Programmation Orientée Objet (POO)
1ère année SRI

M.C. Lagasquie

31 octobre 2020

Table des matières

Chapitre 1

Introduction

POO = programmation guidée par la définition d'objets et la manière dont ils interagissent.
Pourquoi des objets ? Parce que les objets sont partout !
Un exemple : la salle de cours. Il y a des tables, des chaises, un tableau, un écran, des gens (étudiants, prof). Si on veut "simuler" cet environnement, il faut être capable de "simuler" chacune des "entités" qui sont dans la pièce.

Aspect statique Chaque entité est caractérisée par des attributs : une `personne` est grande ou petite (attribut : `taille`) ; une `chaise` a 4 pieds, une `table` aussi alors qu'une `personne` n'en a que 2 (attribut : `nb_pieds`) ; une `personne` est vivante alors qu'une `table` est inanimée (attribut : `animée_ou_pas`). Un `écran` est blanc alors que le `tableau` est noir (attribut : `couleur`). On a donc pour chaque entité des triplets ⟨`entité, attribut, valeur`⟩. Certains attributs sont communs à plusieurs entités :

⟨`table, nb_pieds, 4`⟩
⟨`personne, nb_pieds, 2`⟩
⟨`ecran, couleur, blanc`⟩
⟨`tableau, couleur, noir`⟩.

D'autres attributs sont propres à certaines entités :

⟨`ecran, déployé_ou_non, déployé`⟩
⟨`néon, allumé_ou_pas, allumé`⟩.

On voit aussi que certains attributs sont liés au "type d'entité" (`table`, `personne`, `ecran`) et d'autres aux "individus" inclus dans ces types :

⟨`cette_table_ci, etat, mauvais`⟩
⟨`cette_table_là, etat, bon`⟩
⟨`Dupont, taille, petit`⟩
⟨`Durand, taille, grand`⟩
⟨`Lagasquie, taille, moyen`⟩
⟨`ce_néon_ci, allumé_ou_pas, allumé`⟩
⟨`ce_néon_là, allumé_ou_pas, éteint`⟩.

On doit donc faire un distinguo entre toutes ces choses-là.

Aspect dynamique Les entités interagissent entre elles. Par exemple, l'entité `personne` (`Lagasquie`) a "agi" sur l'entité `écran` pour l'enrouler. Cette action se fait par transmission d'un "message" de l'entité `personne Lagasquie` vers l'entite `écran cet_écran_là`. Cette transmission de message a provoqué l'évolution d'un attribut de l'écran d'une valeur vers une autre valeur.

envoi du message :

$$\texttt{Lagasquie} \xrightarrow{\text{rembobine-toi}} \texttt{cet\_écran\_là}$$

effet du message :

⟨`cet_écran_là, déployé_ou_non, déployé`⟩ devient ⟨`cet_écran_là, déployé_ou_non, rembobiné`⟩

Autres exemples de messages :

`personne` $\overset{\text{écoute-moi}}{\longrightarrow}$ `personne`

`personne` $\overset{\text{asseyez-vous}}{\longrightarrow}$ collection de `personne`

`Durand` $\overset{\text{il\_faut\_que\_je\_te\_raconte\_blablablabla}}{\longrightarrow}$ `Dupont`

Les objets sont partout. Ils peuvent être modélisés sous la forme d'un ensemble d'attributs, chaque attribut étant lié à un ensemble de valeurs possibles, et un ensemble de messages pour interagir avec les autres objets.

Remarque : toutes les notions vues dans ce cours vont être illustrées à l'aide d'un langage de programmation spécifique, le JAVA (excepté pour les notions de base qui seront aussi illustrées en C++, langage qui sera étudié l'année prochaine).
Attention : certains mécanismes de mise en œuvre particuliers à Java peuvent ne pas être implémentés de la même façon dans un autre langage (cela sera signalé au fur et à mesure du cours).
La documentation sur Java (en particulier toute la bibliothèque Java) est accessible depuis l'url :

`http://docs.oracle.com/javase/8/docs/api/`

Ce cours a été bâti en utilisant les ouvrages [Ber10, CD01, Bar09].

Chapitre 2

Définitions de base

Nous avons donc à définir *a minima* les notions suivantes :
- les attributs et leur valeur,
- les objets : leur "type" (appelé "classe" en programmation objet) et leurs "individus" (appelés "instances" en programmation objet),
- les messages entre objets (appelés "méthodes" en programmation objet).

Pour chaque notion, nous donnerons la manière dont on peut la définir en Java et en C++.

2.1 Les attributs

La notion d'attribut évoquée en introduction est aussi appelée "attribut" en programmation objet. Il s'agit de variables (au sens de tous les langages de programmation ; donc une zone de la mémoire accessible par une étiquette) rattachées à une entité. Donc qui n'existent que si l'entité existe.

