



Haute Ecole de Namur - Liège - Luxembourg
Département économique
Implantation IESN



Bachelier en Informatique de Gestion
1^{re} année

Langage Java

Syllabus

Françoise Dubisy

Introduction : La programmation orientée objet comparée à la programmation classique

Java est un langage orienté objet. On notera parfois O.O. pour Orienté Objets.
Comme le langage C sur lequel il est basé, le langage Java est « case sensitive » : les majuscules ont de l'importance. Par exemple, varA et vara sont interprétés différemment.

1. Objectifs du cours

L'objectif est d'arriver à écrire des applications à interface utilisateur graphique (fenêtre, bouton, menu,...) capables d'accéder à des bases de données relationnelles. Le langage SQL, langage d'interrogation et de manipulation de bases de données relationnelles, sera abordé également. L'aspect internet sera abordé au travers de la notion d'applet (application java exécutable à partir d'une page HTML)

2. Historique

- ① Langage **machine (première génération)**:
un programme est une suite de 0 et de 1
- ② Langage **assembleur (deuxième génération)**:
apparition d'instructions (ex : load, store, add , ...)
- ③ Langage **de haut niveau (troisième génération)**:
 - A. Fortran (IBM): 1954-1957
 - B. Cobol (Common Business Oriented Language): 1959
 - C. Basic (université de Dartmouth): 1965
 - D. Langage C (par Ritchie aux laboratoires Bell) : 1972
 - Pascal (utilisation académique: surtout dans les universités)
 - ...
 - E. Java : 1995

3. Caractéristiques de Java

- Syntaxe proche du langage C

Cfr: if, else, for, while, switch, ...

- Exploitable dans un environnement distribué (serveurs) → aspect sécurité important

En vue de sécuriser d'avantage les programmes, *la notion de pointeur a disparu*. Impossible dorénavant d'accéder à une zone de la mémoire qui ne nous est pas allouée.

- 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 architecture 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 plate-formes (linux, windows,...), à la seule condition que la plate-forme d'accueil dispose d'un interpréteur java (machine virtuelle).

-Performances

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

- Réutilisation

Le but en programmation orientée objet 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 (cfr héritage en java).

4. Programmation classique versus programmation O.O.

Programmation classique

Un programme classique est un ensemble d'instructions combinant :

- des affectations (ex : A = 3)
- des structures de contrôle (ex : if-else, boucles while,...)
- des appels de fonctions. ex :

```
s= surface (largeur, hauteur) ;  
      ↓      ↓  
fonctions arguments  
      ↑      ↑  
m = moyenne (cours1, cours2, cours3) ;
```

Une fonction est donc appelée avec des données que sont les arguments.

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

Programmation O.O.

En programmation O.O., on voit le monde **en terme d'objets**. Exemples d'objets : 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.

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** pour fabriquer des objets d'un même type. Exemple : le moule Etudiant, le moule Véhicule,...

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 deux types d'information :

- des caractéristiques ou propriétés statiques des objets de la classe
- des fonctions (notion de *méthodes*: cfr section 1.3) que l'on pourra appeler sur tout objet de la classe.

Exemples :

- a) 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 : un objet rect1 qui contient ses propres valeurs pour largeur et hauteur.
- 3° On appelle la fonction surface sur l'objet rect1: **rect1.surface()**
- b) 1° On a besoin d'un objet étudiant → Créer la classe étudiant avec ses propriétés (pourcentage des différents cours) & fonctions (moyenne, ...)
- 2° On crée un exemplaire d'étudiant : un objet etud1 qui contient ses propres valeurs pour les pourcentages des différents cours.
- 3° On appelle la fonction moyenne sur l'objet etud1: **etud1.moyenne()**

Tableau récapitulatif des différences entre C et Java

C	Java
Un programme est un ensemble de fonctions Syntaxe : nomFonction(arguments)	Un programme est un ensemble d'objets Syntaxe : objet.nomFonction(...)
Fonction puis arguments ←	→ Objet puis fonction

En O.O., une fonction ne peut être appelée que sur un objet (ou sur une classe : cfr chapitre 4). Par conséquent, tout appel de fonction doit être préfixé par le nom d'un objet :

objet.nomFonction(...)

Chapitre 1 : Objets & classes

1.1. Objet

Au début de l'informatique, construire un programme consistait principalement à trouver les bonnes *instructions*. Ensuite, les *données* sont apparues comme aussi importantes que les instructions.

Dans le monde réel, les objets sont partout.

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

Exemples :

Objet	Etat	Comportement
un chien	nom, couleur, race, ...	aboyer, dormir, manger, ...
une voiture	modèle, marque, plaque, ...	rouler, tourner, stopper, ...
	Propriétés statiques	Fonctions (méthodes)

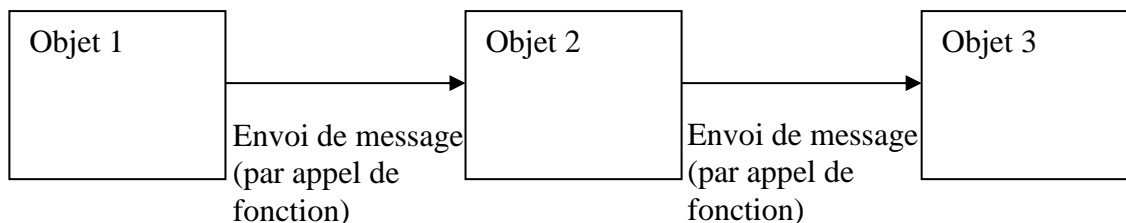
Tout objet est caractérisé par :

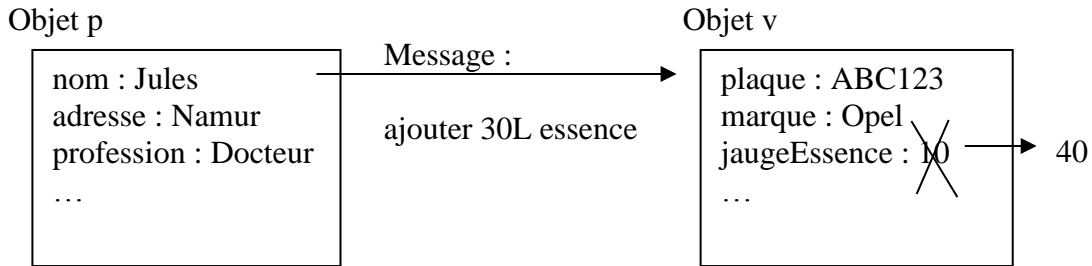
- des propriétés statiques → variables qui peuvent contenir des valeurs (1)
- des actions correspondant à son comportement : on peut invoquer ces actions sur l'objet lui-même. Il s'agit de fonctions qui peuvent lire ou modifier les variables (1) et/ou exécuter n'importe quelle autre instruction (ex : afficher des infos dans une fenêtre).

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

Dialogues d'objets entre eux : (par appels de fonctions)

Exemple :





1.2. Classe

Comme expliqué précédemment, pour disposer d'un objet, il faut d'abord disposer de son "moule", c-à-d de sa classe.

Ensuite, on créera des exemplaires à partir de cette 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 statiques sont appelées **variables d'instance** :
A tout objet (tout exemplaire) de la classe correspondra en mémoire *ses variables d'instance* avec leur propre valeur. Ex : l'objet v (de type Vehicule) aura en mémoire sa propre variable plaque avec la valeur ABC123.
- les fonctions sont appelées **méthodes**.
Il y aura *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 (v_1, v_2, \dots, v_{10}) :

- il y aura en mémoire 10 variables d'instance plaque (une pour chaque objet v_i)
- il y aura une seule copie en mémoire de chacune des 5 méthodes

Les méthodes déterminent *les comportements possibles de la classe*. On parle aussi d'interface.

1.3. 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 caractérisé par sa largeur et sa hauteur. On aimerait également pouvoir calculer la surface d'un rectangle. On voudrait en outre pouvoir élargir un rectangle (càd augmenter sa largeur).

```
class Rectangle
{
    int largeur, hauteur ;           /* variables d'instance*/

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

    void elargir ( int a)            /* méthode qui ne retourne pas de valeur */
    { largeur += a ; }
} /* termine la classe */
```

Pas d'arguments 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*).

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 maintenant que nous devons manipuler des véhicules. Un véhicule est caractérisé par une plaque minéralogique, une marque (ex : Renault), un modèle (ex : Clio) et le niveau de sa jauge à essence. On doit pouvoir rajouter de l'essence dans le véhicule (et le refléter en augmentant le niveau de sa jauge).

```
class Vehicule
{ int jaugeEssence ;                /* variable de type primitif */
  String plaque, marque, modele ;   /* variables de type référence */

  void ajouterEssence (int nbLitres) /* méthode qui ne retourne rien */
  { jaugeEssence += nbLitres ; }

  String typeVehicule( )            /* méthode qui retourne un objet de la classe String */
  { return marque ⊕ " " + modele ; }
}
```

Opérateur de concaténation

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

Types primitifs : (commencent par une minuscule)				
Entiers :	byte (1 byte)	short (2 bytes)	int (4 bytes)	long (8 bytes)
Réels :	float (4 bytes)		double (8 bytes)	
Caractère :	char (2 bytes)			
Booléen :	boolean (seules valeurs possibles: <i>true</i> ou <i>false</i>)			

Types référence (commencent par une majuscule)

Les types références sont des **classes**. Ils commencent donc par une majuscule. Ex : **String**. Cette classe permet de gérer facilement les chaînes de caractères. Plus besoin de jouer avec des tableaux de caractères comme en langage C. Une variable de type chaîne de caractères sera désormais **un objet de la classe String**. 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.

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

Exemple : int a = 4 ;
 int b = 2 ;
 c = a + b ; /* la variable c contiendra 6 */

L'**opérateur + appliqué à deux objets de type String** est l'**opérateur de concaténation** (cfr 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**.

Exemple 1 : String var1 = "Salut, " ;
 String var2 = "ca va?" ;
 String res = var1 + var2 ;
 La variable *res* contiendra "Salut, ca va?"

Exemple 2 : "Bonjour, " + "bienvenue!"
 Construit la chaîne de caractères : "Bonjour, bienvenue!"

Exemple 3 : String var = "Jules Dupond" ;
 String res = "Monsieur " + var ;
 La variable *res* contiendra "Monsieur Jules Dupond"

La méthode *typeVehicule()* construit et retourne une chaîne de caractères en concaténant la marque et le modèle du véhicule .

Exemple : Renault Clio

1.4. Constructeur

Comment créer des exemplaires de ces classes et les utiliser ?

A l'exécution, la machine virtuelle java recherche et exécute le contenu d'une méthode de signature réservée, à savoir la méthode **main** (cfr fonction main du langage C).

On entend par **signature d'une méthode** la combinaison du *nom de la méthode* et du *type des arguments*.

La déclaration de la méthode main est :

public static void main (String[] args)

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

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

```
class Principal
{
    public static void main (String[ ] args)    /* méthode main */
    {
        Rectangle r ;                          /* déclaration d'un objet de type Rectangle */
    }
}
```

Rappel: toutes les variables commencent par une *minuscule*. Par conséquent, la variable de type rectangle est appelée **r** et non **R**.

Il s'agit d'une simple déclaration de variable. Cette ligne de code (càd : *Rectangle r ;*) 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 *r* (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 le nom de 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'initialiser ses variables d'instance.

Suggestion: prévoir un constructeur avec autant d'arguments qu'il y a de variables d'instance dans la classe.

Exemple 1

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

```
class Rectangle
{
    int largeur, hauteur ;                               /* variables d'instance*/
    Rectangle (int larg, int haut)                       /* constructeur */
    { largeur = larg;
      hauteur = haut ; }
    int surface ( )                                       /* méthodes */
    { return largeur * hauteur ; }
    void elargir ( int a)
    { largeur += a ; }
}
```

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 r;                          /*déclaration d'une variable de type Rectangle */
                                                /* MAIS pas d'allocation mémoire */
        r = new Rectangle ( 3, 5 ) ;         /* appel au constructeur de la classe Rectangle*/
                                                /* allocation mémoire + initialisation */
    }
}
```

Ces deux instructions ont pour effet de créer en mémoire un objet référencé par la variable r.

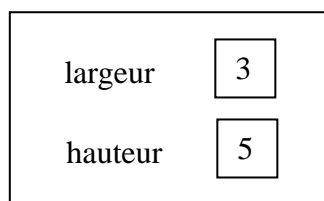
La place mémoire nécessaire pour chacune des variables d'instance est réservée.

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

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

En mémoire :

Objet *r*



Exemple 2

```

class Vehicule
{ String plaque, marque, modele ;           /* variables d'instance*/
  int jaugeEssence ;

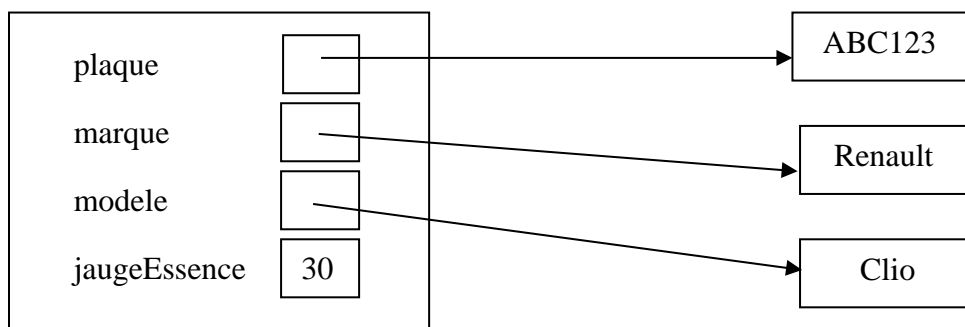
  Vehicule (String p, String ma, String mo, int j)  /* constructeur */
  { plaque = p ;
    marque = ma ;
    modele = mo ;
    jaugeEssence = j ;}

  void ajouterEssence (int nbLitres)             /* méthodes */
  { jaugeEssence += nbLitres ;}

  String typeVehicule( )
  { return marque + " " + modele ;}
}

class Principal
{ public static void main (String[ ] args)      /* signature de la méthode main */
  { Vehicule v = new Vehicule(" ABC123","Renault","Clio",30) ;
    /*déclaration + allocation de mémoire + initialisation des variables d'instance en une
    seule instruction */
  }
}

```

*En mémoire :*Objet *v*

Les variables d'instance *plaque*, *marque* 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 initialise les variables d'instance à des valeurs par défaut (nombre : 0, booléen: false, objet: référence null).

1.5. Accès aux variables d'instance

1.5.1. 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 quelque chose à l'écran est :

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

Exemple 1

```
class Principal
{ public static void main (String[ ] args)
  { Rectangle r = new Rectangle (3,5);
    System.out.println ("la hauteur de r est: " ⊕ r.hauteur + " cm");
    ⇒ affiche à l'écran : la hauteur de r est : 5 cm
                                /* 5 est la valeur de la variable hauteur de l'objet r */
  }
}
```

Opérateur de concaténation

Exemple 2

```
class Principal
{ public static void main (String[ ] args)
  { Vehicule v = new Vehicule(" ABC123","Renault","Clio", 30);
    System.out.println ("le véhicule immatriculé " + v.plaque +
                        "\ndispose d'une réserve de " + v.jaugeEssence + " litres");
    ⇒ affiche à l'écran : le véhicule immatriculé ABC123
                        dispose d'une réserve de 30 litres
  }
}
```

1.5.2. Accès en écriture

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

Exemple:

```
class Principal
{ public static void main (String[ ] args)
  { Rectangle r = new Rectangle (3,5);
    r.largeur = 2 ;                /* affecte la valeur 2 à la variable largeur de l'objet r */
    System.out.println ("la largeur de r est: " + r.largeur + " cm");
    ⇒ affiche à l'écran : la largeur de r est : 2 cm

    Vehicule v = new Vehicule(" ABC123","Renault","Clio",30);
    v.jaugeEssence = 45 ;        /* affecte la valeur 45 à la variable jaugeEssence de l'objet v */
  }
}
```

```
System.out.println ("le réservoir du véhicule v contient: " + v.jaugeEssence + " litres");  
    ⇨ affiche à l'écran : le réservoir du véhicule v contient: 45 litres  
}  
}
```

1.6. Appel de méthodes sur des objets

1.6.1. 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:

```
class Principal  
{ public static void main (String[ ] args)  
  {   Rectangle r = new Rectangle (3,5) ;  
      int res ;  
      res = r.surface() ;  
          /* res reçoit le résultat de l'exécution de la méthode surface() sur l'objet r */  
      System.out.println ("la surface de r est: " + res);  
          ⇨ affiche à l'écran : la surface de r est: 15  
  }  
}
```

Exemple 2 :

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

Solution 1 (3 instructions):

```
class Principal  
{ public static void main (String[ ] args)  
  {   Vehicule v = new Vehicule(" ABC123","Renault","Clio",30) ;  
      ① String type ;  
      ② type = v.typeVehicule() ;  
      ③ System.out.println ("le véhicule v est du type: " + type);  
          ⇨ affiche à l'écran: le véhicule v est du type: Renault Clio  
  }  
}
```

Solution 2 (1 seule instruction) :

```
class Principal
{ public static void main (String[ ] args)
  {   Vehicule v = new Vehicule(" ABC123","Renault","Clio",30) ;
      ① System.out.println ("le véhicule v est du type: " + v.typeVehicule() );
          ⇨ affiche à l'écran: le véhicule v est du type: Renault Clio
  }
}
```

1.6.2. 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**.

```
class Principal
{ public static void main (String[ ] args)
  { Rectangle r = new Rectangle (3,5) ;
    r.elargir(1) ; /* augmente de 1 la largeur de l'objet r */
    System.out.println ("la largeur de r est: " + r.largeur + " cm");
        ⇨ affiche à l'écran : la largeur de r est : 4 cm

    Vehicule v = new Vehicule("ABC123","Renault","Clio",30) ;
    v.ajouterEssence(10) ; /* ajoute 10 au contenu de la variable jaugeEssence
                           de l'objet v */
    System.out.println ("le réservoir du véhicule v contient: " + v.jaugeEssence + " litres");
        ⇨ affiche à l'écran : le réservoir du véhicule v contient: 40 litres
  }
}
```

1.7. Difficultés de choisir entre variables d'instance et méthodes

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

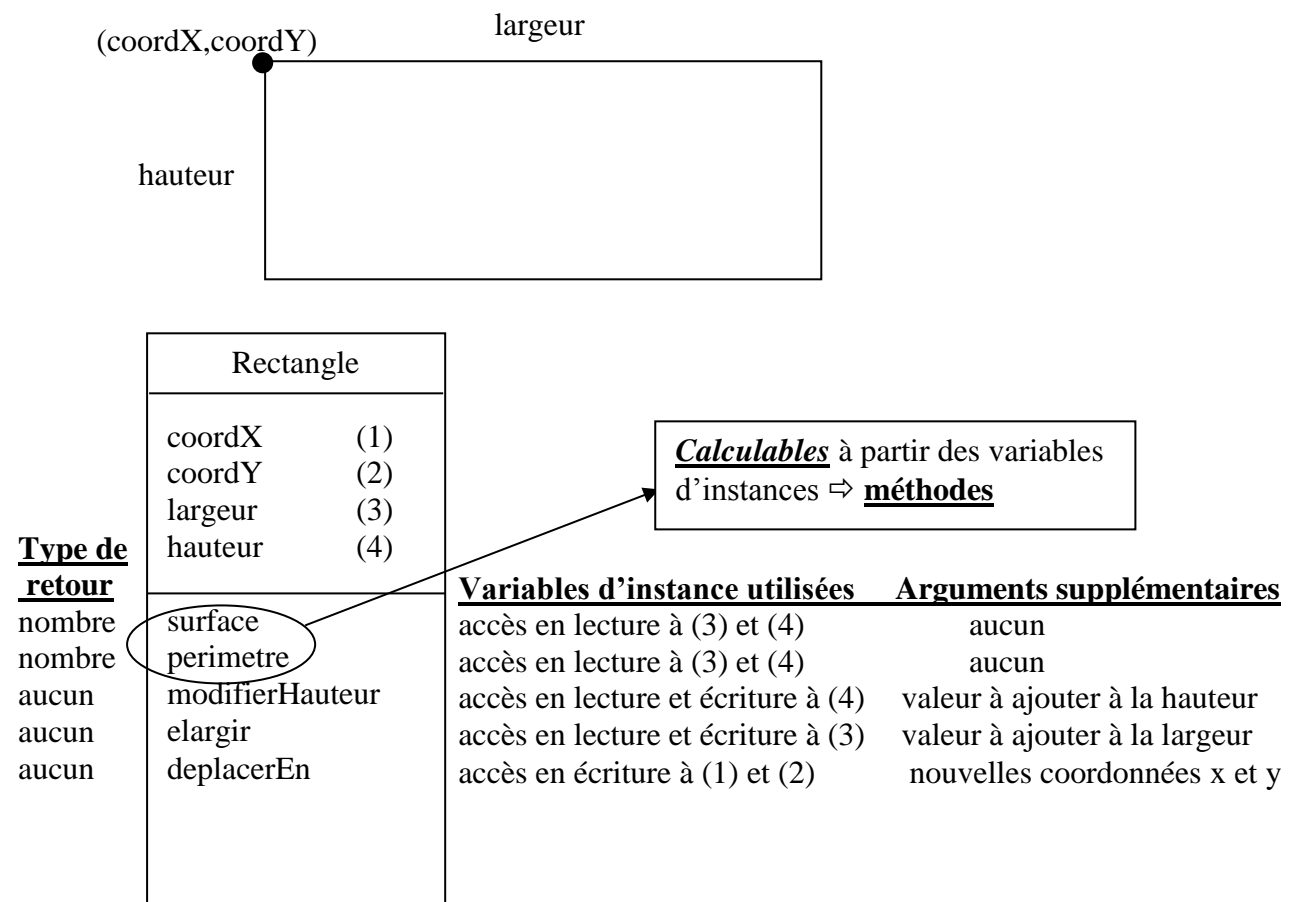
Toute caractéristique dérivable ou calculable à partir d'autres doit être déclarée sous forme de méthode.

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.



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 coordX, coordY, largeur, hauteur;

    Rectangle(int x, int y, int largeurDonnee, int hauteurDonnee)
        { coordX = x; coordY = y; largeur = largeurDonnee; hauteur = hauteurDonnee;}

    int surface( )
        {return largeur * hauteur;}
    int perimetre( )
        {return (2*largeur) + (2*hauteur);}
    void elargir (int augmentation)
        {largeur += augmentation; }
    void modifierHauteur (int augmentation)
        {hauteur += augmentation; }
    void deplacerEn (int newX, int newY)
        {coordX = newX; coordY = newY;}
}
```

Exemple 2 :

Objectif : traiter des étudiants inscrits en informatique :

- identifier tout étudiant (càd renseigner ses coordonnées : nom, adresse, âge, login name,...)
- calculer la réussite (ou l'échec) d'un étudiant à partir des résultats obtenus dans différentes branches (langues, math, informatique,...)

