbStagiaires] != null)  
            nbStagiaires++;  
        return nbStagiaires;  
    }  
}
```

```

    public Etudiant getStagiaire(int position) {
        return stagiaires[position-1];
    }

    // Setters liés à la gestion des stagiaires

    public void setStagiaire(Etudiant etudiant, int position) {
        if (position <= nbStagiaires())
            stagiaires[position-1] = etudiant;
    }

    public void ajouterStagiaire(Etudiant etudiant) {
        int nbStagiaires = nbStagiaires();  ⇐ Optimisation de code !
        if (nbStagiaires < NOMBRE_MAXIMUM_STAGIAIRES)
            stagiaires[nbStagiaires] = etudiant;
    }
}

```

La méthode *nbStagiaires* retourne le nombre réel de stagiaires stockés dans le tableau.

La méthode *getStagiaire(int position)* retourne la référence vers l'objet de type *Etudiant* stockée dans le tableau à une position donnée (cf. argument *position*).

La méthode *setStagiaire(Etudiant etudiant, int position)* remplace la référence de l'étudiant stockée à une position donnée par la référence de l'étudiant donné en argument, à condition qu'il y ait un étudiant stocké à cette position. *On peut discuter de l'intérêt d'une telle méthode si ce n'est dans la réorganisation du tableau.*

La méthode ***ajoutStagiaire(Etudiant etudiant)*** ajoute la référence de l'étudiant donné en argument, et ce, à la première position libre dans le tableau, à condition qu'il y ait encore de la place dans le tableau !

Le raisonnement est donc le même que pour des tableaux de types primitifs.

Il faut cependant veiller à déclarer correctement les arguments des méthodes *setStagiaire* et *ajoutStagiaire* ; elles prennent toutes deux un argument de type *Etudiant*. La méthode *getStagiaire*, quant à elle, retourne une référence vers un objet de type *Etudiant*.

```

public class Principal {
    public static void main(String [] args) {

        // Création de 3 étudiants
        Etudiant aline = new Etudiant( ... );
        Etudiant marc = new Etudiant( ... );
        Etudiant jean = new Etudiant( ... );

        // Création d'un professeur
        Professeur prof = new Professeur("Anne Petit");

        // Attribution de trois stagiaires à prof : les étudiants line, marc et jean
        prof.ajouterStagiaire(aline);
        prof.ajouterStagiaire(marc);
        prof.ajouterStagiaire(jean);

        // Boucle sur tous les stagiaires de prof
        for (int i = 1; i <= prof.nbStagiaires(); i++)
            System.out.println (prof.getStagiaire(i)); // Appel à toString d'Etudiant
    }
}

```

Notons au passage que la déclaration de la méthode *main* contient un tableau d'objets :

```

| public static void main(String [] args)

```

L'argument *args* est un tableau de *String*.