Un exemple en java (idem en C++) : dans l'entité `personne`, on trouvera :

```
int age ;                                   /* attribut donnant l'age d'une personne */
```

Un attribut étant une variable, il a donc un "type". Cela peut être un type dit "primitif" (comme pour `age`) ; parmi les types primitifs, on trouve les entiers (`int`), les réels (`float`), les caractères (`char`), les booléens (`boolean` en Java et `bool` en C++) :

```
int age ;                                   /* attribut donnant l'age d'une personne */
float taille ;                            /* attribut donnant la taille d'une personne */
char sexe ;                         /* attribut donnant le sexe d'une personne (M ou F) */
boolean estVivant ;          /* attribut indiquant qu'une personne est vivante ou pas */
```

ou en C++ :

```
bool estVivant ;             /* attribut indiquant qu'une personne est vivante ou pas */
```

Cela peut aussi être un autre objet ; par exemple, toujours dans l'entité `Personne`, on trouvera :

```
Personne père ;                           /* attribut donnant le père d'une personne */
Personne mère ;                           /* attribut donnant la mère d'une personne */
String nom ;                               /* attribut donnant le nom d'une personne */
```

ou en C++ :

```
string nom ;                               /* attribut donnant le nom d'une personne */
```

2.2 Les valeurs d'attribut

Les valeurs d'attribut sont fonction du type de cet attribut. On retrouve là les mêmes valeurs que celles utilisées dans la plupart des programmes. Pour les attributs de type primitif, on a :
- pour un attribut de type `int`, les valeurs sont tous les nombres entiers positifs ou négatifs limités par la taille maximale d'un entier sur la machine sur laquelle le programme tourne ;
- pour un attribut de type `float`, les valeurs sont tous les nombres réels positifs ou négatifs limités par la taille maximale d'un float sur la machine sur laquelle le programme tourne ;

— pour un attribut de type `char`, les valeurs sont tous les caractères représentables exprimés entre simple quote ('a', 'b', ..., 'A', ..., '0', '1', ...) :
— pour un attribut de type `boolean` (ou `bool` en C++), les seules valeurs possibles sont `true` et `false`.

Pour les attributs correspondants à des objets, la valeur par défaut en Java est `null`, ce qui signifie que la variable existe mais que l'objet correspondant n'existe pas. Puis dès que l'objet est créé, la valeur de la variable est alors l'adresse de l'objet créé. En ce qui concerne C++, la simple déclaration d'une variable objet peut suffire à créer l'objet (voir section 2.5).
A noter, qu'en Java, il existe une initialisation par défaut de tous les attributs (chose qui n'existe pas en C, ni en C++, par exemple) : tous les attributs numériques et caractères sont positionnés à 0, tous les attributs booléens sont positionnés à `false` et tous les attributs correspondant à des objets sont positionnés à `null`.
Attention : ces initialisations par défaut ne sont faites **que pour les attributs** et pas pour les variables locales !

2.3 Les méthodes

Rappelons qu'un objet peut agir sur les attributs d'un autre objet par l'envoi d'un message. En programmation objet, chaque message est appelé "méthode" et quand un objet o_2 veut agir sur un objet o_1, o_2 appelle une méthode de o_1.

Déclaration d'une méthode Les méthodes sont implémentées sous la forme de fonctions (au sens de tous les langages de programmation). Elles ont donc une liste de paramètres (éventuellement vide), une valeur de retour (éventuellement vide, mot-clé `void`) et un corps (suite d'instructions). Soit l'objet o_1 possédant l'attribut `attribut1` et la méthode `methode1()` qui change la valeur de `attribut1`, on aura en Java :

```
boolean attribut1 ; /* initialisée par défaut à false */
void methode1(){ attribut1 = true ; }
```

Ici la signature est `void methode1()` et le corps est { `attribut1 = true ;` }.
En C++, on aura exactement la même chose, exception faite du type `boolean` remplacé par le type `bool`.

Appel d'une méthode L'objet o_2 peut alors envoyer le message

```
o1.methode1();                    /* et la valeur de attribut1 deviendra true */
```

L'appel d'une méthode se fait donc tout simplement en mettant l'objet suivi d'un point suivi du nom de la méthode et de sa liste d'arguments (chaque argument étant alors associé au paramètre correspondant ; *respect de l'ordre de déclaration*). Considérons par exemple la méthode suivante liée à l'objet o_1 :

```
int methode2(int i, float f){ if ((i==10) && (f<=-0.56)) return i+f ; return 0 ;}
```

Ici la signature est `int methode2(int i, float f)` et le corps est { `if ((i==10) && (f<=-0.56)) return i+f ; return 0 ;` }. On dira que `i` et `f` sont des paramètres formels.
Cett méthode devra être appelée en mettant d'abord un entier, puis un flottant dans la liste d'arguments. Par exemple, o_1`.methode2(5,33.3333)` renverra un entier (ici 0). On dira que 5 et 33.3333 sont des arguments (dits aussi paramètres effectifs).
Bien-sûr un objet peut aussi utiliser ses propres méthodes. L'appel se fait alors en donnant uniquement le nom de la méthode et sa liste d'arguments : `methode1()`.