EtudiantInfo	
pre nom	(1)
nom	(2)
adresse	(3)
dateNaissance	(4)
section	(5)
annee	(6)
pourLangues	(7)
pourMath	(8)
pourInfo	(9)
age	
loginName	
moyenneCours	(a)
nbEchecs	(b)
nbDispenses	
aReussi	

Calculables à partir des variables d'instances ⇒ méthodes

Type de retour	Variables d'instance utilisées	Arguments supplémentaires
nombre	accès en lecture à (4)	aucun
chaîne car.	accès en lecture à (1), (2), (5), (6)	aucun
nombre	accès en lecture à (7), (8), (9)	aucun
nombre	accès en lecture à (7), (8), (9)	aucun
nombre	accès en lecture à (7), (8), (9)	aucun
booléen	aucun : appel des méthodes (a) et (b)	aucun

↪ cfr section 1.8.

Les caractéristiques prénom, nom, adresse, date de naissance, section, année, et les pourcentages en langues, math et informatique ne sont pas calculables à partir d'autres. Elles sont donc déclarées sous forme de variables d'instance.

L'âge, le login name, la moyenne des cours, le nombre d'échecs et de dispenses d'un étudiant peuvent être calculés à partir du prénom, du nom, de la date de naissance, de la section et l'année, et des pourcentages en langues, math et informatique. Ces caractéristiques sont donc déclarées sous forme de méthodes.

La réussite d'un étudiant peut être calculée à partir des méthodes *moyenneCours()* et *nbEchecs()*. Il s'agit donc également d'une méthode (méthode *aReussi()*).

Le code complet de la classe EtudiantInfo est donc :

```
package etudiant;
```

```
import java.util.GregorianCalendar;      /* import de la classe GregorianCalendar */
```

```
class EtudiantInfo
```

```
{
```

```
    String prenom, nom, adresse, section;
```

```
    GregorianCalendar dateNaissance;      /* classe de référence : permet de gérer facilement  
                                           des dates */
```

```
    int annee;
```

```
    float pourcLangues, pourcMath, pourcInfo;
```


```
    EtudiantInfo( String p, String n, String ad, String s, GregorianCalendar d, int an,  
                  float pl, float pm, float pi)  
    { prenom = p; nom = n; adresse = ad; section = s; dateNaissance = d; annee = an;  
      pourcLangues = pl; pourcMath = pm; pourcInfo = pi; }
```

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

```
    String loginName( )  
    { ... }
```

```
    float moyenneCours( )  
    { return (pourcLangues + pourcMath + pourcInfo)/3; }
```

```
    int nbEchecs( )  
    { int echecs = 0;  
      if (pourcLangues < 50) echecs ++;  
      if (pourcMath < 50) echecs ++;  
      if (pourcInfo < 50) echecs ++;  
      return echecs; }
```



```
variable locale à la méthode  
( cfr section 1.9.)
```

```

int nbDispenses( )
{
    int dispenses = 0;
    if (pourcLangues >=60) dispenses ++;
    if (pourcMath >=60) dispenses ++;
    if (pourcInfo >=60) dispenses ++;
    return dispenses;
}

boolean aReussi( )
{
    ...
}
/* cfr section 1.8. */

```

En conclusion :

- ① Si une caractéristique est **calculable** à partir d'autres caractéristiques (variables d'instance ou méthodes), alors cette caractéristique est déclarée sous forme de **méthode** et en aucun cas sous forme de variable d'instance.
- ② Les **variables d'instance** d'une classe ne sont **jamais** passées comme **arguments** des méthodes de la classe : en effet, toute variable d'instance est directement accessible par les méthodes.

1.8. Une méthode peut appeler une autre méthode de la même classe

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

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

Reprenons la méthode `aReussi()` de la classe `EtudiantInfo`. On peut calculer si un étudiant a réussi à partir de la moyenne qu'il a obtenue pour ses cours et du nombre de ses échecs. Par simplification, partons du principe que pour réussir, il faut minimum 60 % de moyenne et aucun échec (aucune délibération possible).

```

class EtudiantInfo
{
    ...                               /* variables d'instances et constructeur*/
    float moyenneCours( ) { .... } ; /* calcul de la moyenne d'un étudiant */
    int nbEchecs( ) { ..... } ;      /* calcul du nombre d'échecs d'un étudiant */

    boolean aReussi( )                /* méthode qui vérifie si l'étudiant a réussi */
    {
        if ( this.nbEchecs( ) == 0 && this.moyenneCours( ) >=60 )
            return true ;
        else return false ;
    }
}
/* this : référence vers l'objet courant */

```

Il est possible de demander à java d'exécuter une autre méthode sur l'objet courant. La syntaxe est : **this**.nomMethode(...). Le mot réservé **this** fait donc référence à l'objet en cours. Lors de l'appel de la méthode aReussi sur un objet de type EtudiantInfo, la méthode moyenneCours() (via l'instruction **this.moyenneCours()**) est exécutée sur cet objet ainsi que la méthode nbEchecs() (via l'instruction **this.nbEchecs()**).

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

```
class Principal
{ public static void main (String[] args)
  { EtudiantInfo e = new EtudiantInfo(...);
    if ( e.aReussi() == true) /* NB peut aussi s'écrire: if (e.aReussi() ) */
      ...
      ↓
à l'exécution, cela équivaut à : { if (@nbEchecs() == 0 && @moyenneCours() >=60)
    }
  }
```

N.B1 : Il existe encore une écriture plus courte : this.nomMethode(...) peut s'écrire directement nomMethode(...). Ainsi la méthode aReussi() peut s'écrire comme suit :

```
{ if (○nbEchecs() == 0 && ○moyenneCours() >=60)
  return true ;          /* this est considéré par défaut */
else return false ;
}
```

N.B2: 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.

Exemple :

```
boolean aReussi() { ... } ;
float moyenneCours() { .... } ;
int nbEchecs() { ..... } ;
```

Même si les méthodes sont déclarées dans cet ordre, la méthode *aReussi()* peut appeler les méthodes *moyenneCours()* et *nbEchecs()* 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.

1.9. Portée des variables

On peut identifier trois catégories de variable dans une classe : les variables d'instance, les arguments et les variables locales aux méthodes.

1. Les variables d'instance déclarées dans la classe sont accessibles de partout dans cette classe (par toutes les méthodes, constructeurs ou non).
2. 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.
3. 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 :

```
class Entrainement
{
    String nomCoureur ;
    int longueurParcours, tempsParcours ;

    Entrainement (String qui, int quoi, int combien)
    {
        nomCoureur = qui ;
        longueurParcours = quoi ;
        tempsParcours = combien ;
    }

    float vitesseMoyenne( )
    {
        float moy ;
        moy = longueurParcours / tempsParcours ;
        return moy ;
    }

    int comparerAuMeilleur (int tempsMeilleur) /* compare au temps mis par le meilleur */
    {
        return tempsParcours - tempsMeilleur ;
    }
}
```

La variable *moy* déclarée dans la méthode *vitesseMoyenne()* reste locale à cette méthode.

Aucune autre méthode (ni constructeur) ne peut y avoir accès !

Les arguments *qui*, *quoi* et *combien* 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 variable d'instance : accessible par n'importe quelle méthode / n'importe quel constructeur de la classe.
- ② argument : local à la méthode / au constructeur.
- ③ variable déclarée dans une méthode/ constructeur : locale à cette méthode / à ce constructeur

1.10. Surcharge (overloading) des constructeurs et méthodes

1.10.1. 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. Pour rappel, la signature d'une méthode et/ou constructeur, c'est la combinaison de son nom et de ses arguments.

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

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

Exemple 1:

Première version :

```
class Vehicule
{ String plaque, marque, modele;
  int jaugeEssence;

  Vehicule (String plaq, String mark, String mod, int jauge)
    { plaque = plaq ;  marque = mark ;  modele = mod ;  jaugeEssence = jauge ;}

  Vehicule (String plaq, String mark, String mod)
    { plaque = plaq ;  marque = mark ;  modele = mod ; jaugeEssence = 0 ;}
    /* signature différente du premier constructeur car moins de paramètres) */
  ...
}
```

Le second constructeur pourra être utilisé pour créer des véhicules dont la jauge à essence est vide.

```
class Principal
{ public static void main (String[ ] args)
  { Vehicule v1 = new Vehicule("ABC123","Renault","Clio",30) ;
    Vehicule v2 = new Vehicule("DEF456","Ford","Fiesta");
    System.out.println ("le réservoir du véhicule v2 contient: " + v2.jaugeEssence + " litres");
    ⇒ affiche à l'écran : le réservoir du véhicule v2 contient: 0 litres
  }
}
```

Seconde version :

Il existe une version raccourcie permettant d'économiser des lignes de code.

Dans un constructeur, il est possible de faire appel à un autre constructeur de la même classe.

La syntaxe est : this(...).

```

class Vehicule
{ String plaque, marque, modele;
  int jaugeEssence;

  Vehicule (String plaq, String mark, String mod, int jauge)
    { plaque = plaq ;  marque = mark ;  modele = mod ;  jaugeEssence = jauge ;}

  Vehicule (String plaq, String mark, String mod)          /* signature différente du premier
                                                              constructeur car moins de paramètres) */
    { this(plaq, mark, mod, 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**. A ne pas confondre avec **this.nomMethode(...)**, qui appelle la méthode *nomMethode(...)* sur l'objet courant.

Exemple 2 :

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

```

class Etudiant
{ String prenomNom, section ;
  int annee;

  Etudiant (String nom, String sect, int an)
    { prenomNom = nom;
      section = sect;
      annee = an;
    }

  Etudiant (String nom, int an)          /* pour créer des étudiants de technologie*/
    { this(nom, "techno", an) ;
    }

  Etudiant (String nom, String sect)     /* pour créer des étudiants de première année*/
    { this(nom, sect, 1) ;
    }

  Etudiant (String nom)                  /* pour créer des étudiants de 1ère technologie*/
    { this(nom, 1) ;
      /* ou: this(nom, "techno") ;*/
    }
}

```

Quatre constructeurs ont été prévus dans cette classe. Le premier avec autant d'arguments qu'il y a de variables d'instance. Le deuxième constructeur facilite la tâche lors de la création d'étudiants inscrits en technologie de l'informatique (deux arguments: le nom (+ prénom) et l'année). 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 technologie de l'informatique (un seul argument : le nom (+ prénom)). Les deuxième et troisième constructeurs ont, tous deux, deux arguments, mais ils diffèrent de par leur type.

```
class Principal
{
    public static void main (String[ ] args)
    {
        Etudiant e1 = new Etudiant ("J. Petit","compta",3) ;
        System.out.println (e1.prenomNom + " est inscrit en " + e1.annee + "e " + e1.section);
        ⇒ affiche à l'écran: J. Petit est inscrit en 3e compta

        Etudiant e2 = new Etudiant ("P. Grant",2);
        System.out.println (e2.prenomNom + " est inscrit en " + e2.annee + "e " + e2.section);
        ⇒ affiche à l'écran: P. Grant est inscrit en 2e techno

        Etudiant e3 = new Etudiant ("F. Medium","droit") ;
        System.out.println (e3.prenomNom + " est inscrit en " + e3.annee + "e " + e3.section);
        ⇒ affiche à l'écran: F. Medium est inscrit en 1e droit

        Etudiant e4 = new Etudiant ("X. Extra");
        System.out.println (e4.prenomNom + " est inscrit en " + e4.annee + "e " + e4.section);
        ⇒ affiche à l'écran: X. Extra est inscrit en 1e techno
    }
}
```

A chaque appel de constructeur, java analyse la signature de celui-ci pour déterminer quel constructeur utiliser.

1.10.2. 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 leur signature diffère (type des 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 pourrait 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 maMethode (int x, float y, String z)
{ ... }
int maMethode (int a, float b, String c)
{ ... }
```

Exemples corrects :

Exemple 1 :

```
class Vehicule
{
    String plaque, marque, modele;
    int jaugeEssence, capaciteReservoir;

    Vehicule(String p, String mq, String md, int j, int c)
        { plaque = p; marque = mq; modele = md;
          jaugeEssence = j; capaciteReservoir = c; }

    Vehicule(String p, String mq, String md, int c)
        { this(p,mq,md,0,c); }

    String typeVehicule( )
        { return marque + " " + modele; }

    void ajouterEssence (int nbLitres)
        { jaugeEssence +=nbLitres; }

    void ajouterEssence ( )
        { jaugeEssence = capaciteReservoir; }
}
```

}

Surcharge de
constructeurs

}

Surcharge de
méthodes

La méthode `void ajouterEssence (int nbLitres)` permet de rajouter *nbLitres* d'essence dans le réservoir, tandis que la méthode `void ajouterEssence ()` permet d'enregistrer que le plein d'essence a été fait. Autrement dit, la méthode `ajouterEssence()` permet de mettre à jour la variable d'instance *jaugeEssence* pour représenter l'état de la jauge à essence après passage à la pompe pour faire le plein. La jauge étant remplie, la variable d'instance *jaugeEssence* contient désormais le nombre de litres d'essence égal à la valeur contenue dans la variable d'instance *capaciteReservoir*.

```
class Principal
{ public static void main(String[ ] args)
  { Vehicule v = new Vehicule("ABC123", "Renault", "Espace", 50);
    v.ajouterEssence(30);
    System.out.println("Nombre de litres dans le réservoir: " + v.jaugeEssence);
    ⇒ affiche à l'écran: Nombre de litres dans le réservoir: 30

    v.ajouterEssence( );
    System.out.println("Nombre de litres dans le réservoir: " + v.jaugeEssence);
    ⇒ affiche à l'écran: Nombre de litres dans le réservoir: 50
  }
}
```


Exemple 2 :

```

package etudiant;
import java.util.GregorianCalendar;

class EtudiantInfo {

    String prenom, nom, adresse, section;
    GregorianCalendar dateNaissance;
    int annee;
    float pourcLangues, pourcMath, pourcInfo;
    ...                               /* constructeurs et méthodes */

    int nbDispenses()
    { int dispenses = 0;
      if (pourcLangues >=60) dispenses ++;
      if (pourcMath >=60) dispenses ++;
      if (pourcInfo >=60) dispenses ++;
      return dispenses; }

    int nbDispenses(int seuil)
    { int dispenses = 0;
      if (pourcLangues >= seuil) dispenses ++;
      if (pourcMath >= seuil) dispenses ++;
      if (pourcInfo >= seuil) dispenses ++;
      return dispenses; }

}

```

**Surcharge de
méthodes**

Les méthodes nbDispenses(...) permettent soit de calculer le nombre de dispenses à partir de 60%, soit de les calculer à partir d'un autre pourcentage donné en argument (ex : 70%).

```

package etudiant;
import java.util.GregorianCalendar;

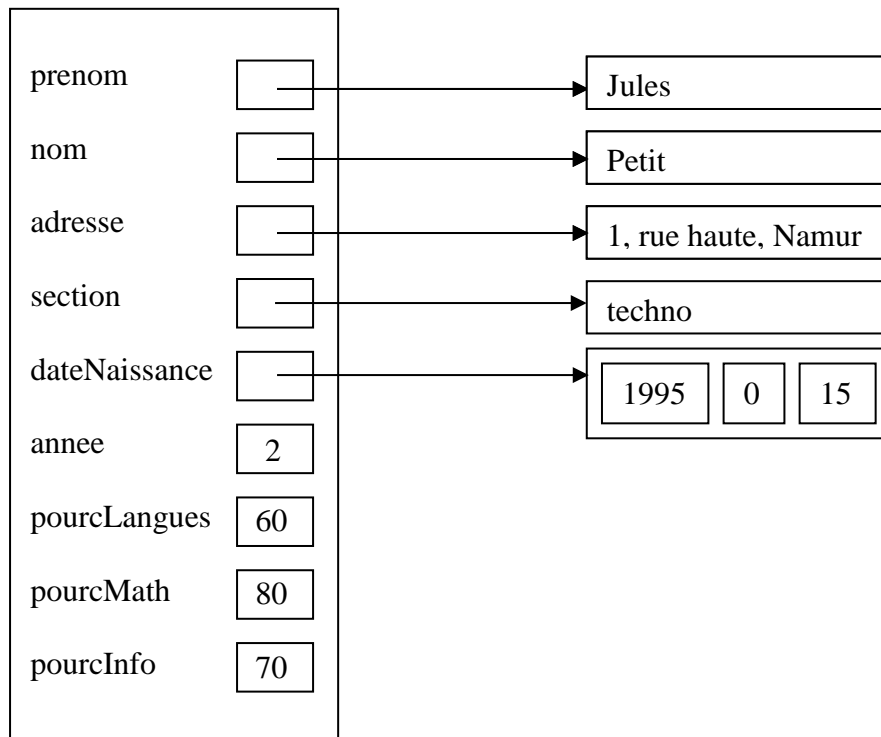
class Principal
{ public static void main(String[] args)
  { GregorianCalendar naiss = new GregorianCalendar(1995,0,15);
    EtudiantInfo e ;
    e = new EtudiantInfo("Jules","Petit","1, rue haute, Namur","techno",naiss, 2, 60, 80, 70);
    System.out.println("dispenses = " + e.nbDispenses() );
    ⇒ affiche à l'écran: dispenses = 3
    System.out.println("dispenses = " + e.nbDispenses(70) );
    ⇒ affiche à l'écran: dispenses = 2
  }
}

```

L'instruction `new GregorianCalendar(1995,0,15)` permet de créer un objet de type `GregorianCalendar`. Le constructeur utilisé prend trois arguments : le premier correspond à l'année, le deuxième correspond au mois et le troisième correspond au jour. Les valeurs possibles pour le mois vont de 0 (= janvier) à 11 (= décembre). Les valeurs possibles pour le jour vont de 1 à 31.

N.B. En mémoire, l'objet *e* a la structure suivante :

Objet *e*



En effet, la variable d'instance `dateNaissance` est une référence vers un objet (de type `GregorianCalendar`) se trouvant ailleurs en mémoire centrale.

1.11. La méthode `toString()`

Il s'agit d'une méthode dont le nom est un mot réservé. Sa déclaration est immuable:

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

Cette méthode permet de décrire tout objet de n'importe quelle classe sous forme d'une chaîne de caractères. Elle est appelée *automatiquement* par Java chaque fois qu'un nom d'objet apparaît "là où on attend" une chaîne de caractères.

Exemple 1 :

```
class Rectangle
{
    int coordX, coordY, largeur, hauteur ;

    ...                               /* constructeurs et méthodes */

    public String toString()
    {
        return "point d'encrage : (" + coordX + ", " + coordY + ")"
            + "\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'àd via quelle chaîne de caractères.

```
class Princip
{
    public static void main(String[ ] args)
    {
        Rectangle r = new Rectangle (2, 3, 4, 5) ;           /* création de l'objet r */
        System.out.println( r ) ;                             /* appel implicite de r.toString() */
        ⇒ affiche à l'écran:  point d'encrage : (2,3)
                               largeur : 4
                               hauteur : 5

        String temp = "Coordonnées du rectangle r: \n" + r; /* appel implicite de r.toString() */
        System.out.println( temp ) ;
        ⇒ affiche à l'écran:  Coordonnées du rectangle r:
                               point d'encrage : (2,3)
                               largeur : 4
                               hauteur : 5
    }
}
```

La méthode `toString()` est appelée automatiquement par 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 temp = "Coordonnées du rectangle r: \n" + r;`

L'opérateur de concaténation (+) est sensé concaténer deux chaînes de caractères. Or, on lui demande ici de concaténer une chaîne de caractères et un objet `r`. La méthode `toString()` est appelée ici aussi automatiquement par java sur cet objet `r`.

Exemple 2 :

```
class EtudiantInfo
{ String prenom, nom, adresse, section;
  GregorianCalendar dateNaissance;
  int annee;
  float pourcLangues, pourcMath, pourcInfo;
  ...                               /* constructeurs et méthodes */
  public String toString( )
  { return "L'étudiant " + prenom + " " + nom + " inscrit en " + annee + "e "+ section;
  }
}

import java.util.*;

class Principal
{ public static void main(String[] args)
  { GregorianCalendar naiss = new GregorianCalendar(1995,0,15);
    EtudiantInfo e ;
    e = new EtudiantInfo("Jules","Petit","1, rue haute, Namur","techno",naiss, 2, 60, 80, 70);
    System.out.println(e);
    ⇒ affiche à l'écran: L'étudiant Jules Petit inscrit en 2e techno

    String s = e + " a une moyenne de " + e.moyenneCours( )+ " %";
    System.out.println(s);
    ⇒ affiche à l'écran: L'étudiant Jules Petit inscrit en 2e techno a une moyenne de 70.0 %
  }
}
```

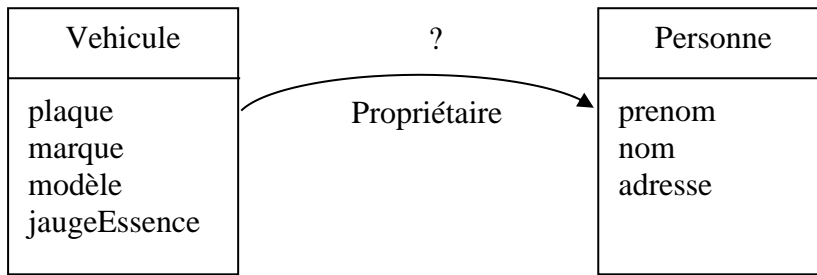
Notons que si la méthode `toString()` n'est pas définie dans une classe, java propose sa propre description de tout objet de la classe. La chaîne de caractères proposée par java n'est pas très conviviale (ex : *dirX.Personne@16f0472*). Il est dès lors fortement conseillé de définir la méthode `toString()` dans chacune des classes que l'on écrit.

1.12. Liens entre objets

1.12.1. Liens 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`.



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**.

```

class Personne
{
    String prenom, nom, adresse ;

    Personne (String p, String n, String a)          /* constructeur */
        { prenom = p;   nom = n;   adresse = a; }

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

class Vehicule
{
    String plaque, marque, modele;
    int jaugeEssence, capaciteReservoir;
    Personne proprio;                          /* variable d'instance de type Personne */

    Vehicule(String plaque, String marque, String modele, int jauge,
              int capaciteMax, Personne propriétaire)
    { this.plaque = plaque;
      this.marque = marque;
      this.modele = modele;
      jaugeEssence = jauge;
      capaciteReservoir= capaciteMax;
      proprio = proprietaire; }

    ...                                           /* méthodes */
}
  
```

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

*N.B. S'il y a ambiguïté dans les noms de variable (ex : même nom d'argument qu'une variable d'instance), on peut lever l'ambiguïté en utilisant le mot réservé **this** :*

***this.nomVariable** fait appel à la variable d'instance portant le nom **nomVariable**. Par conséquent, l'instruction **this**.plaque = plaque a pour effet de prendre le contenu de l'argument plaque et de le placer dans la variable d'instance plaque.*

```

public class Principal
{ public static void main(String[] args)
  { Personne pers = new Personne("Paul","Petit","1, rue de fer, 5000 Namur");
    Vehicule veh = new Vehicule("ABC123"," Renault ","Clio",10, 50, pers);

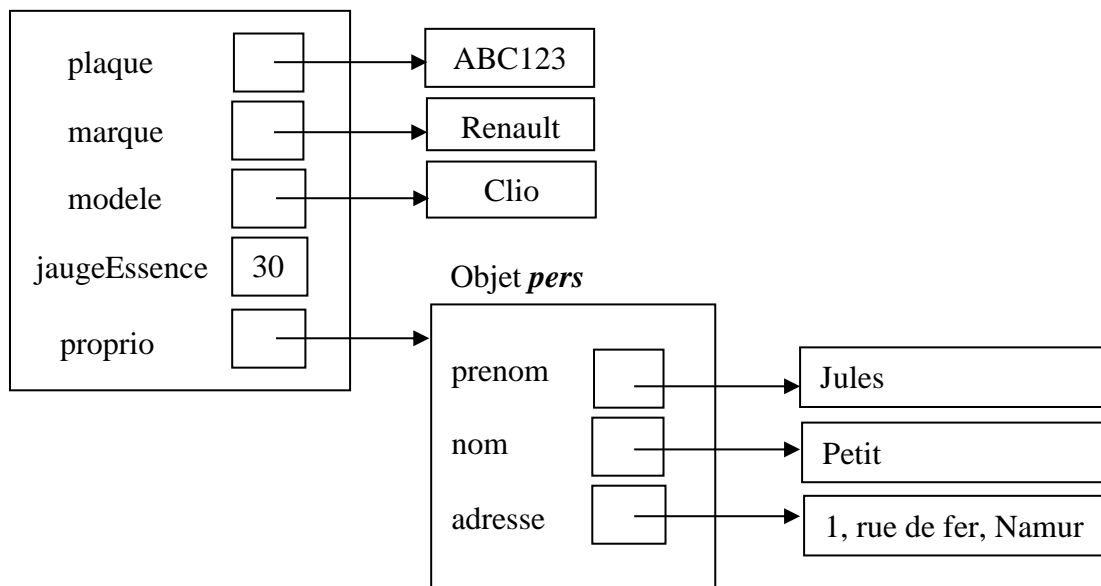
    System.out.println(veh.plaque);
    ⇒ affiche à l'écran: ABC123

    System.out.println("L'adresse du propriétaire du véhicule est : " + veh.proprio.adresse);
    ⇒ affiche à l'écran:
      L'adresse du propriétaire du véhicule est : 1, rue de fer, 5000 Namur
  }
}

```

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

Objet **veh**



En effet, la variable d'instance **proprio** du véhicule **veh** 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.

veh.proprio fait référence à l'objet de type **Personne** référencé par la variable d'instance **proprio** de l'objet **veh**. Autrement dit, **veh.proprio** référence l'objet **pers**.

veh.proprio.adresse fait référence à l'objet de type **String** référencé par la variable d'instance **adresse** de l'objet **pers**.

1.12.2. Appel implicite à la méthode toString() d'un objet référencé par une variable d'instance

```

class Vehicule
{ String plaque, marque, modele;
  int jaugeEssence, capaciteReservoir;
  Personne proprio;                                /* variable d'instance de type Personne */
  ...                                                /* constructeur et méthodes */
  String typeVehicule( )
  {return marque + ": " + modele;}

  public String toString( )
  {return "La " + this.typeVehicule( ) + " immatriculée " + plaque +
    "\nappartient à " + proprio;}
    /* proprio est un objet de type Personne, pour l'afficher
    Java fera appel au toString( ) de la classe Personne */
}

```

La méthode `toString()` de la classe `Vehicule` fait appel à la variable d'instance `proprio` 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 `proprio`.

```

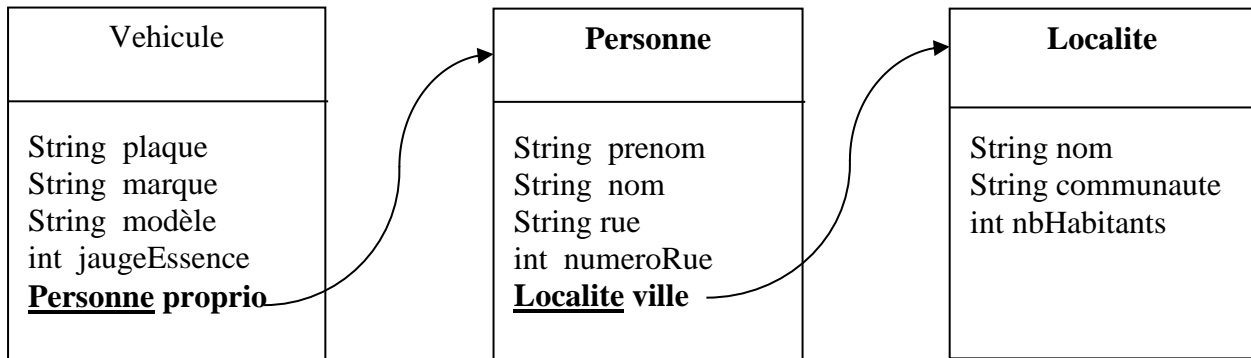
public class Principal
{ public static void main(String[ ] args)
  { Personne pers = new Personne("Jules","Petit","1, rue de fer, 5000 Namur");
    Vehicule veh = new Vehicule("ABC123"," Renault ","Clio",10, 50, pers);
      1 objet personne
    System.out.println( veh.proprio);          /* appel au toString( ) sur proprio */
      ⇒ affiche à l'écran:
        Jules Petit domicilié au 1, rue de fer 5000 Namur

    System.out.println(veh);
      ⇒ affiche à l'écran:
        La Renault Clio immatriculée ABC123
        appartient à Jules Petit domicilié au 1, rue de fer 5000 Namur
  }
}

```

1.12.3. Plus de deux classes reliées

Exemple 1 :



```

class Localite
{ String nom, commune;
  int nbHabitants;

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

  String langue( )
  { String lang = "";
    if (commune.equals("flamande") == true) lang = "néerlandais";
    else if (commune.equals("française") == true) lang = "français";
    else lang = "allemand";
    return lang; }

  public String toString( )
  { return nom + " (" + nbHabitants + " habitants)"; }
}
  
```

Pour **comparer le contenu de deux objets de type String**, il faut utiliser la méthode `equals(...)` de la classe `String` : **`obj1.equals(obj2)`**. Cette méthode est appelée 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).

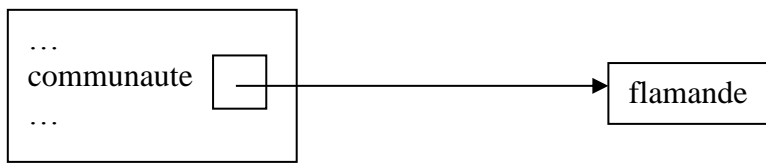
Ex : `String a = " Jules " ;`
`String b = " Julos " ;`
 If (`a.equals(b) == true`) ...

N.B. Il serait illogique d'écrire : `if (a == b)` : 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: **`obj.equals(val)`**. L'argument `val` est la valeur à laquelle il faut comparer la chaîne de caractères référencée par l'objet `obj`.

Ex : `commune.equals("flamande")` compare la chaîne de caractères référencée par la variable d'instance `commune` à la valeur `flamande`.

En mémoire :



class **Personne**

```
{ String prenom, nom, rue;
  int numeroRue;
  Localite ville;
```

```
  Personne(String prenom, String nom, String rue, int numero, Localite localite )
    {this.prenom = prenom; this.nom = nom; this.rue = rue; numeroRue = numero;
      ville = localite; }
```

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

Appel au toString() de Localite

class **Vehicule**

```
{ String plaque, marque, modele;
  int jaugeEssence, capaciteReservoir;
  Personne proprio;
```

```
  Vehicule(String plaque, String marque, String modele, int jauge,
    int capaciteMax, Personne propriétaire)
    {this.plaque = plaque; this.marque = marque; this.modele = modele;
      jaugeEssence = jauge; capaciteReservoir= capaciteMax;
      proprio = proprietaire; }
```

```
  Vehicule(String plaque, String marque, String modele, int capaciteMax,
    Personne propriétaire)
    {this(plaque,marque,modele,0,capaciteMax,proprietaire);}
```

```
  String typeVehicule( )
    {return marque + " " + modele;}
  void ajouterEssence (int nbLitres)
    {jaugeEssence +=nbLitres;}
  void ajouterEssence ( )
    {jaugeEssence = capaciteReservoir;}
```

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

Appel au toString() de Personne

```

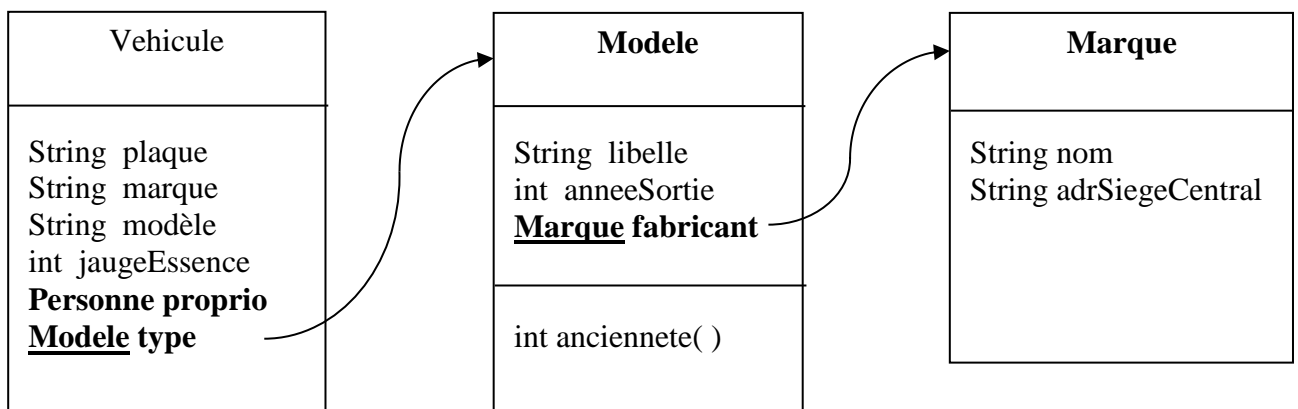
class Principal
{ public static void main(String[] args)
  { Localite loc = new Localite("Namur","française",15000);
    Personne pers = new Personne("Paul","Petit","de fer",50,loc);
    Vehicule v = new Vehicule("ABC123","Ford","Escort",40,pers);

    System.out.println(v);
    ⇒ affiche à l'écran:
      La Ford Escort immatriculée ABC123
      appartient à Paul Petit domicilié au 50, rue de fer à Namur (15000 habitants)
      1 objet personne

    System.out.println( v.proprio);
    ⇒ affiche à l'écran:
      Paul Petit domicilié au 50, rue de fer à Namur (15000 habitants)
      1 objet localité

    System.out.println( v.proprio.ville);
    ⇒ affiche à l'écran:
      Namur (15000 habitants)
    System.out.println(v.proprio.ville.nom);
    ⇒ affiche à l'écran:
      Namur
    System.out.println(v.proprio.ville.langue( ));
    ⇒ affiche à l'écran:
      français
  }
}

```

Exemple 2 :

```

class Marque
{ String nom, adrSiegeCentral;

    Marque(String nom, String adresse)
        {this.nom = nom;  adrSiegeCentral = adresse; }

    public String toString( )
        {return nom ; }
}

class Modele
{ String libelle;
  int anneeSortie;
  Marque fabricant;

  Modele(String libelle, int annee, Marque marque)
      {this.libelle = libelle;  anneeSortie = annee;
       fabricant = marque; }

  int anciennete ( ) /* méthode non portable ! Mieux : retrouver année de la date système */
      {return 2015 - anneeSortie;} /* cfr Chapitre 4: constants */

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

class Vehicule
{ String plaque;
  int jaugeEssence, capaciteReservoir;
  Personne proprio;
  Modele type;

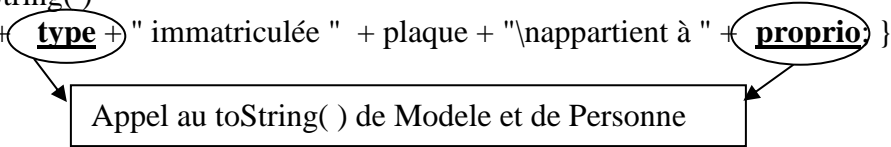
  Vehicule(String plaque, int jauge, int capaciteMax, Personne propriétaire, Modele modele)
      {this.plaque = plaque;  jaugeEssence = jauge;  capaciteReservoir = capaciteMax;
       proprio = propriétaire; type = modele; }

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

  void ajouterEssence (int nbLitres)
      {jaugeEssence += nbLitres;}
  void ajouterEssence ( )
      {jaugeEssence = capaciteReservoir;}

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

```



```

class Principal
{ public static void main(String[ ] args)
  { Localite loc = new Localite("Namur","française",15000);
    Personne pers = new Personne("Paul","Petit","de fer",50,loc);
    Marque marq = new Marque("Ford","1, rue haute, 1000 Bruxelles");
    Modele mod = new Modele("Escort",2000,marq);
    Vehicule v = new Vehicule("ABC123",40,pers,mod);
  }
}

```

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

⇒ affiche à l'écran:

La Ford Escort (sortie en 2000) immatriculée ABC123

appartient à Paul Petit domicilié au 50, rue de fer à Namur (15000 habitants)

1 objet modele

```
System.out.println( v.type );
```

⇒ affiche à l'écran:

Ford Escort (sortie en 2000)

```
System.out.println(v.type.anciennete( ));
```

⇒ affiche à l'écran: *15*

1 objet marque

```
System.out.println( v.type.fabricant );
```

⇒ affiche à l'écran: *Ford*

```
System.out.println(v.type.fabricant.adrSiegeCentral);
```

⇒ affiche à l'écran: *1, rue haute, 1000 Bruxelles*

```

}
}

```

1.13. 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 existante.

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.

La classe Point contient une méthode appelée **union**.

Le principe de la méthode **union** 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.

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

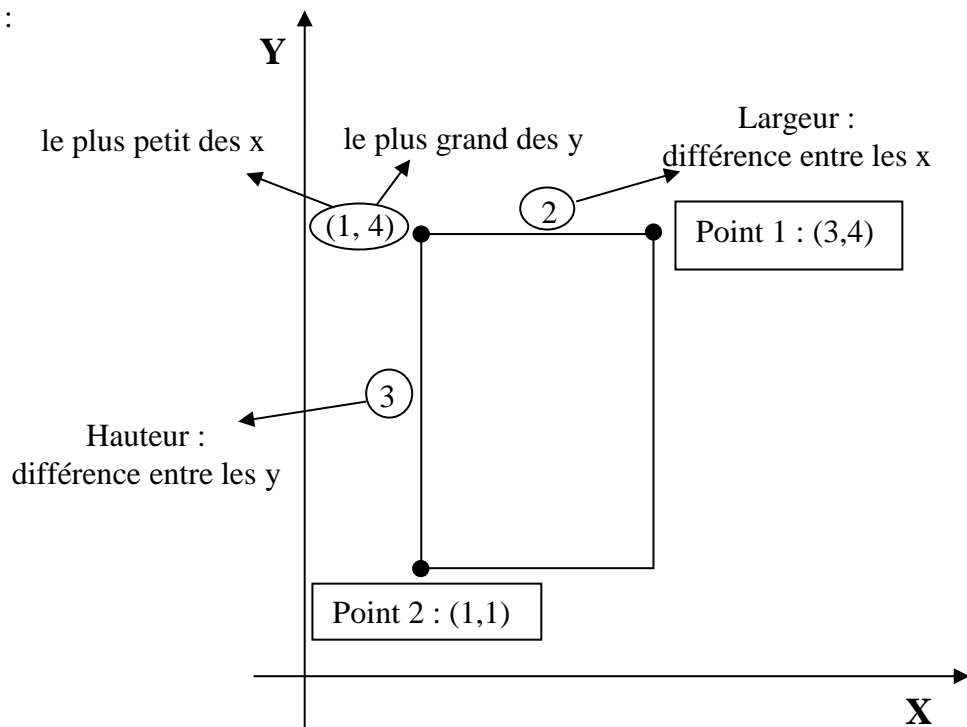
Les coordonnées x et y, la largeur et la hauteur du nouveau rectangle à créer à partir de deux points donnés sont calculés comme suit :

- 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 ;

Le résultat de la méthode **union** de la classe Point est de type **Rectangle**, classe que nous avons créée précédemment. Ce qui signifie que la méthode **union** retourne une référence vers un objet de type **Rectangle**. Cet objet de type Rectangle est créé dans le code de la méthode **union**.

La méthode **union** contiendra donc l'instruction : ... **new Rectangle(...)**.

En guise d'illustration, construisons le rectangle correspondant à l'union des points (1,1) et (3,4) :



La méthode ***union***(...) appartient à la classe Point. Par conséquent, elle sera appelée sur un objet de type Point. Cette méthode ne recevra donc qu'un seul argument supplémentaire qui est aussi un objet de type Point.

```
class Point
```

```
{ int x, y;
```

```
    Point(int x, int y)
```

```
    { this.x = x;
```

```
      this.y = y; }
```

```
    Rectangle union (Point p)
```

```
    { int nouvX, nouvY, nouvHauteur, nouvLargeur;
```

```
      if(this.x < p.x) nouvX = this.x;
```

```
        else nouvX = p.x;
```

```
      if(this.y > p.y) nouvY = this.y;
```

```
        else nouvY = p.y;
```

```
      if(this.x > p.x) nouvLargeur = this.x - p.x;
```

```
        else nouvLargeur = p.x - this.x;
```

```
      if(this.y > p.y) nouvHauteur = this.y - p.y;
```

```
        else nouvHauteur = p.y - this.y;
```

```
      return new Rectangle(nouvX, nouvY, nouvLargeur, nouvHauteur);
```

```
    }
```

```
    /* ou Rectangle union(Point p)
```

```
        { return new Rectangle (this.x<p.x?this.x:p.x,
```

```
                                this.y>p.y?this.y:p.y,
```

```
                                this.x>p.x?this.x-p.x:p.x-this.x,
```

```
                                this.y>p.y?this.y-p.y:p.y-this.y); }*/
```

```
    }
```

```
class Principal
```

```
{ public static void main(String[ ] args)
```

```
    { Point p1 = new Point(1,-2);
```

```
      Point p2 = new Point(4,-4);
```

```
      System.out.println(p1.union(p2));
```

```
        ⇒ affiche à l'écran:
```

```
            point d'encrage (1,-2)
```

```
            largeur: 3
```

```
            hauteur: 2
```

```
      Point p3 = new Point(2,1);
```

```
      Point p4 = new Point(4,5);
```

```
      System.out.println(p3.union(p4));
```

```
        ⇒ affiche à l'écran:
```

```
            point d'encrage (2,5)
```

```
            largeur: 2
```

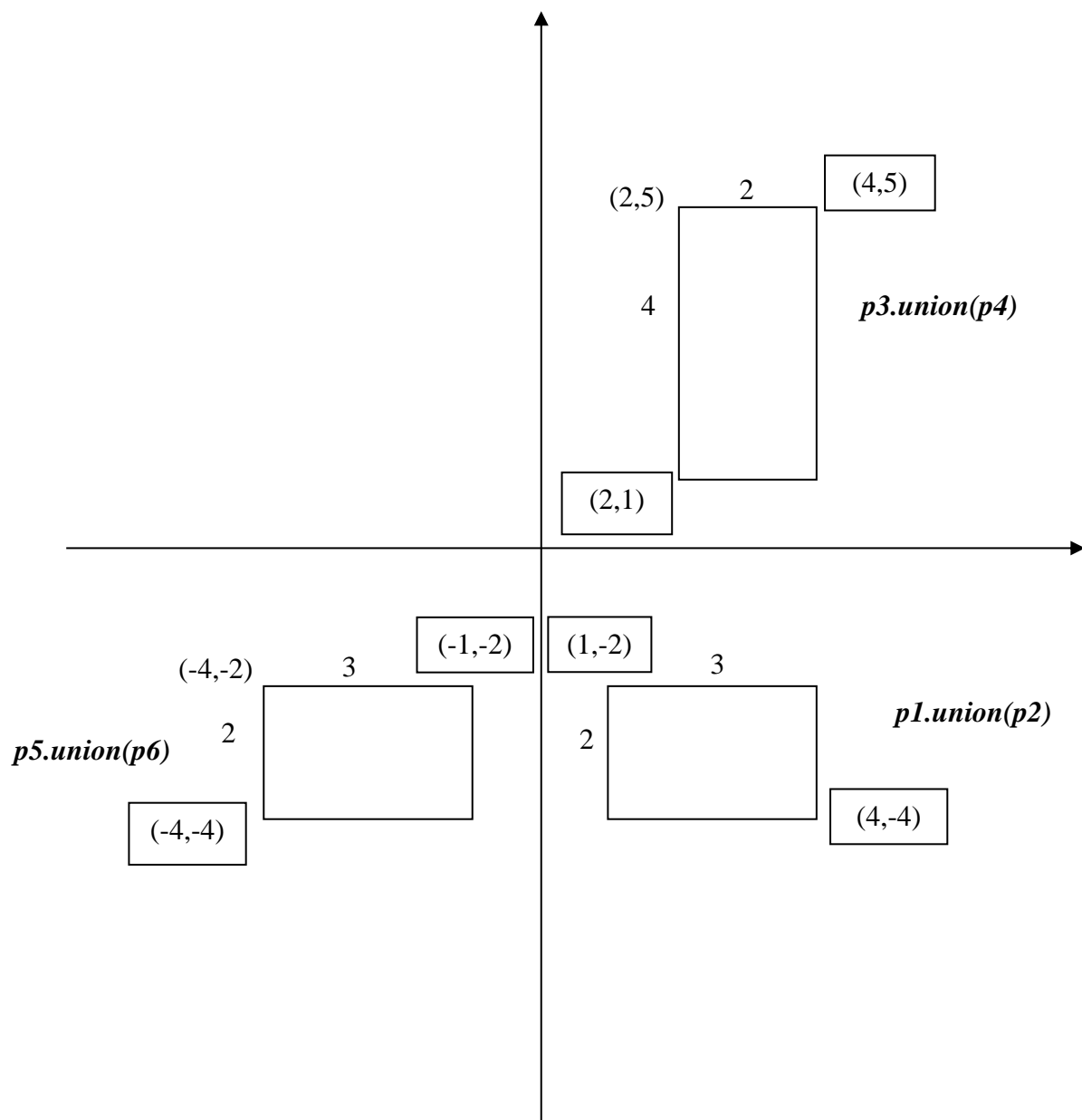
```
            hauteur: 4
```

```

Point p5 = new Point(-1,-2);
Point p6 = new Point(-4,-4);
System.out.println(p5.union(p6));
    ⇒ affiche à l'écran:
        point d'encrage (-4,-2)
        largeur: 3
        hauteur: 2
}
}

```

Les rectangles construits dans la classe *Principal* sont illustrés sur le graphique ci-dessous.



Chapitre 2 : Protections (Information hiding)

2.1. Information hiding : protections *private* et *public*

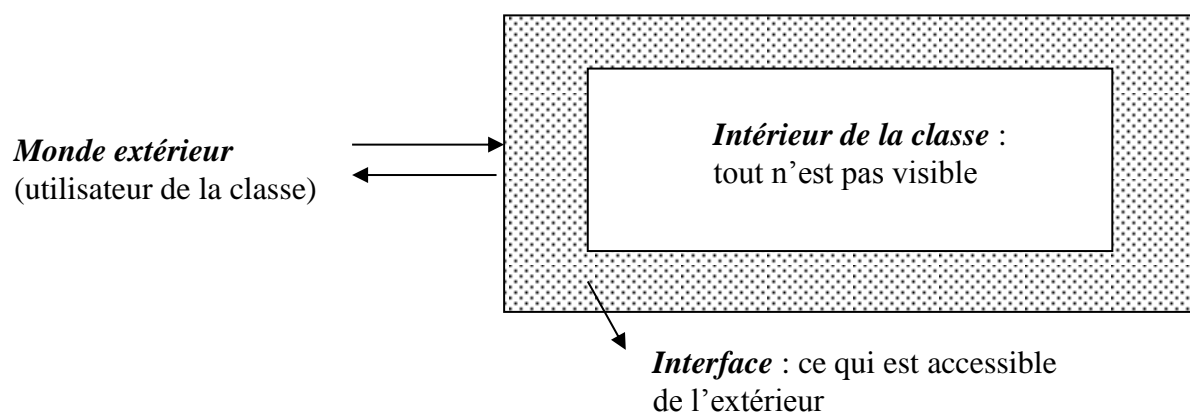
Un des objectifs de la programmation O.O. est de diminuer le temps de programmation, et ce, en réutilisant le plus possible des composants existants. Un programmeur peut par exemple réutiliser des classes qu'il a créées précédemment ou des classes créées par d'autres programmeurs.

Pour rappel, un des objectifs du cours de langage de programmation est d'arriver à réaliser des applications à interface (utilisateur) graphique. Pour réaliser de tels programmes, inutile de réinventer la roue : bon nombre de composants (classes et autres) écrits par d'autres concepteurs sont disponibles.

Pour arriver à les utiliser, il faut comprendre ce qu'on peut réutiliser dans ces composants. En effet, le concepteur (programmeur) d'un composant peut décider de cacher certaines informations qui sont internes au composant et d'en rendre publiques d'autres. Seules les informations rendues publiques sont accessibles, et donc réutilisables, 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 méthodes et aux constructeurs**. En définissant les protections ou types d'accès, il délimite l'interface auquel le monde extérieur aura accès.

Plusieurs types d'accès sont possibles.

En voici deux premiers :

- **public** : accessible par le monde extérieur ;
- **private** : accessible seulement de 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, **toutes les variables d'instances d'une classe sont déclarées *private***. Selon ces conventions, aucun utilisateur d'une classe créée par un autre concepteur n'a donc accès aux variables d'instance de cette 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.

Le principe qui consiste à cacher certaines informations porte le nom anglais de *information hiding*.

2.2. Méthodes publiques d'accès aux variables d'instance privées : *gettors* et *settors*

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, le nom des méthodes permettant l'accès à des variables d'instance (déclarées *private*) commence soit par *get*, soit par *set* suivi du nom de la variable d'instance:

- les méthodes permettant l'accès en **lecture** portent un nom commençant par **get** ;
- les méthodes permettant l'accès en **écriture** portent un nom commençant par **set**.

Les méthodes d'accès en lecture sont appelées **gettors** en anglais, et les méthodes d'accès en écriture sont appelées **settors**.

Prenons comme exemple la classe Rectangle. Appliquons le principe de l'**information hiding** : déclarons toutes les variables d'instance **private**. Or, il paraît intéressant de permettre à l'utilisateur d'une telle classe de pouvoir obtenir les coordonnées x et y du point d'encrage ainsi que la largeur et la hauteur de tout rectangle qu'il aurait créé. De même, il paraît intéressant de permettre à l'utilisateur de modifier quand il le désire ces caractéristiques. Dans la classe Rectangle sont donc prévues 4 méthodes de type *gettors* et 4 méthodes de type *settors*, toutes étant déclarées **public**.

*Exemple :***public** class **Rectangle** {**private** int coordX, coordY, largeur, hauteur;**public** Rectangle(int x, int y, int largeurDonnee, int hauteurDonnee)

```
{ coordX = x;
  coordY = y;
  largeur = largeurDonnee;
  hauteur = hauteurDonnee; }
```

private Rectangle(int largeurDonnee, int hauteurDonnee)**{ this(0,0,largeurDonnee,hauteurDonnee); }****public** Rectangle()**{ this(1,1); }****public** int **getCoordX**()

{ return coordX; }

public int **getCoordY**()

{ return coordY; }

public int **getLargeur**()

{ return largeur; }

public int **getHauteur**()

{ return hauteur; }

public void **setCoordX**(int newX)

{ coordX = newX; }

public void **setCoordY**(int newY)

{ coordY = newY; }

public void **setLargeur**(int newLargeur)

{ largeur = newLargeur; }

public void **setHauteur**(int newHauteur)

{ hauteur = newHauteur; }

public int surface()

{ return largeur * hauteur; }

public int perimetre()

{ return (2*largeur) + (2*hauteur); }

private void elargir (int augmentation)

{ largeur += augmentation; }

private void modifierHauteur (int augmentation)

{ hauteur += augmentation; }

public void deplacerEn (int newX, int newY)

{ coordX = newX; }

{ coordY = newY; }

```
public String toString( ) { return "point d'encrage : (" + coordX + ", " + coordY + ")"
                           "\nlargeur : " + largeur + "\nhauteur : " + hauteur ; }
```

}

*surcharge des
constructeurs*

public getters

public setters

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 (ex : coordX = newX ;)

Il en va de même pour l'appel des méthodes et constructeurs. Le constructeur *Rectangle()* sans argument fait appel au constructeur à deux arguments via l'instruction **this(1,1)**, ce constructeur à deux arguments étant pourtant déclaré **private**.

```
public class Principal
```

```
{ public static void main(String[ ] args)
```

```
{   Rectangle r = new Rectangle(0,0,4,2);           /* OK car constructeur public */
```

```
    System.out.println("largeur de r : " + r.largeur+ "\nhauteur de r : " + r.hauteur);  
                                /* pas OK car largeur et hauteur déclarées private */
```

```
    System.out.println("largeur de r : " + r.getLargeur( )+ "\nhauteur de r : " +  
                        r.getHauteur( ));  
                                /* OK car getLargeur( ) et getHauteur( ) déclarées public */
```

```
    r.coordX = 1;                /* pas OK car coordX déclaré private */  
    r.coordY = 2;                /* pas OK car coordY déclaré private */  
    r.largeur = 3;               /* pas OK car largeur déclaré private */  
    r.hauteur = 4 ;              /* pas OK car hauteur déclaré private */
```

```
    r.setCoordX(1);              /* OK car setCoordX(...) déclaré public */  
    r.setCoordY(2);              /* OK car setCoordY(...) déclaré public */  
    r.setLargeur(3);             /* OK car setLargeur(...) déclaré public */  
    r.setHauteur(4);             /* OK car setHauteur(...) déclaré public */
```

```
    System.out.println("surface de r : " + r) ;      /* OK car toString( ) est public */
```

```
    System.out.println("surface de r : " + r.surface( ));  
                                /* OK car surface( ) déclaré public */
```

```
    System.out.println("surface de r : " + r.perimetre( ));  
                                /* OK car perimetre( ) déclaré public */
```

```
    r.elargir(5);                /* pas OK car elargir(...) déclaré private */
```

```
    r.modifierHauteur(10);       /* pas OK car modifierHauteur(...) déclaré private */
```

```
    Rectangle r2 = new Rectangle(1,2); /* pas OK car ce constructeur est déclaré private */
```

```
    Rectangle r3 = new Rectangle( );    /* OK car ce constructeur est déclaré public */
```

```
    }  
}
```

Dans le code précédent, le fait d'accéder directement en lecture ou en écriture aux variables d'instance *coordX*, *coordY*, *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 **public** *getCoordX()*, *getCoordY()*, *getLargeur()* et *getHauteur()*, et y accéder en écriture via les setters **public** *setCoordX(...)*, *setCoordY(...)*, *setLargeur(...)* et *setHauteur(...)*.

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

Il faut noter que le constructeur à 4 arguments est déclaré **public** : on peut donc créer des occurrences de rectangle en y faisant appel dans la classe *Principal*.

Par contre, le constructeur à deux arguments ne peut être appelé dans la classe *Principal*. Il est, en effet, déclaré **private**.

Quant au constructeur sans argument, il peut être appelé dans la classe *Principal* car déclaré **public**, et ce, *même s'il fait lui-même appel au constructeur à deux arguments qui lui est déclaré private!*

Notons au passage que la protection **public** a été attribuée aux classes également. Cela a pour effet de les rendre accessibles 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 !**

Certaines signatures de méthodes sont immuables : ex : la méthode *toString()* qui doit impérativement débiter sa signature par la protection **public**.

En conclusion :

- ① Le concepteur d'une classe décide ce qu'il cache et ce qu'il rend visible (càd 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 public 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 une valeur du type de la variable d'instance accédée.
- ⑦ *Par convention*, les méthodes public 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.

2.3. Notion de package (+ import)

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

Il est cependant 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.util.GregorianCalendar;*

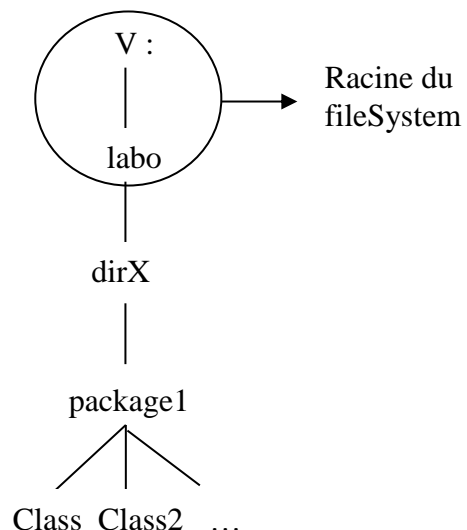
importe la classe *GregorianCalendar* qui se trouve dans le package *util* du répertoire *java*, lui-même situé dans le répertoire (racine) contenant tous les répertoires des classes prédéfinies du langage java. L'adresse du répertoire racine des classes prédéfinies du langage java est connue de la machine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

2. soit la structure de répertoires suivante :

Si la racine du file system associé au projet est *v:\labo*, et si le package à importer s'appelle *package1* et qu'il se trouve dans le sous-répertoire *dirX* du répertoire *labo*, l'instruction d'import à écrire dans n'importe quel programme du même projet est :

import dirX.package1.* ;

L'adresse du package est ici donnée en adressage relatif : on ne précise que l'adresse du chemin à partir de la racine du filesystem 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 rectanglePack;

```
public class Rectangle
{ private int coordX, coordY, largeur, hauteur;
  ...
}
```

package autrePack ;

import rectanglePack.* ; */* importe toutes les classes du package rectanglePack */*

```
public class Principal
{ public static void main(String[] args)
  { Rectangle r ; /* OK car la classe Rectangle est déclarée public */

    r = new Rectangle(1,2,4,3) ; /* OK car ce constructeur est déclaré public */

    System.out.println("surface de r : " + r) ; /* OK car toString( ) est public */

    System.out.println("largeur de r : " + r.largeur+ "\nhauteur de r : " + r.hauteur);
    /* pas OK car largeur et hauteur déclarées private */

    System.out.println("largeur de r : " + r.getLargeur( )+ "\nhauteur de r : " +
      r.getHauteur( ));
    /* OK car getLargeur( ) et getHauteur( ) déclarées public */

    r.coordX = 1; /* pas OK car coordX déclaré private */

    r.setCoordX(1); /* OK car setCoordX(...) déclaré public */

    r.elargir(5); /* pas OK car elargir(...) déclaré private */

    System.out.println("surface de r : " + r.perimetre( ));
    /* OK car perimetre( ) déclaré public */
  }
}
```

Les classes *Rectangle* et *Principal* sont placées dans des packages différents : la classe *Rectangle* est déclarée dans le package **rectanglePack** et la classe *Principal* est déclarée dans le package **autrePack** .

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 **rectanglePack**, à savoir la classe *Principal*.

2.4. 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) :

Protection de type **private** (la plus restrictive)



Protection de type **package**



Protection de type **public** (la plus permissive)

Exemple :

package forme ;

class Rectangle

{ int largeur, hauteur ;

Rectangle (int larg, int haut) { largeur = larg; hauteur = haut ;}

int surface () { return largeur * hauteur ; }

void elargir (int a) { largeur += a ; }

}

Protection de type package

package forme ;

public class Principal

{ public static void main (String[] args)

{ Rectangle r;

r = new Rectangle (3, 5) ;

System.out.println ("la hauteur de r est: " + r.hauteur + " cm");

System.out.println ("la surface de r est: " + r.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 ont une protection de type package n'est d'aucune utilité: étant dans un autre package, on ne pourra y avoir accès.

Chapitre 3 : Héritage

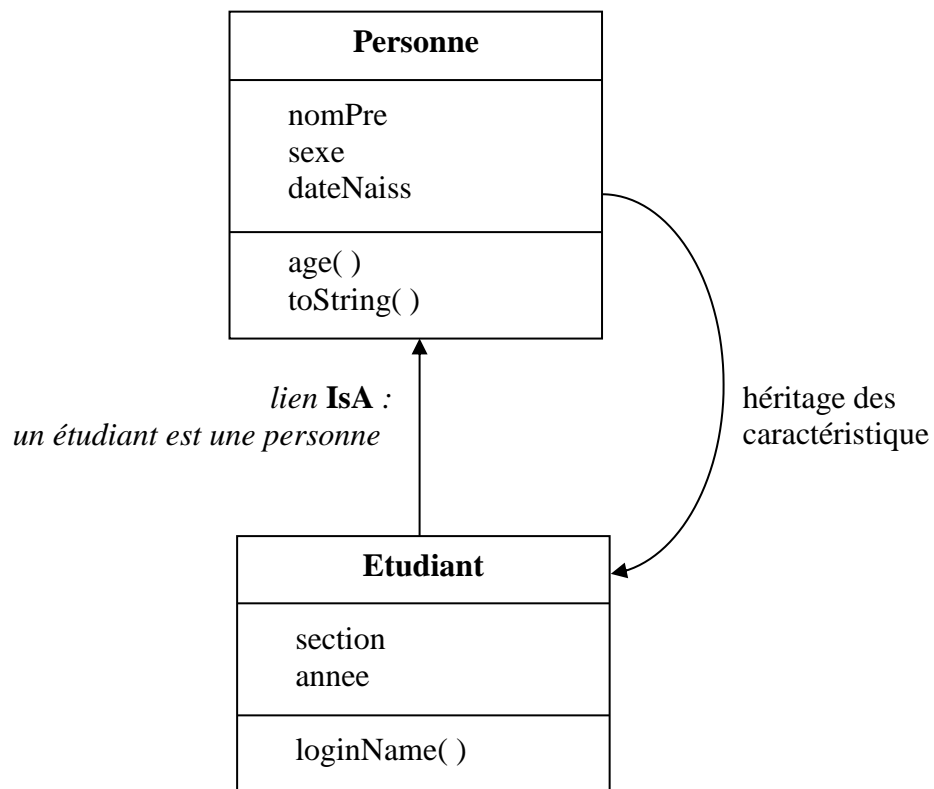
Pour rappel, un des objectifs de la programmation O.O. est de diminuer le temps de programmation en réutilisant le plus possible des composants existants. Un des avantages de la programmation O.O., c'est que les composants réutilisés peuvent être adaptés s'il y a lieu. Il est fondamental de comprendre comment on peut réutiliser des composants existants et comment les adapter, notamment si l'on désire écrire des applications à interface utilisateur graphique. En effet, bon nombre de composants graphiques (fenêtre, boutons, menus, ...) existent et sont à disposition du programmeur. Il n'y a plus qu'à les réutiliser et à les adapter à l'application à créer.

3.1. Sous-classes (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 sexe, la date de naissance et l'âge d'une personne. Ces notions restent d'actualité pour un étudiant, mais un étudiant présente des caractéristiques qui lui sont propres, comme sa section (ex : informatique, comptabilité, droit, ...), son année (ex : 1,2,3,...) et le login name qu'il reçoit (ex : TI2LeroyF). 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*.



```
class Personne
{ String nomPre;
  char sexe;
  GregorianCalendar dateNaiss;

  Personne(String nP, char x, int an, int mois, int jour)
  { nomPre= nP;
    sexe = x;
    dateNaiss = new GregorianCalendar (an, mois, jour);}

  int age( ) { ... }

  public String toString( ) {return "La personne " + nomPre;}
}

class Etudiant extends Personne    // dans même package que Personne
{ String section;
  int annee;      caractéristiques de la personne   caractéristiques propres à l'étudiant

  Etudiant (String nP, char x, int a, int m, int j, String s, int aE)
  { super(nP, x, a, m, j);
    section = s;
    annee = aE;}

  String loginName( ) { ...}
}
```

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*.

Le lien ***IsA*** (de l'anglais *is a*, traduit par *est un*) 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, ***nomPre***, ***sexe*** et ***dateNaissance*** en plus des variables d'instance propres à la classe *Etudiant*, à savoir, ***section*** et ***annee***.

La classe *Etudiant* propose un constructeur permettant d'initialiser ces 5 variables d'instance. 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***(nP, x, a, m, j) » signifie que l'on appelle le constructeur de la super-classe avec ces arguments. Cette instruction est donc équivalente à :

```
    nomPre= nP;
    sexe = x;
    dateNaiss = new GregorianCalendar (a, m, j) ;
```

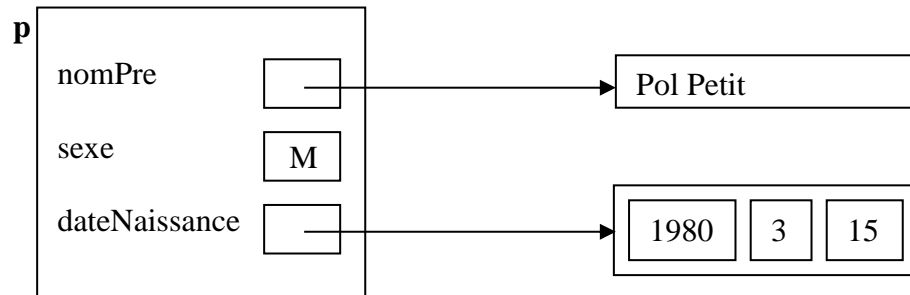
Attention, l'instruction super(...) doit toujours être placée en première position dans le constructeur de la sous-classe !

3.2. 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* (ex : la méthode *age()*).

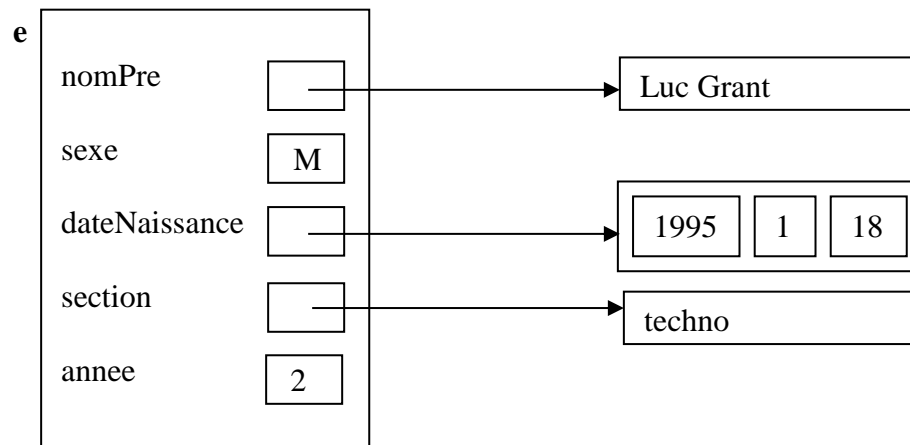
```
class Principal      // dans même package que Personne et Etudiant
{ public static void main(String[] args)
  { Personne p = new Personne("Pol Petit",'M',1980,3,15);
```

En mémoire :



```
Etudiant e = new Etudiant("Luc Grant",'M',1995,1,18, "techno",2);
```

En mémoire :



```

System.out.println(e.nomPre) ; // Accessible car nomPre a la protection package
    ⇒ affiche à l'écran: Luc Grant
if (e.sexe == 'F') // Accessible car sexe a la protection package
    System.out.println (" Sois la bienvenue") ;
else System.out.println (" Sois le bienvenu");
    ⇒ affiche à l'écran: Sois le bienvenu
System.out.println(e); // Appel implicite à toString( ) public hérité de Personne
    ⇒ affiche à l'écran: La personne Luc Grant
System.out.println(e.age()); //Appel à age( ) (protection package) de Personne
    ⇒ affiche à l'écran: 20
}

```

Les variables d'instance *nomPre*, *sexe* et *dateNaissance* d'un étudiant sont héritées de la classe *Personne*.

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.

Ex : **Personne** p2 = new **Etudiant**("Marie Flore",'F',1995,4,28, "techno",2) ;
 System.out.println(p2.age()); // appel possible des méthodes de la classe *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.

3.3. Redéfinition de méthodes

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 de **rajouter 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 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 (cfr méthode *toString()* de la classe *EtudiantInfo*).

class **Etudiant extends Personne**

```
{ String section;
  int annee;
```

```
    Etudiant(String nP, char x, int a, int m, int j, String s, int aE)
    {super(nP, x, a, m, j);
      section = s;
      annee = aE;}
}
```

```
String loginName( ) { ...}  
  
public String toString( )  
    {return nomPre + " (" + this.age( ) + "ans) inscrit en " + section +  
      "\n" + "a reçu le login name: " + this.loginName( );  
    }  
}
```

La méthode *toString*() de la classe *Etudiant* écrase la méthode *toString*() héritée de la classe *Personne*.

```
class Principal  
{ public static void main(String[ ] args)  
    { Personne p = new Personne("Pol Petit", 'M', 1980, 3, 15);  
      System.out.println(p);  
        ⇒ affiche à l'écran: La personne Pol Petit  
  
      Etudiant e = new Etudiant("Luc Grant", 'M', 1995, 1, 18, "techno", 2);  
      System.out.println(e);  
        ⇒ affiche à l'écran: Luc Grant (20 ans) inscrit en techno  
                           a reçu le login name : TI2GrantL  
    }  
}
```

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 ? Le langage Java commence toujours sa recherche 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*.

3.4. Hiérarchie d'héritage

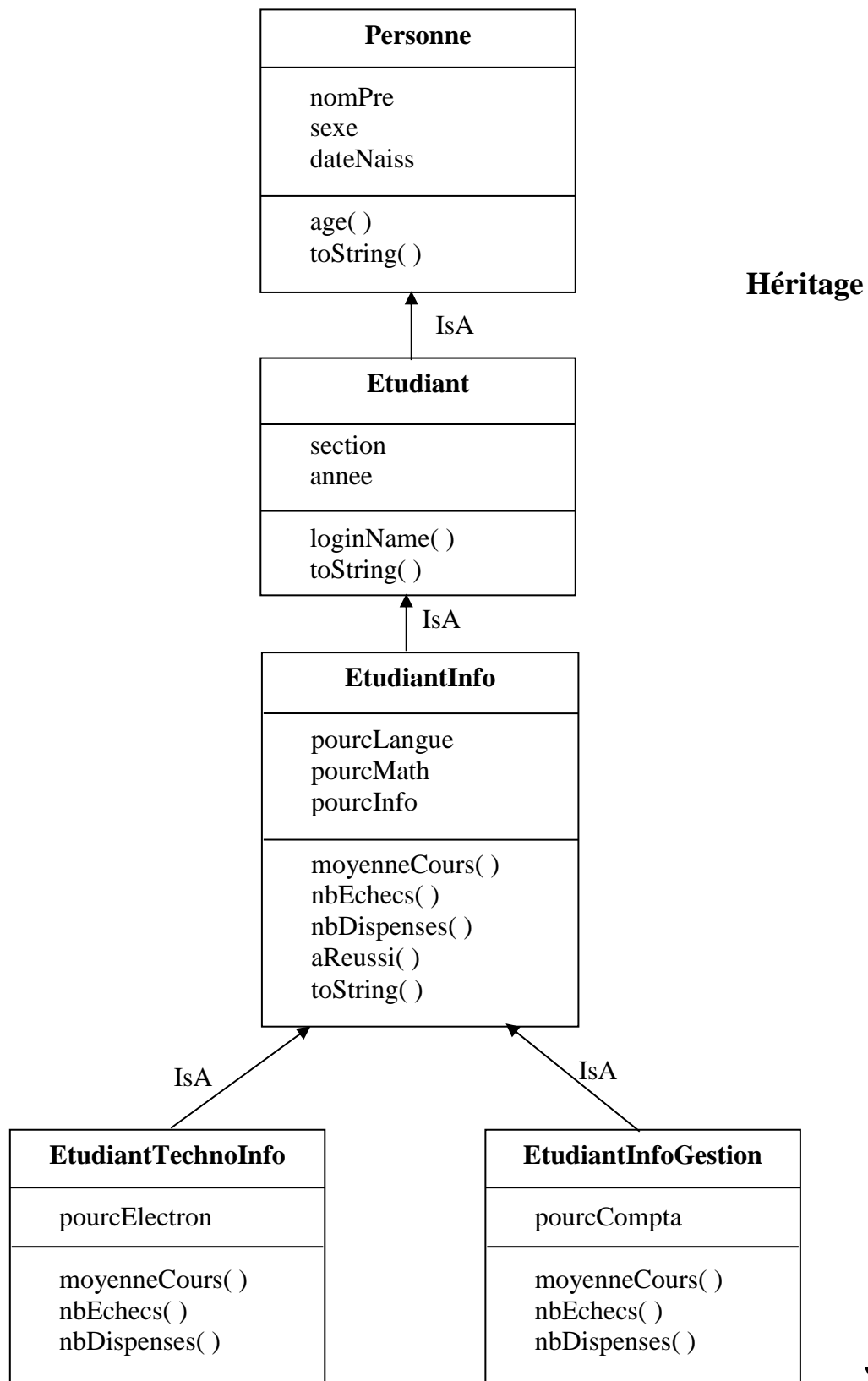
Le mécanisme d'héritage est applicable à plus de deux classes. On peut ainsi construire des **hiérarchies de classes**.

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

- Les classes *Personne* et *Etudiant* déjà rencontrées.
- La classe *EtudiantInfo* permettant de gérer des étudiants inscrits en informatique : les pourcentages dans les cours de langues, en mathématique et en informatique sont enregistrés sous forme de variables d'instance. Le calcul de la moyenne des cours, du nombre d'échecs, du nombre de dispenses et de la réussite de tout étudiant en informatique est proposé sous forme de méthodes.
- La classe *EtudiantTechnoInfo* permettant de gérer des étudiants inscrits en technologie de l'informatique : le pourcentage dans le cours d'électronique est enregistré sous forme de variable d'instance. Les méthodes de calcul de la moyenne des cours, du nombre d'échecs, du nombre de dispenses et de la réussite de tout étudiant en technologie de l'informatique sont **redéfinies pour tenir compte du pourcentage obtenu dans le cours d'électronique**.
- La classe *EtudiantInfoGestion* permettant de gérer des étudiants inscrits en informatique de gestion : le pourcentage dans le cours de comptabilité est enregistré sous forme de variable d'instance. Les méthodes de calcul de la moyenne des cours, du nombre d'échecs, du nombre de dispenses et de la réussite de tout étudiant en informatique de gestion sont **redéfinies pour tenir compte du pourcentage obtenu dans le cours de comptabilité**. Notons que le pourcentage obtenu dans le cours d'informatique a un **poids double** par rapport aux autres cours pour un étudiant inscrit en informatique de gestion.

Le mécanisme d'héritage est implémenté 4 fois :

- entre la classe *Etudiant* et la classe *Personne*,
- entre la classe *EtudiantInfo* et *Etudiant*,
- entre la classe *EtudiantTechnoInfo* et *EtudiantInfo*,
- entre la classe *EtudiantInfoGestion* et *EtudiantInfo*.



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

class **EtudiantInfo** extends Etudiant

{ float **pourcLangue**, **pourcMath**, **pourcInfo**;

EtudiantInfo(String nP, char x, int a, int m, int j, String s, int aE, float pL, float pM, float pI)
 { super(nP, x, a, m, j, s, aE);
 pourcLangue = pL;
 pourcMath = pM;
 pourcInfo = pI ; }

float **moyenneCours**()
 { return (pourcLangue + pourcMath + pourcInfo)/3; }

int **nbEchecs**()
 { int n = 0;
 if (pourcLangue <=10) n ++;
 if (pourcMath <=10) n ++;
 if (pourcInfo <=10) n ++;
 return n; }

int **nbDispenses**()
 { int n = 0;
 if (pourcLangue >=12) n ++;
 if (pourcMath >=12) n ++;
 if (pourcInfo >=12) n ++;
 return n; }

boolean **aReussi**() */* par facilité, les conditions de réussite sont simplifiées:
 pas de deliberation possible! */*
 { if (this.**moyenneCours**() >= 60 && this.**nbEchecs**() == 0) return true;
 else return false; }

public String **toString**() *appel à la méthode héritée*
 { return super.toString() + " et\n" + (this.**aReussi**()?"a réussi ":"a échoué ")
 + " avec une moyenne de " + this.**moyenneCours**() + "%"; }
 }

Dans la section 3.3, nous avons vu 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*.

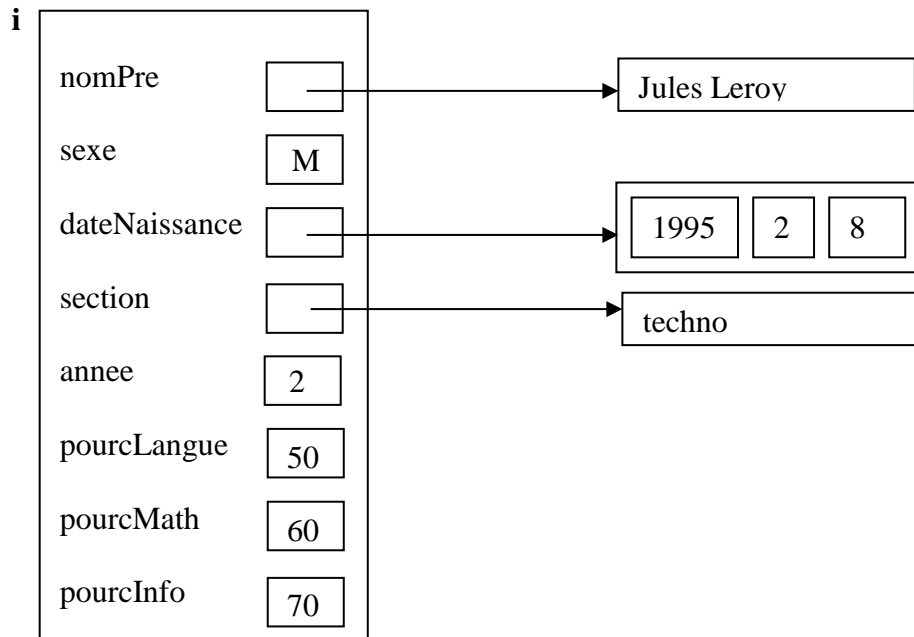
Ici aussi, la méthode *toString*() de la classe *EtudiantInfo* redéfinit la méthode *toString*() héritée de la classe *Etudiant*. Mais, le code de la méthode *toString*() de la classe *EtudiantInfo* ne remplace pas purement et simplement le code de la méthode *toString*() héritée. Il **récupère et étend**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

class **Principal**

```
{ public static void main(String[ ] args)
    {EtudiantInfo i = new EtudiantInfo("Jules Leroy",'M',1995,2,8, "techno",2, 50, 60, 70) ;
```

En mémoire:



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

⇒ affiche à l'écran: *Jules Leroy (20 ans) inscrit en techno* (1)
a reçu le login name : TI2LeroyJ et (2)
a réussi avec une moyenne de 60% (3)

```
}  
}
```

Les chaînes de caractères (1) et (2) affichées à l'écran proviennent de l'appel à la méthode `toString()` héritée de la classe *Etudiant*. La chaîne de caractères (3) a été ajoutée par la méthode `toString()` spécifique à la classe *EtudiantInfo*.

class **EtudiantTechnoInfo** extends EtudiantInfo

```
{ float pourcElectron;
```

```
EtudiantTechnoInfo(String nP, char x, int a, int m, int j, int aE,  
int pL, int pM, int pI, int pE)
```

```
{super(nP, x, a, m, j,"techno", aE, pL, pM ,pI); /* inscrit d'office en techno */  
pourcElectron = pE;}
```

```
float moyenneCours( ) /* remplacement de la méthode moyenneCours( ) héritée */
{ return (pourcLangue + pourcMath + pourcInfo + pourcElectron)/4; }

int nbEchecs( )
{ int n = super.nbEchecs( ); /* appel à la méthode nbEchecs( ) héritée */
  if (pourcElectron <=10) n ++;
  return n; }

int nbDispenses( )
{ int n = super.nbDispenses( ); /* appel à la méthode nbDispenses( ) héritée */
  if (pourcElectron >=12) n ++;
  return n; }
}
```

La classe *EtudiantTechnoInfo* présente aussi un exemple de redéfinition de méthode qui remplace totalement le code de la méthode héritée (cfr méthode *moyenneCours*()) et deux exemples de redéfinition de méthode qui récupèrent et étendent le code de la méthode héritée (cfr méthodes *nbEchecs*() et *nbDispenses*()).

class **EtudiantInfoGestion** **extends EtudiantInfo**

```
{ float pourcCompta;
```

```
    EtudiantInfoGestion (String nP, char x, int a, int m, int j, int aE,
                          int pL, int pM, int pI, int pC)
    { super(nP, x, a, m, j, "gestion", aE, pL, pM ,pI);
      pourcCompta = pC; }
```

```
float moyenneCours( ) /* remplacement de la méthode moyenneCours( ) héritée */
{ return (pourcLangue + pourcMath + (pourcInfo * 2) + pourcCompta)/5; }
```

```
int nbEchecs( )
{ int n = super.nbEchecs( ); /* appel à la méthode nbEchecs( ) héritée */
  if (pourcCompta <=10) n ++;
  return n; }
```

```
int nbDispenses( )
{ int n = super.nbDispenses( ); /* appel à la méthode nbDispenses( ) héritée */
  if (pourcCompta >=12) n ++;
  return n ; }
```

```
}
```

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(...) ...

3.5. Polymorphisme

Etant 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 le langage 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, càd, 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*, java l'exécute. 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.

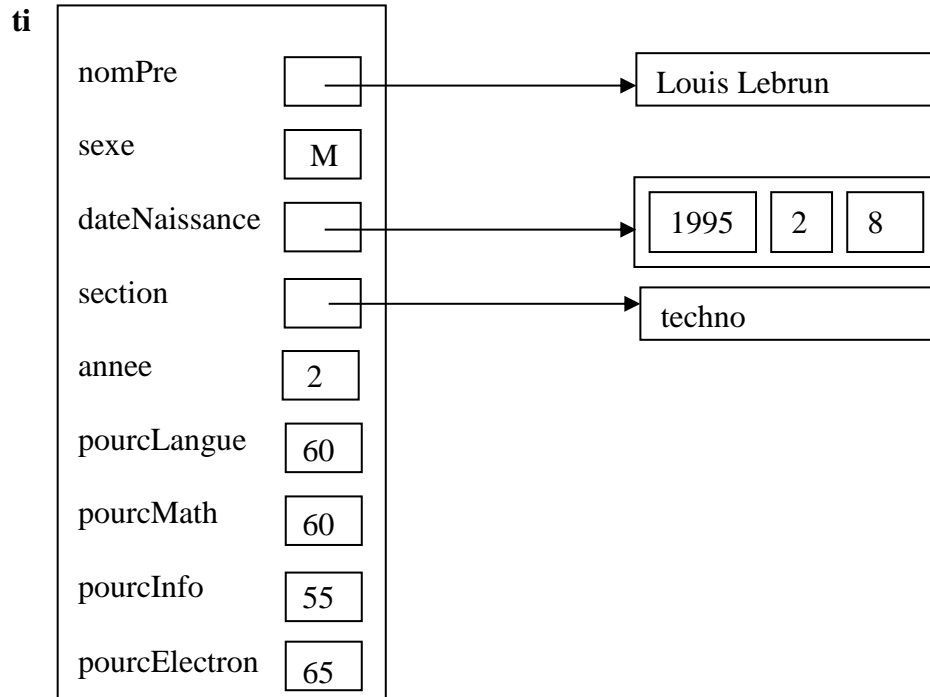
class **Principal**

```
{ public static void main(String[ ] args)
```

```
{ EtudiantTechnoInfo ti;
```

```
ti = new EtudiantTechnoInfo("Louis Lebrun", 'M', 1995, 2, 8, 2, 60, 60, 55, 65) ;
```

En mémoire:



```

        System.out.println(ti);
    }
}

```

Notons que la variable d'instance *section* de l'étudiant existe quand-même en mémoire, même si elle n'apparaît pas dans le constructeur.

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

```

public String toString()
{
    return super.toString() + " et\n" + (this.aReussi()?"a réussi ":"a échoué ")
        + " avec une moyenne de " + this.moyenneCours() + "%";
}

```

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

La chaîne de caractères retournée par cet appel est :

*Louis Lebrun (20 ans) inscrit en techno
a reçu le login name : TI2LebrunL*

Ensuite, la méthode *aReussi()* est appelée sur l'objet courant **ti**. Java tente de nouveau de rechercher cette méthode à partir de la classe correspondant à **ti**, c'est-à-dire dans la classe *EtudiantTechnoInfo*. La méthode *aReussi()* ne s'y trouve pas. Java remonte donc la hiérarchie d'héritage à la recherche de cette méthode. La méthode *aReussi()* se trouve dans la super-classe : la classe *EtudiantInfo*.

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

```
boolean aReussi( )  
    { if (this.moyenneCours( ) >= 60 && this.nbEchecs( ) == 0) return true;  
      else return false; }
```

La méthode *moyenneCours*() doit à son tour être exécutée. Notons que la méthode *moyenneCours*() est définie dans la classe *EtudiantInfo*. Mais, comme convenu, java recherche cette méthode en partant d'abord de la classe correspondant à l'objet *ti*. La recherche débute donc dans la classe *EtudiantTechnoInfo*. Or, la méthode *moyenneCours*() s'y trouve redéfinie. C'est donc ce code-là qui sera exécuté et en aucun cas, la méthode *moyenneCours*() qui se trouve dans la classe *EtudiantInfo*, **même si c'est la méthode *aReussi*() de cette classe-là qui a été exécutée.**

Code de la méthode *moyenneCours*() de la classe *EtudiantTechnoInfo* qui **sera exécuté** :

```
float moyenneCours( )  
    { return (pourcLangue + pourcMath + pourcInfo + pourcElectron)/4; }
```

Code de la méthode *moyenneCours*() de la classe *EtudiantInfo* qui ne sera **pas exécuté** :

```
float moyenneCours( )  
    { return (pourcLangue + pourcMath + pourcInfo)/3; }
```

Le raisonnement est le même pour la méthode *nbEchecs*(). C'est la méthode de la classe *EtudiantTechnoInfo* qui sera exécuté et non celui de la classe *EtudiantInfo*.

Code de la méthode *nbEchecs*() de la classe *EtudiantTechnoInfo* qui **sera exécuté** :

```
int nbEchecs( )  
    { int n = super.nbEchecs( );           /* appel à la méthode nbEchecs( ) héritée */  
      if (pourcElectron <=10) n ++;  
      return n; }
```

Code de la méthode *nbEchecs*() de la classe *EtudiantInfo* qui **sera appelée** :

```
int nbEchecs( )  
    { int n = 0;  
      if (pourcLangue <=10) n ++;  
      if (pourcMath <=10) n ++;  
      if (pourcInfo <=10) n ++;  
      return n; }
```

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.

```
class Principal
```

```
{ public static void main(String[ ] args)
```

```
{ EtudiantTechnoInfo ti;
```

```
ti = new EtudiantTechnoInfo("Louis Lebrun ", 'M', 1995, 2, 25, 2, 60, 60, 55, 65) ;
```

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

```
⇒ affiche à l'écran:
```

```
Louis Lebrun (20 ans) inscrit en techno
```

```
a reçu le login name : TI2LebrunL et
```

```
a réussi avec une moyenne de 60%
```

```
EtudiantInfoGestion ig;
```

```
ig = new EtudiantInfoGestion("Pol Lenoir", 'M', 1995, 1, 14, 2, 60, 60, 55, 65) ;
```

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

```
⇒ affiche à l'écran:
```

```
Pol Lenoir (20 ans) inscrit en gestion
```

```
a reçu le login name : IG2LenoirP et
```

```
a échoué avec une moyenne de 59%
```

```
}
```

```
}
```

Le fait que des étudiants inscrits dans deux sections différentes n'obtiennent pas le même résultat alors qu'ils ont les mêmes pourcentages, prouve bien que les bonnes méthodes ont été exécutées en fonction du type de l'objet sur lequel elles ont été appelées. L'étudiant Pol Lenoir a échoué car son pourcentage en informatique (55%) intervient avec un **poids double** dans le calcul de sa moyenne (cfr méthode *moyenneCours()* de la classe *EtudiantInfoGestion*).

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. Il est impossible de compiler le code de l'exemple précédent, car ce n'est qu'à l'exécution qu'on sait quelles méthodes *moyenneCours()* et *nbEchecs()* exécuter en fonction du type de l'objet (*EtudiantTechnoInfo* ou *EtudiantInfoGestion*).

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 (càd celle qui se trouve le plus bas possible dans la hiérarchie d'héritage).

② On parle de polymorphisme quand ce n'est qu'à l'**exécution** qu'on sait déterminer **quelle méthode exécuter** quand la même méthode (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

3.6. Protection de type *protected*

Les trois types de protection déjà rencontrées 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 (càd protection de type *package*).

Exemples : *public* MyClass { ... }

ou MyClass { ... } \Rightarrow protection de *type package*

La protection de type *protected* associée à une variable d'instance ou une méthode 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 de la hiérarchie d'héritage analysées dans la section 3.4. mais répartissons-les dans deux packages différents.

Le **packageA** comprend les classes *Personne*, *Etudiant* et *EtudiantInfo*.

Le **packageB** comprend les classes *EtudiantTechnoInfo*, *EtudiantInfoGestion* et la classe *Principal*.

Déclarons ensuite *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 nomPre; \Leftrightarrow variables d'instance

protected char sexe;

protected GregorianCalendar dateNaiss;

protected **Personne**(...) { ... } \Leftrightarrow constructeur

protected int age() { ... } \Leftrightarrow méthodes

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

}

package **packageA**;

public class **Etudiant** **extends Personne**

{ **protected** String section; ⇔ variables d'instance
protected int annee;

protected Etudiant(...) {...} ⇔ constructeur

protected String loginName() {...} ⇔ méthodes

public String toString() {...}
 }

package **packageA**;

public class **EtudiantInfo** **extends Etudiant**

{ **protected** float pourcLangue, pourcMath, pourcInfo; ⇔ variables d'instance

protected EtudiantInfo(...) {...} ⇔ constructeur

protected float moyenneCours() {...} ⇔ méthodes

protected int nbEchecs() {...}

protected int nbDispenses() {...}

protected boolean aReussi() {...}

public String toString() {...}

}

Les trois classes du packageB importent toutes les classes du packageA pour pouvoir les utiliser (cfr l'instruction : **import packageA.*** ;).

La classe *EtudiantInfoGestion* peut être déclarée **sous-classe** de la classe *EtudiantInfo*, 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.* ;

class **EtudiantInfoGestion** **extends EtudiantInfo**

{ float pourcCompta;

EtudiantInfoGestion (String nP, char x, int a, int m, int j, int aE,
 int pL, int pM, int pI, int pC)

{ **super**(nP, x, a, m, j, "gestion", aE, pL, pM, pI); (1)
 pourcCompta = pC; }

protected float moyenneCours() (2)

{return (***pourcLangue*** + ***pourcMath*** + (***pourcInfo*** * 2) + pourcCompta)/5; } (3)


```

protected int nbEchecs( )                                (4)
{ int n = super.nbEchecs( );                             (5)
  if (pourcCompta <=10) n ++;
  return n; }
...
}

```

Le constructeur de la classe *EtudiantInfoGestion* peut faire appel au constructeur hérité via l'instruction **super(...)** (cfr (1)), 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 *moyenneCours()* et *nbEchecs()* doivent être déclarées *protected* (cfr (2) et (4)). 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 équivaudrait à la protection de type *package* (aucun mot clé). La protection de type *package* est plus restrictive que la protection *protected*.

Les variables d'instance *pourcLangue*, *pourcMath* et *pourcInfo* sont accessibles car déclarées *protected* dans la super-classe, et donc accessibles par les sous-classes même situées dans un autre package (cfr (3)).

L'instruction **super.nbEchecs()** (cfr (5)) peut être exécutée. Il s'agit de l'appel à la méthode *nbEchec()* héritée. Or, celle-ci est déclarée *protected* dans la super-classe. Elle est donc accessible .

```

package packageB;
import packageA.* ;

```

```

class Principal

```

```

{ public static void main(String[] args)

    { Etudiant e ;                                (6)
      e = new Etudiant("Luc Grant",'M', 1995, 7, 20,"techno",2); →☹ (7)

      EtudiantInfoGestion i ;                      (8)

      i = new EtudiantInfoGestion("Louis Petit",'M',1995,5,15,2,65,60,55,60); (9)

      System.out.println( i );                     (10)

      System.out.println(i.age( ));                 →☹ (11)

      System.out.println(i.moyenneCours( ) );      (12)
    }
}

```

Un objet de type *Etudiant* peut être déclaré dans la classe *Principal* (cfr (6)), 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* (cfr (7)), car ce constructeur est déclaré *protected* et **la classe *Principal* n'est pas une sous-classe de la classe *Etudiant***. Cela provoque donc une erreur à la compilation.

Un objet de type *EtudiantInfoGestion* peut être déclaré (cfr (8)) car la classe *EtudiantInfoGestion* a été déclarée avec la protection de type *package* et se trouve dans le même package que la classe *Principal*. Cet objet peut donc être créé en mémoire et initialisé en appelant le constructeur de la classe *EtudiantInfoGestion* (cfr (9)), car celui-ci est déclaré de type *package* et la classe *EtudiantInfoGestion* se trouve dans le même package que la classe *Principal*.

La méthode *toString()* peut être appelée sur l'objet *i* (de type *EtudiantInfoGestion*) car elle est déclarée *public* (cfr (10)).

Par contre, la méthode *age()* ne peut être appelée sur un objet de type *EtudiantInfoGestion* (cfr (11)), 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 *moyenneCours()* peut, elle, être appelée sur un objet de type *EtudiantInfoGestion* (cfr (12)), car elle est déclarée *protected* dans la classe *EtudiantInfoGestion*, et que **la classe *Principal* se trouve dans le même package que la classe *EtudiantInfoGestion***.

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()*....

3.7. Protection de type *private* et héritage

Une sous-classe ne peut pas accéder directement aux variables d'instance, méthodes et constructeurs déclarés *private* dans la super-classe.

Exemples :

```
public class Personne
{ private String nomPre;
  private char sexe;
  private GregorianCalendar dateNaiss;

  private Personne(String nP, char x, int an, int mois, int jour)
  { nomPre= nP;
    sexe = x;
    dateNaiss = new GregorianCalendar (an, mois, jour);}

  private int age() { ... }

  public String toString() // rappel, déclaration immuable, donc public obligatoirement
  {return "La personne " + nomPre;}
}
```

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

  public Etudiant (String nP, char x, int a, int m, int j, String s, int aE)
  { super(nP, x, a, m, j);
    section = s;
    annee = aE;}
                                     (1)

  public String loginName() { ...}

  public String toString()
  {return nomPre +
    " (" + this.age() +"ans) inscrit en " + section +
    "\na reçu le login name: " + this.loginName( );
    }
                                     (2)
                                     (3)
}
```

Le constructeur de la classe *Etudiant* ne peut pas faire appel au constructeur de la classe *Personne* (via *super(...)*), car le constructeur de la classe *Personne* est déclaré *private*. L'instruction (1) provoque une erreur à la compilation.

De même, la méthode *toString()* ne peut faire appel à la variable d'instance *nomPre* héritée de la classe *Personne*, car cette variable *nomPre* est déclarée *private* (cfr (2)).

La méthode *toString()* ne peut pas plus faire appel à la méthode *age()* héritée de la classe *Personne*, car cette méthode est déclarée **private** (cf(3)).

Ces deux dernières erreurs (cfr (2) et (3)) sont détectées à la compilation.

Pour accéder aux variables d'instance privées héritées de la classe *Personne*, il faut prévoir des getters déclarés public dans la classe *Personne* :

```
public class Personne
{ private String nomPre;
  private char sexe;
  private GregorianCalendar dateNaiss;

  public String getNomPre() {return nomPre ;}
  public char getSexe() {return sexe ;}
  public GregorianCalendar getDateNaiss() {return dateNaiss ;}
  ...
}
```

Au sein de la classe *Etudiant*, il faut faire appel le cas échéant aux getters de la classe *Personne*.

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

  public String toString()
  {return getNomPre() +
    " (" + this.getAge() +"ans) inscrit en " + section +
    "\nreçu le login name: " + this.loginName( );
  }
}
```

En conclusion :

- ① Une variable d'instance, constructeur ou méthode déclaré **private** dans une super-classe n'est pas accessible au sein de la sous-classe.
- ② Pour accéder en lecture à une variable d'instance héritée, il faut utiliser le getter public de la super-classe.

Chapitre 4 : *static* et *final*

Variables et méthodes de classe (*static*)

4.1. Variables 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*

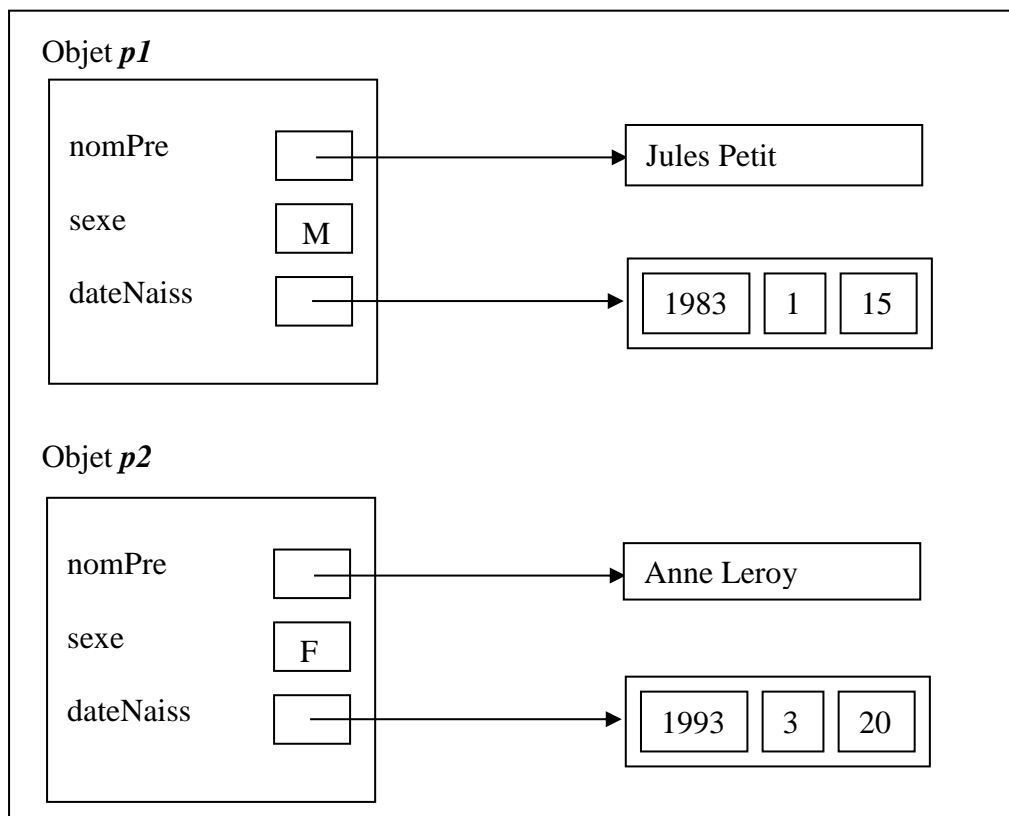
Personne
nomPre sexe dateNaiss
age() toString()

Soient les instructions de création des objets *p1* et *p2* de type *Personne* :

`Personne p1 = new Personne ("Jules Petit", 'M', 1983, 1, 15) ;`

`Personne p2 = new Personne ("Anne Leroy", 'F', 1993, 3, 20) ;`

En mémoire, les objets *p1* et *p2* ont la structure suivante :



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 (càd le nombre d'objets de type *Personne* qui sont de sexe féminin)

ou encore,

- *la moyenne d'âge* des personnes (càd 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 de sexe féminin créés par le programme principal

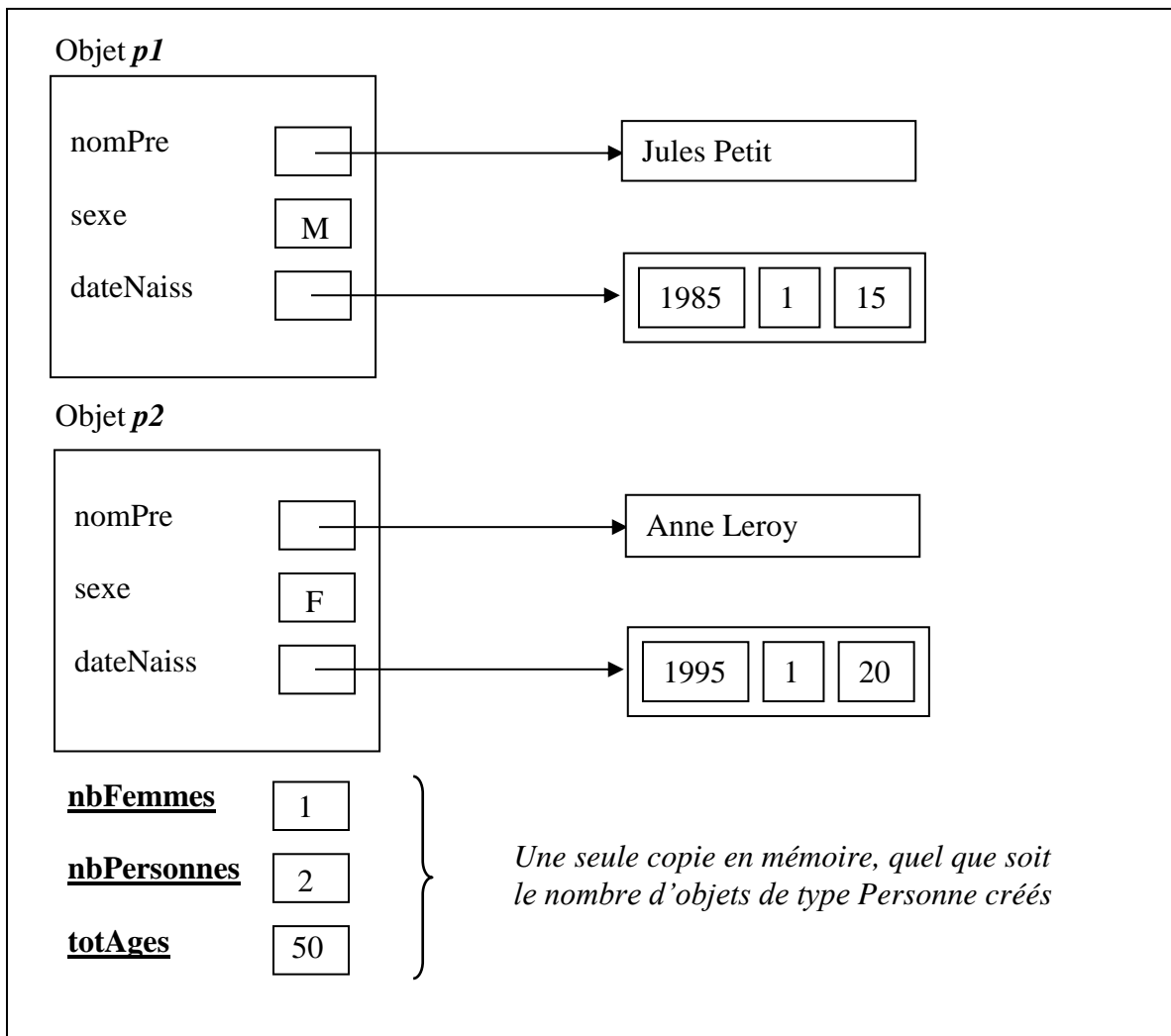
- les variables ***nbPersonnes*** et ***totAges*** pour comptabiliser le nombre de personnes créées et totaliser les âges de ces personnes, en vue de calculer la moyenne d'âge:

(moyenne = totAges / nbPersonnes).

Personne
nomPre sexe dateNaiss <i>nbFemmes</i> <i>nbPersonnes</i> <i>totAges</i>
age() toString()

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

```
class Personne
{ String nomPre;
  char sexe;
  GregorianCalendar dateNaiss;
  static int nbFemmes, = 0 ;
  static int nbPersonnes = 0 ;
  static float totAges = 0;
  ...
}
```

En mémoire :Où mettre à jour les variables de classe ?

Càd, 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 de rajouter dans tout constructeur, les instructions de mise à jour des variables de classe.

class **Personne**

```
{ String nomPre;
  char sexe;
  GregorianCalendar dateNaiss;
  static int nbFemmes = 0, nbPersonnes = 0;
  static float totAges = 0;
```

```
Personne(String nP, char x, int an, int mois, int jour)
{
    nomPre= nP;
    sexe = x;
    dateNaiss = new GregorianCalendar(an, mois, jour);
    if (sexe == 'F' || sexe == 'f') nbFemmes ++;
    nbPersonnes ++;
    totAges += age( );
}

int age( ) { ... }

...
}
```

Comment accéder aux variables de classe ?

Les variables de classe étant des caractéristiques de la classe, elles existent indépendamment des objets de cette classe. Elles sont accessibles qu'il y ait ou non des objets de cette classe créés.

On accède aux variables de classe via le nom de la classe :

... NomClasse.variableDeClasse ... ;

Exemple :

```
public class Principal

{
    public static void main(String[ ] args)

    {
        Personne p1 = new Personne ("Jules Petit",'M',1985,1,15) ;
        Personne p2 = new Personne ("Anne Leroy",'F',1995,1,20) ;

        System.out.println("nombre de femmes: " + Personne.nbFemmes);
        ⇒ affiche à l'écran:
           nombre de femmes: 1

        System.out.println("nombre de personnes: " + Personne.nbPersonnes);
        ⇒ affiche à l'écran:
           nombre de personnes: 2

        System.out.println("total des âges: " + Personne.totAges);
        ⇒ affiche à l'écran:
           total des âges: 50

        float ageMoyen = Personne.totAges / Personne.nbPersonnes;
        System.out.println("âge moyen: " + ageMoyen);
        ⇒ affiche à l'écran:
           âge moyen: 25
    }
}
```


4.2. Méthodes de classe

Les caractéristiques de classe peuvent aussi bien être des variables (**variables de classe**) que des méthodes (**méthodes dites de classe**). Le raisonnement tenu pour les variables de classe est valable pour les méthodes de classe. On déclare ces méthodes *static* et on fait appel à ces méthodes via le nom de la classe.

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 *totAges* tels que définis dans le point 4.1.

class **Personne**

```
{ String nomPre;
  char sexe;
  GregorianCalendar dateNaiss;
  static int nbFemmes = 0, nbPersonnes = 0;
  static float totAges = 0;

  Personne(String nP, char x, int an, int mois, int jour) { ... }

  int age() { ... }

  static float moyenneAge( )
    { return totAges / nbPersonnes; }
    ...
}
```

Comment appeler une méthode de classe ?

On appelle une méthode de classe via le nom de la classe :

... NomClasse.*methodeDeClasse*(...) ... ;

public class **Principal**

```
{ public static void main(String[ ] args)
{
    ...

    System.out.println("âge moyen: " + Personne.moyenneAge( ));
    ⇒ affiche à l'écran:
        âge moyen: 25
}
}
```

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 :

```
Personne p1 = new personne( ... );  
System.out.println("nombre de personnes: " + p1.nbPersonnes);  
System.out.println("âge moyen: " + p1.moyenneAge( ));
```

4.3. Variables de classe et le principe de l'Information Hiding

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 ...

Par conséquent, si le concepteur de la classe désire permettre l'accès en lecture ou en écriture aux variables de classe en dehors du package, 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* et sont 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 nomPre;  
    private char sexe;  
    private GregorianCalendar dateNaiss;  
    private static int nbPersonnes = 0, nbFemmes = 0;  
    private static float totAges = 0;  
  
    public Personne(String nP, char x, int an, int mois, int jour) { ... }  
  
    public int age( ) { ... }  
  
    public static float moyenneAges( )  
        { return totAges/nbPersonnes;}  
  
    public static int getNbFemmes( )  
        {return nbFemmes;}  
  
    public static int getNbPersonnes( )  
        {return nbPersonnes;}  
  
    public static float getTotAges( )  
        {return totAges;}  
}
```

```

public class Principal

{ public static void main(String[ ] args)

{ Personne p1 = new Personne ("Jules Petit",'M',1985,1,15) ;
  Personne p2 = new Personne ("Anne Leroy",'F',1995,1,20) ;

  System.out.println("nombre de femmes: " + Personne.getNbFemmes( ));
    ⇒ affiche à l'écran:
        nombre de femmes: 1

  System.out.println("nombre de personnes: " + Personne.getNbPersonnes( ));
    ⇒ affiche à l'écran:
        nombre de personnes: 2

  System.out.println("total des âges: " + Personne.getTotAges( ));
    ⇒ affiche à l'écran:
        total des âges: 50
    }
}

```

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**.

Le mot réservé *final*

4.4. 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 (ex : `objet1.montant = 10`) ;
- 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 (ex : `objet1.setMontant(10)` ;).

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**.

Exemple :

```
public class ClassX
{ public final int VALMIN = 10 ;
  ...
}
```

On accède à une constante comme on accède à une variable d'instance :

... **nomObjet.CONSTANTE** ... ;

Exemple :

```
public class Principal
{ public static void main(String[ ] args)
  { ClassX objetC = new ClassX(...);
    ... objetC.VALMIN ... ;
  }
}
```

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

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 ***final static***.

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

... **NomClasse.CONSTANTEdeCLASSE** ... ;

Illustration de l'emploi des constantes de classe :
la gestion des dates via la classe *GregorianCalendar*

Pour rappel, pour créer un objet de type date, on fait appel à la classe *GregorianCalendar*.

Exemples :

```
GregorianCalendar now = new GregorianCalendar( );  
GregorianCalendar date1 = new GregorianCalendar(2000,0,10);
```

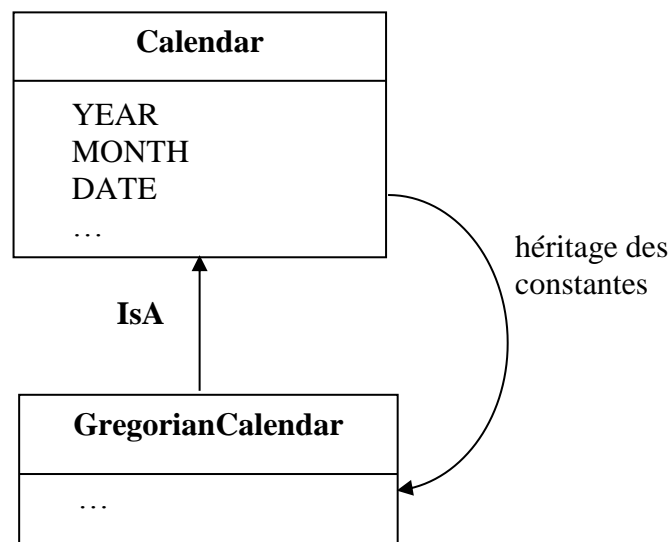
La date référencée par l'objet *now* correspond à la **date système** (appel au constructeur sans argument).

La date référencée par l'objet *date1* correspond au 10 janvier 2000. En effet, le second argument correspondant au mois peut prendre une valeur entre **0** (janvier) et **11** (décembre).

La classe *GregorianCalendar* est une **sous-classe de la classe *Calendar***.

La classe *Calendar* contient de nombreuses **constantes de classe** telles que par exemple :

```
public final static int YEAR = 1;  
public final static int MONTH = 2;  
public final static int DATE = 5;  
...
```



La classe *GregorianCalendar* étant une sous-classe de la classe *Calendar*, elle hérite de ses constantes de classe. Pour accéder à la constante *YEAR*, on peut donc écrire :

Calendar.YEAR
ou encore : **GregorianCalendar**.YEAR

Où utiliser de telles constantes ?

Ces constantes sont utilisées lorsque l'on veut récupérer l'année, le mois ou le jour d'un objet date de type *GregorianCalendar*.

La méthode disponible pour accéder à une des zones (année, mois, jour, ...) d'un objet de type *GregorianCalendar* est la méthode **get(...)** qui prend **un argument**, à savoir la **constante** qui détermine à quelle zone de la date on veut accéder.

Exemples :

```
GregorianCalendar date1 = new GregorianCalendar(2000,0,10);
```

```
System.out.println("l'année de date1 = " + date1.get(Calendar.YEAR));
```

⇒ affiche à l'écran:

l'année de date1 : 2000

```
System.out.println("le mois de date1 = " + date1.get(Calendar.MONTH));
```

⇒ affiche à l'écran:

le mois de date1 : 0

```
System.out.println("le jour de date1 = " + date1.get(Calendar.DATE));
```

⇒ affiche à l'écran:

le jour de date1 : 10

Notons que l'on aurait obtenu les mêmes résultats avec les instructions suivantes :

```
System.out.println("l'année de date1 = " + date1.get(GregorianCalendar.YEAR));
```

```
System.out.println("le mois de date1 = " + date1.get(GregorianCalendar.MONTH));
```

```
System.out.println("le jour de date1 = " + date1.get(GregorianCalendar.DATE));
```

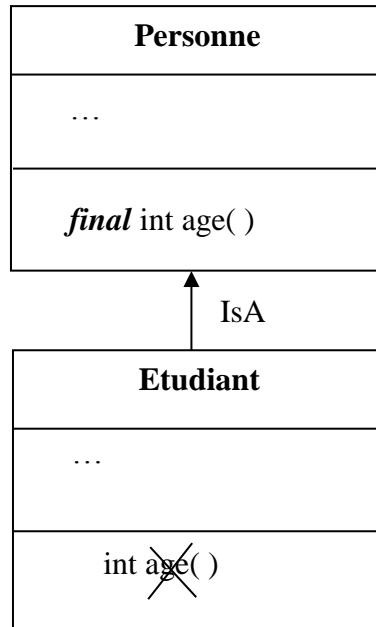
D'autres constantes de classe existent dans la classe *Calendar* permettant de récupérer à partir d'un objet de type date, par exemple :

- l'**ère** à laquelle appartient la date,
- le **numéro du jour dans l'année**,
- l'**heure** ou les **minutes** correspondant à la date, ...

4.5. Méthode déclarée *final*

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

Exemple :

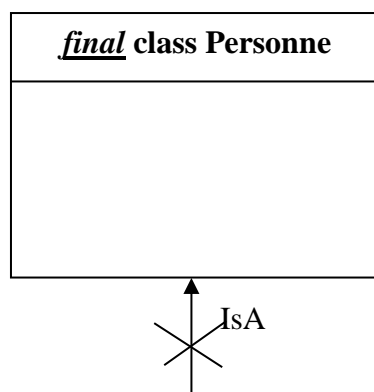


Dans l'exemple proposé ci-dessus, la méthode *age()* est déclarée ***final*** dans la classe *Personne*. Elle ne peut donc *plus être redéfinie* dans *aucune sous-classe* de la classe *Personne*, et donc en aucun cas dans la classe *Etudiant*.

4.6. 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, càd 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 de **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 être redéfinie* dans une sous-classe.

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

Chapitre 5 : Classes abstraites et interfaces

5.1. 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'occurrences 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.

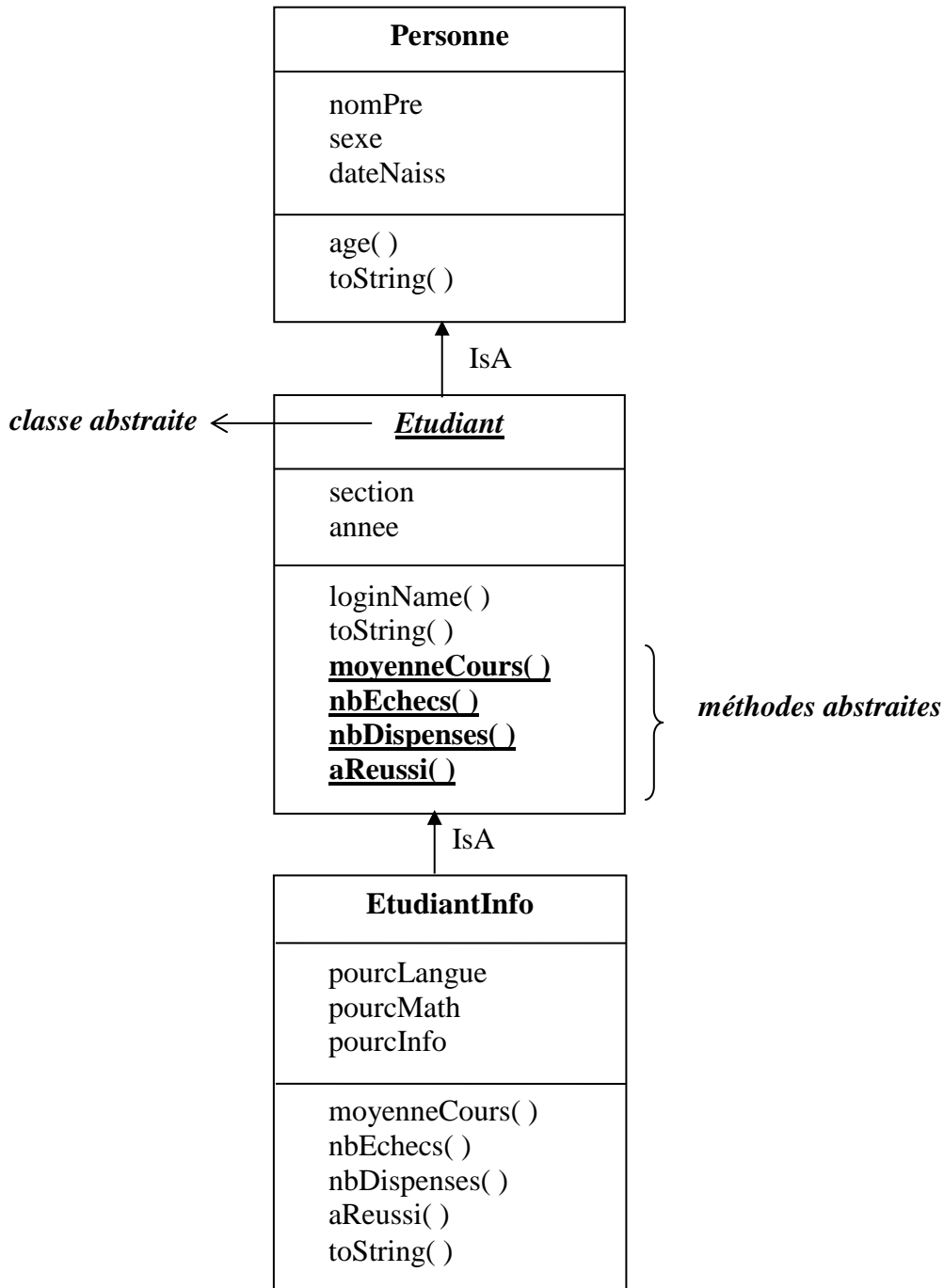
Déclaration correcte d'une méthode abstraite : **abstract** *typeRetour methodeX(...) ;*

Déclaration incorrecte d'une méthode abstraite : *typeRetour methodeX(...) { }*

Une *classe abstraite* doit contenir le mot réservé **abstract** dans sa déclaration.

Dans l'exemple 1 ci-dessous, la classe *Etudiant* est abstraite car elle contient 4 méthodes qui sont abstraites, à savoir, les méthodes *moyenneCours()*, *nbEchecs()*, *nbDispenses()* et *aReussi()*. En effet, il est impossible de leur donner une implémentation. Il faudrait, pour ce faire, connaître les résultats obtenus par l'étudiant pour calculer sa réussite. Or, il n'y a aucune variable d'instance dans la classe *Etudiant* reprenant les résultats de l'étudiant. Par contre, la classe *EtudiantInfo*, qui est une sous-classe de la classe *Etudiant*, contient les variables d'instance *pourcLangue*, *pourcMath* et *pourcInfo*. Ce qui permet de donner une implémentation aux méthodes *moyenneCours()*, *nbEchecs()*, *nbDispenses()* et *aReussi()*. La classe *EtudiantInfo* étant une sous-classe d'*Etudiant*, elle hérite de ces méthodes. Elle peut donc les **implémenter**. Puisque ces 4 méthodes abstraites héritées de la classe *Etudiant* contiennent une implémentation, la classe *EtudiantInfo* n'est pas une classe abstraite. **On peut donc créer des occurrences de type *EtudiantInfo*, 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 *EtudiantInfo* qui appelle le constructeur de la classe *Etudiant* via l'instruction *super(...)*.

Exemple 1:

```

public class Personne
{ private String nomPre;
  private char sexe;
  private GregorianCalendar dateNaiss;

  public Personne(String nP, char x, int an, int mois, int jour)
  { nomPre= nP;
    sexe = x;
    dateNaiss = new GregorianCalendar (an, mois, jour);}
  }

```

```

public int age() { ... }

public String toString() { ... }
}

```

```

public abstract class Etudiant extends Personne
{ private String section;
  private int annee;

```

constructeur prévu même si on ne peut pas créer des occurrences de la classe Etudiant

```

  public Etudiant (String nP, char x, int a, int m, int j, String s, int aE)
  { super(nP, x, a, m, j);
    section = s;
    annee = aE; }

```

```

public String loginName() { ... }

```

```

public abstract float moyenneCours() ;
public abstract int nbEchecs() ;
public abstract int nbDispenses() ;
public abstract boolean aReussi() ;
}

```

*méthodes **abstraites** : pas d'implémentation :
déclaration de méthode abstraite terminée par ;*

```

public class EtudiantInfo extends Etudiant

```

```

{ private float pourcLangue, pourcMath, pourcInfo;

```

```

  public EtudiantInfo(String nP, char x, int a, int m, int j, String s, int aE, int pL, int pM, int pI)
  { super(nP, x, a, m, j, s, aE);  $\longrightarrow$  appel au constructeur de la super-classe abstraite
    pourcLangue = pL;
    pourcMath = pM;
    pourcInfo = pI ; }

```

```

  public float moyenneCours()
  { return (pourcLangue + pourcMath + pourcInfo)/3; }

```

```

  public int nbEchecs()
  { int n = 0;
    if (pourcLangue <=10) n ++;
    if (pourcMath <=10) n ++;
    if (pourcInfo <=10) n ++;
    return n; }

```

```

  public int nbDispenses()
  { int n = 0;

```

```
if (pourcLangue >=12) n ++;  
if (pourcMath >=12) n ++;  
if (pourcInfo >=12) n ++;  
return n; }
```

```
public boolean aReussi( )  
{ if (this.moyenneCours( ) >= 60 && this.nbEchecs( ) == 0) return true;  
  else return false;}
```

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

```
public class Principal
```

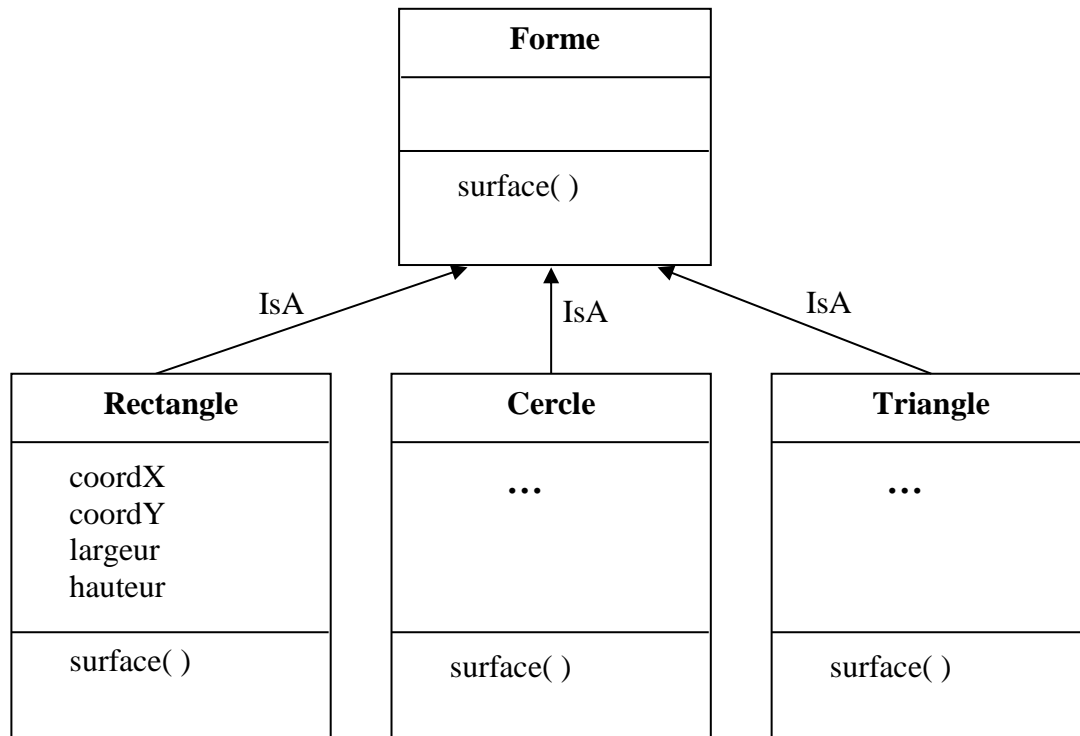
```
{ public static void main(String[ ] args)
```

```
{ Etudiant e1 = new Etudiant ("Jules Petit",'M',1995,1,15, "droit",2) ;  
  ↪ cette instruction provoque une erreur à la compilation,  
    car la classe Etudiant est abstraite,  
    et ce, même si le constructeur existe
```

```
EtudiantInfo e2 = new EtudiantInfo ("Jules Petit",'M',1995,1,15, "droit",2, 50, 60, 70) ;  
  ↪ 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 met 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 coordX, coordY, largeur, hauteur ;

    public Rectangle (int x, int y, int larg, int haut)
        { ... }
    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'occurrences 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.

5.2. 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* :

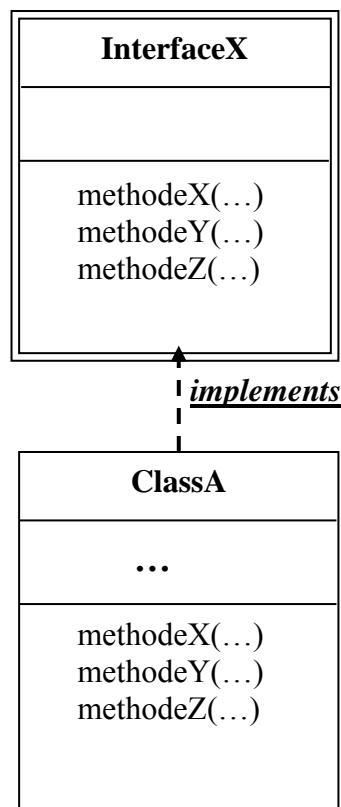
```
interface NomInterface
{ ...
}
```

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 une utilité que si des **classes s'engagent à redéfinir les méthodes de l'interface** et donc à leur donner une *implémentation*. Une classe qui s'engage à implémenter les méthodes dont les déclarations sont reprises dans une interface contient la clause *implements* dans sa déclaration suivi du nom de l'interface :

```
class ClassX 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) ;
}
```

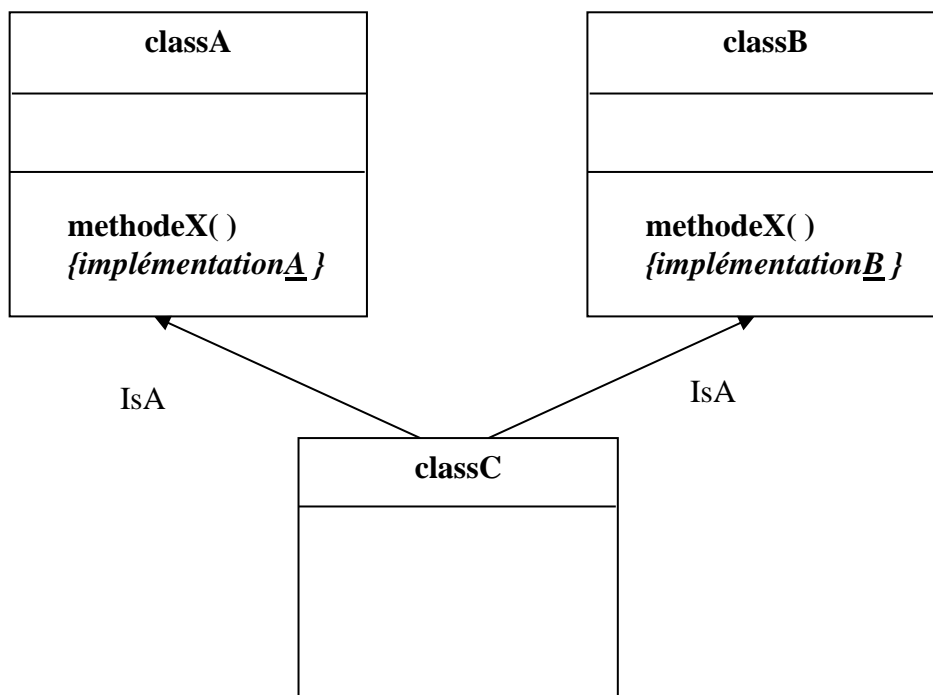
toutes les méthodes sont implicitement **public** et **abstract**

```
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 (cfr chapitre 3).

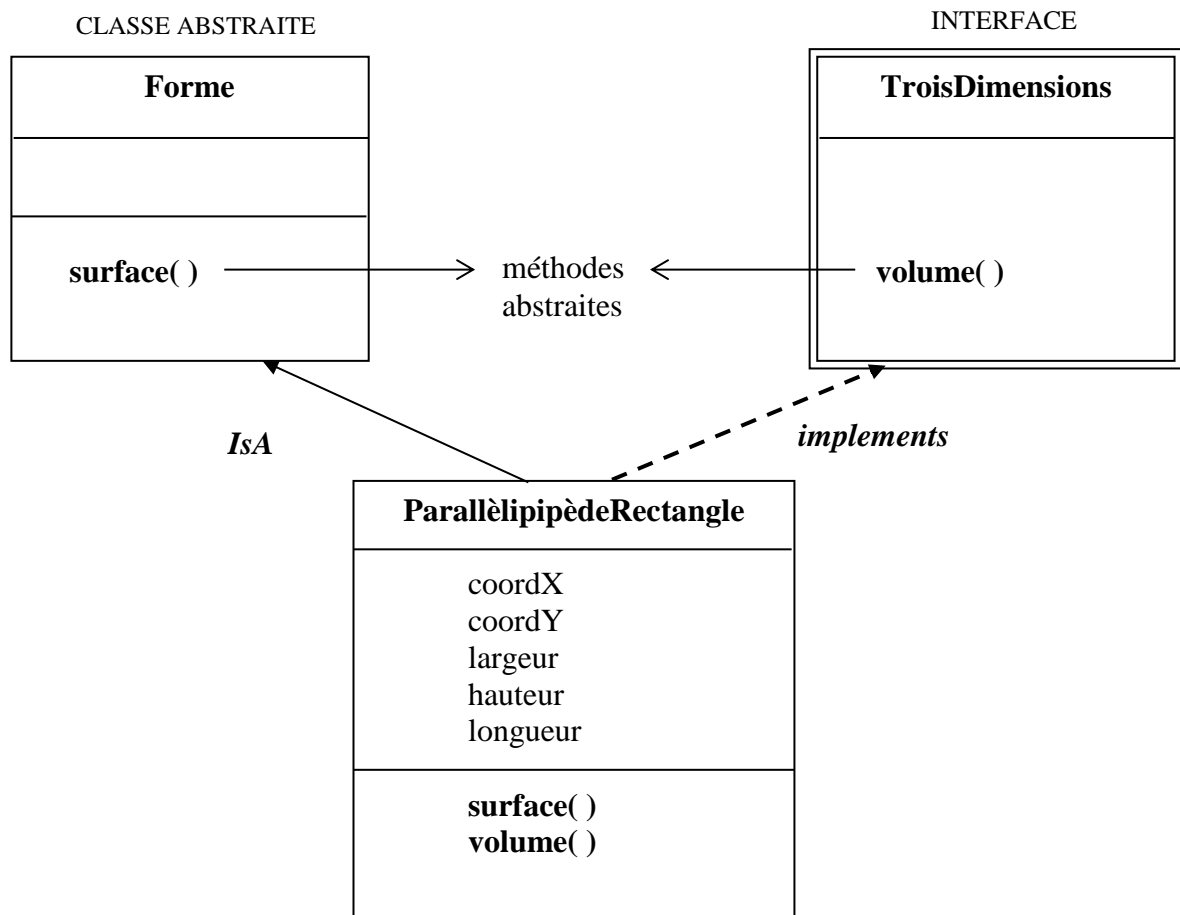
Il n'y a **pas d'héritage multiple permis en Java**. Une classe ne peut hériter de plusieurs super-classes à la fois. Cette restriction se justifie par le ***risque de conflit*** qu'un héritage multiple pourrait engendrer. En effet, que faire si deux méthodes avec la même signature se trouvent dans deux des super-classes ? Laquelle des deux implémentations exécuter lorsque cette méthode est appelée sur une occurrence de la sous-classe ?



Dans la hiérarchie ci-dessus, 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
{ public int volume() ;
}
```

```
public class ParallèlipèdeRectangle extends Forme implements TroisDimensions
{ private int coordX, coordY, largeur, hauteur, longueur ;
  ...
  public int surface()
  { ... }
  public int volume()
  { ... }
}
```

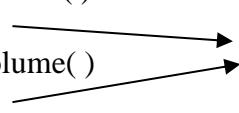
Arrows from the `surface()` and `volume()` methods in the **ParallèlipèdeRectangle** class point to the text *implémentation*.

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
{ public int surface( ) ;
}
```

```
public interface TroisDimensions
{ public int volume( ) ;
}
```

```
public class ParallelipedeRectangle implements Forme, TroisDimensions
{ private int coordX, coordY, largeur, hauteur, 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 ClassX 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).

Chapitre 6 : 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.

Enfin, nous abandonnerons les tableaux pour utiliser la notion de liste.

6.1. 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 (càd le nombre maximum d'éléments que pourra contenir ce tableau)
- prévoir une **variable** de type entier qui mémorisera à chaque instant le **nombre réel d'éléments** que contient le tableau
- prévoir les **méthodes d'accès** nécessaires :
 - o **Getters** (ex : pour lire le $i^{\text{ème}}$ élément, pour connaître le nombre réel d'éléments dans le tableau, ...)
 - o **Setters** (ex : 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**.

Ex : `int [] tabValeur ;`

⇒ déclare un tableau appelé tabValeur 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]**.

Ex : `tabValeur = new int [10] ;`

⇒ réserve la place en mémoire pour 10 éléments de type entier.

La déclaration et la réservation de place en mémoire peuvent se faire en une seule instruction.

Ex : `int [] tabValeur = new int [10] ;`

Accès aux éléments du tableau

Chaque élément du tableau a un indice. Le premier élément a l'indice 0.

Pour accéder à un élément du tableau, on utilise la notation **nomTableau[indice]**.

Ex : `tabValeur[0]` ⇒ désigne le premier élément du tableau.

`tabValeur[9]` ⇒ désigne le 10^{ème} élément du tableau


Exemple :

Supposons que les résultats obtenus aux interrogations doivent être mémorisés pour chaque objet de la classe *Etudiant*. Les interrogations sont cotées sur 20, avec un maximum de deux décimales. Partons du principe que l'on n'enregistrera pas plus de 10 interrogations par étudiant. Un tableau de type **float** et de longueur **10** est donc prévu comme variable d'instance dans la classe *Etudiant*. Ce tableau est appelé *interro*. La déclaration et l'initialisation de ce tableau est donc : **float[] interro = new float[10];**
Une variable appelée *nbInterros* est utilisée pour mémoriser à chaque instant le nombre réel d'interrogations. Cette variable est initialisée à zéro lors de la création de tout nouvel étudiant.

Soient, par exemple, trois cotes d'interrogation stockées dans ce tableau :

tableau *interro*

18.3	15.5	9.5	0	0	0	0	0	0	0
------	------	-----	---	---	---	---	---	---	---


trois cotes dans le tableau
⇒ **nbInterros** vaut 3

Rappel : Toute variable d'instance doit être déclarée private. Le tableau et la variable nbInterros sont donc déclarés tous deux private. Cela nécessite la création de getters et setters.

```
public class Etudiant

{ ...
  private float[ ] interro = new float[10];
  private int nbInterros ;

  // constructeur
  public Etudiant(...)
  { ...
    nbInterros = 0; }

  //getters
  public float getInterro (int position)
  { position --;
    return interro[position]; }

  public int getNbInterros( )
  { return nbInterros; }

  //setters
  public void setInterro (float cote, int position)
  { position - -;
    interro[position] = cote; }

  public void ajoutInterro (float cote)
  { interro[nbInterros] = cote;
    nbInterros ++;      } }
```

La méthode *getInterro* (*int position*) retourne la cote de l'interrogation enregistrée à une position donnée dans le tableau (cfr 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. L'instruction ***position* - -** doit donc être exécutée avant d'accéder au tableau en utilisant la variable *position* comme indice.

La méthode *getNbInterros*() retourne le nombre réel d'interrogations dans le tableau. Cette information est stockée dans la variable *nbInterros*.

La méthode *setInterro* (*float cote*, *int position*) modifie la cote de l'interrogation située à une position donnée dans le tableau (cfr argument *position*). Rappelons que l'argument *position* indique la position de l'interrogation dans le tableau **en partant de 1**. L'instruction ***position* - -** doit donc aussi être exécutée avant d'accéder au tableau en utilisant la variable *position* comme indice.

La méthode *ajoutInterro* (*float cote*) ajoute une nouvelle interrogation à la première position libre dans le tableau. La première position libre dans le tableau est indiquée par la variable *nbInterros*. En effet, s'il y a par exemple 3 interrogations dans le tableau, *nbInterros* vaut 3. La première cellule libre dans le tableau se trouve à la 4^{ème} position, autrement dit à l'indice 3. Après avoir utilisé *nbInterros* comme indice dans l'instruction ajoutant la nouvelle interrogation, il ne faut pas oublier **d'incrémenter de 1 la variable *nbInterros*** pour actualiser le nombre réel d'interrogations dans le tableau (*nbInterros* ++).

N.B. Il serait plus correct de vérifier en plus le non-débordement du tableau : càd ne pas ajouter un nouvel élément dans le tableau s'il est déjà plein. Ceci implique une gestion des **cas d'erreurs**. Un mécanisme puissant existe en java pour gérer les cas d'erreur ; il s'agit du mécanisme des exceptions. Ceci fait l'objet du chapitre suivant. Nous ne testerons donc pas de cas d'erreurs ici puisque cela fera l'objet d'une étude ultérieure.

```
public class Principal
{
    public static void main(String[] args)

        {Etudiant e = new Etudiant(...);

        // ajout de trois cotes d'interrogation dans le tableau interro
        e.ajoutInterro(10.5f);
        e.ajoutInterro(12);
        e.ajoutInterro(9.8f);
        (1)

        // modification du résultat de la deuxième interro
        e.setInterro(16.3f , 2);

        // boucle d'affichage des (résultats des) interros
        for (int i = 1; i <= e.getNbInterros( ); i++)
            System.out.println(e.getInterro(i) );
        }
}
```

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 `e.interro[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 :

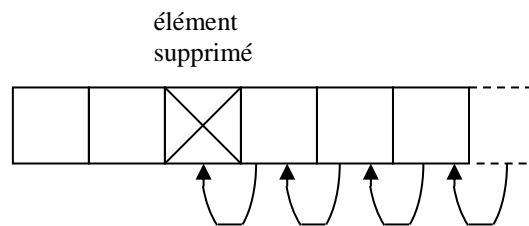
<code>e.ajoutInterro(10.5f);</code>	correct
<code>e.ajoutInterro[10.5f];</code>	incorrect

De même dans l’instruction (2) :

<code>e.getInterro(i)</code>	correct
<code>e.getInterro[i]</code>	incorrect

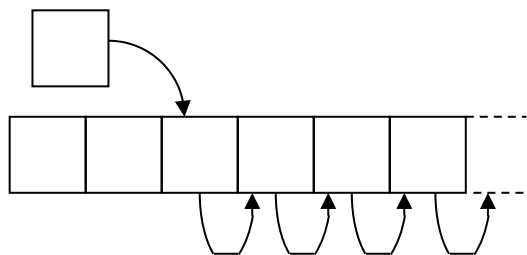
N.B. On pourrait prévoir encore d’autres méthodes d’accès au tableau, comme par exemple :

- une méthode permettant de *supprimer une interrogation qui se trouve à une position donnée*. Dans ce cas, il faudrait déplacer vers la gauche dans le tableau les éléments qui se trouvent à droite de l’élément supprimé, afin d’éviter les cellules vides dans le tableau.



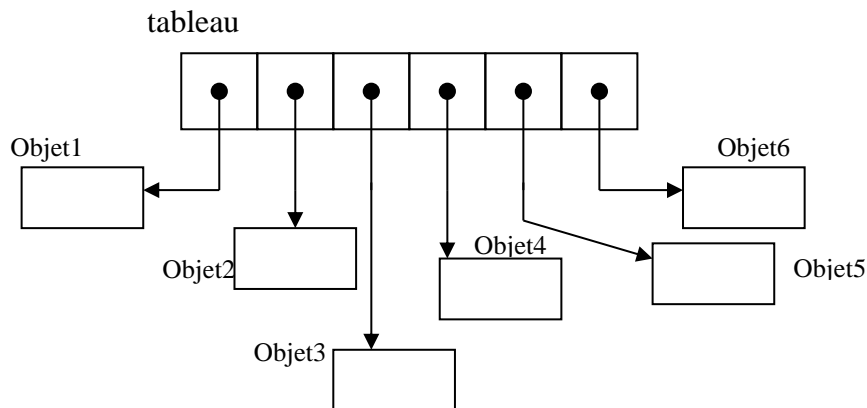
- une méthode permettant d’*insérer une nouvelle interrogation à une position donnée*. Dans ce cas, il faudrait d’abord déplacer vers la droite certains éléments du tableau avant d’y insérer le nouvel élément, afin de n’écraser aucun élément déjà présent dans le tableau.

nouvel élément à insérer



6.2. 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 une **variable** de type entier qui mémorisera à chaque instant le **nombre réel d'éléments** que contient le tableau
- prévoir les **méthodes d'accès** nécessaires :
 - o **Getters** (ex : pour lire le $i^{\text{ème}}$ élément, pour connaître le nombre réel d'éléments dans le tableau, ...)
 - o **Setters** (ex : pour modifier le $i^{\text{ème}}$ élément, pour ajouter un nouvel élément au 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 *stagiaire*. L'instruction `Etudiant [] stagiaire = new Etudiant [5]` a pour effet de créer en mémoire un tableau de 5 éléments, chacun 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*.

Une variable appelée *nbStagiaires* est utilisée pour mémoriser à chaque instant le nombre réel de stagiaires stockés dans ce tableau. Cette variable est initialisée à zéro lors de la création de tout nouveau professeur.

```
public class Professeur

{ ...
  private Etudiant [ ] stagiaire = new Etudiant [5];
  private int nbStagiaires;

  // constructeur
  public Professeur(... )
  { ...
    nbStagiaires = 0; }

  //gettors
  public int getNbStagiaires( )
  { return nbStagiaires;}

  public Etudiant getStagiaire(int position) (1)
  { position --;
    return stagiaire[position];}

  //settors
  public void setStagiaire (Etudiant e, int position) (2)
  { position - -;
    stagiaire[position] = e;}

  public void ajoutStagiaire (Etudiant e) (3)
  { stagiaire[nbStagiaires] = e;
    nbStagiaires ++; }
}
```

La méthode `getNbStagiaires()` 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 (cfr argument *position*).

La méthode `setStagiaire (Etudiant e, int position)` remplace la référence de l'étudiant stockée à une position donnée par la référence de l'étudiant donné. *On peut discuter de l'intérêt d'une telle méthode si ce n'est dans la réorganisation du tableau.*

La méthode `ajoutStagiaire (Etudiant e)` ajoute la référence de l'étudiant donné, et ce, à la première position libre 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` (cfr (2) et (3)).

La méthode `getStagiaire`, quant à elle, retourne une référence vers un objet de type `Etudiant` (cfr (1)) .

```
public class Principal
{
    public static void main(String[] args)

    { // création de trois étudiants ainsi que de leurs interrogations
        Etudiant e1 = new Etudiant(...);
        e1.ajoutInterro(10.5f);
        e1.ajoutInterro(12);
        e1.ajoutInterro(9.8f);
        Etudiant e2 = new Etudiant(...);
        e2.ajoutInterro(6.5f);
        e2.ajoutInterro(18);
        Etudiant e3 = new Etudiant(...);
        e3.ajoutInterro(10f);
        e3.ajoutInterro(4);
        e3.ajoutInterro(19.8f);

        // création d'un professeur
        Professeur prof = new Professeur(...);

        // attribution de trois stagiaires : les étudiants e1, e2 et e3
        prof.ajoutStagiaire(e1);
        prof.ajoutStagiaire(e2);
        prof.ajoutStagiaire(e3);

        // boucle sur tous les stagiaires de prof :
        for (int i = 1; i <= prof.getNbStagiaires( ); i++)
        { System.out.println(prof.getStagiaire(i));           // appel à toString( ) de Etudiant
          // boucle d'affichage des interrogations du stagiaire i :
          for (int j = 1; j <= prof.getStagiaire(i).getNbInterros( ); j++)           (4)
              System.out.println(prof.getStagiaire(i).getInterro(j));           (5)
          }
        }
    }
}
```

Notons que *prof.getStagiaire(**i**).getNbInterros()* retourne le nombre d'interrogations du stagiaire **i** (cfr (4)) et que *prof.getStagiaire(**i**).getInterro(**j**)* retourne la cote obtenue par le stagiaire **i** à l'interrogation **j** (cfr (5)).

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.

Table des matières

Introduction :**La programmation orientée objet comparée à la programmation classique**

1. Objectifs du cours	2
2. Historique	2
3. Caractéristiques de Java	2
4. Programmation classique versus programmation O.O.	3

Chapitre 1 : Objets et classes

1.1. Objet	5
1.2. Classe	6
1.3. Variable d'instance et méthode	7
1.4. Constructeur	9
1.5. Accès aux variables d'instance	12
1.5.1. Accès en lecture	12
1.5.2. Accès en écriture	12
1.6. Appel de méthodes sur des objets	13
1.6.1. Méthode qui retourne une valeur	13
1.6.2. Méthode qui ne retourne pas de valeur	14
1.7. Difficultés de choisir entre variables d'instance et méthodes	14
1.8. Une méthode peut appeler une autre méthode de la même classe	18
1.9. Portée des variables	20
1.10. Surcharge (<i>overloading</i>) des constructeurs et méthodes	21
1.10.1. Surcharge des constructeurs	21
1.10.2. Surcharge des méthodes	23
1.11. La méthode <i>toString()</i>	26
1.12. Liens entre objets	28
1.12.1. Liens entre deux objets (entre deux classes)	28
1.12.2. Appel implicite à la méthode <i>toString()</i> d'un objet référencé par une variable d'instance	31
1.12.3. Plus de deux classes reliées	32
1.13. Méthode retournant un objet	36

Chapitre 2 : Protections (*information hiding*)

2.1. Information hiding : protections <i>private</i> et <i>public</i>	40
2.2. Méthodes publiques d'accès aux variables d'instance privées : <i>getter</i> et <i>setters</i>	41
2.3. Notion de package (+ <i>import</i>)	45
2.4. Protection par défaut : protection de type <i>package</i>	47

Chapitre 3 : Héritage

3.1. Sous-classes (déclaration et constructeur)	49
3.2. Héritage des variables d'instance et des méthodes	51
3.3. Redéfinition de méthodes	52
3.4. Hiérarchie d'héritage	54
3.5. Polymorphisme	59
3.6. Protection de type <i>protected</i>	63
3.7. Protection de type <i>private</i> et héritage	67

Chapitre 4 : static et final

Variables et méthodes de classe (static)

4.1. Variables de classe	69
4.2. Méthodes de classe	73
4.3. Variables de classe et le principe de l' <i>information hiding</i>	74

Le mot réservé final

4.4. Constante : variable déclarée <i>final</i>	76
<i>La gestion des dates via la classe GregorianCalendar</i>	77
4.5. Méthode déclarée <i>final</i>	79
4.6. Classe déclarée <i>final</i>	79

Chapitre 5 : Classes abstraites et interfaces

5.1. Classe abstraite (<i>abstract</i>)	81
5.2. Interface	86

Chapitre 6 : Les tableaux

6.1. Tableau d'éléments de type primitif	90
6.2. Tableau d'objets	94