Focus sur le passage de paramètre Les principes du passage de paramètre suivent les règles du langage utilisé.

En Java (comme en C), il n'existe qu'un seul cas possible, le passage par valeur. C'est-à-dire que les paramètres formels sont mis à jour avec les arguments. Dans l'exemple précédent, le paramètre `i` a été mis à jour avec la valeur `5` et le paramètre `f` a été mis à jour avec la valeur `33.3333`.

Prenons un autre exemple en considérant que l'objet o_1 est de type `TypeObjet` et qu'il est utilisé en tant qu'argument lors de l'appel d'une méthode liée à un 3ème objet o_3 ; la déclaration de cette méthode est :

```
void methode3(TypeObjet o){ ... o.methode2(10,0.) ; ...}
```

et son appel se fera par : o_3`.methode3(`o_1`)`

Dans ce cas, la règle du passage de paramètre par valeur s'applique aussi : le paramètre formel `o` est mis à jour avec l'argument o_1 (la valeur de la variable o_1 est recopiée dans `o`). Or cette valeur est l'adresse correspondant à l'objet désigné par o_1 et désormais à partir de `o`, on peut donc atteindre les mêmes choses qu'à partir de o_1. ATTENTION : l'objet ne sera en aucun cas dupliqué par un simple passage de paramètre (seule son adresse mémoire est dupliquée).

En C++, on trouve deux types de passage de paramètre : par valeur ou par référence. Cela sera décrit beaucoup plus en détail lors du cours C++ de l'an prochain. Un exemple de passage par référence :

```
void methode(int &i) {   // i est une référence du paramètre.
    i = 2;    // Modifie le paramètre passé en référence.
}
```

avec l'appel suivant :

```
int i ;
methode(i);
// Après l'appel de methode, i vaut 2.
// L'opérateur & n'est pas nécessaire pour appeler methode
```

Cela correspond au code C suivant :

```
void methode(int * pi) {  // pi est l'adresse de la variable à modifier
    *pi = 2;    // Modifie la variable en passant par son adresse
}
```

avec l'appel suivant :

```
int i ;
methode(&i);
// Après l'appel de methode, i vaut 2.
// et là, il faut mettre l'adresse, donc utiliser le &
```

Une fonction particulière : le `main` Une méthode très particulière permettra d'exécuter un programme Java. Il s'agit de la méthode `main`. Son proptotype est imposé et sera expliqué au fur et à mesure du cours :

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

En C++, on a aussi une fonction `main` (dont les paramètres sont optionnels) mais elle n'appartient à aucune classe, ce n'est donc pas une méthode (il ne doit donc y en avoir qu'une seule dans tout votre programme [1]) :

```
int main(int argc, char *argv[])
```

2.4 Les classes

Elles correspondent à la définition d'un ensemble d'attributs et d'un ensemble de méthodes regroupés tous ensemble et destinés à être manipulés uniquement au travers d'un objet. On parle d'"encapsulation". C'est cette notion qui explique la notation utilisée pour l'appel d'une méthode d'un objet par un autre objet.
La notion de classe permet ainsi de définir un type d'objet, ce type étant non seulement un "nom" mais aussi un ensemble d'attributs et de méthodes qui ne vont exister que si l'objet existe [2].

Un exemple en Java :

```
class Personne {

   /* Les attributs */
   int age ;            /* attribut donnant l'age d'une personne */
   float taille ;       /* attribut donnant la taille d'une personne */
```

1. Alors qu'en Java, on peut en mettre une pour chaque classe.
2. Une version appauvrie de cette notion peut se retrouver dans d'autres langages (par exemple, en C, on peut avec l'instruction `typedef` définir des types avec des attributs et des méthodes ; il suffit que ce nouveau type soit une structure dont les champs sont soit un attribut, soit une méthode (chaque champ "méthode" étant alors un pointeur de fonction).

```
   boolean estVivant ;  /* attribut indiquant qu'une personne est vivante ou pas */
   char sexe ;          /* attribut donnant le sexe d'une personne (M ou F) */
   String nom ;         /* attribut donnant le nom d'une personne */
   Personne pere ;      /* attribut donnant le père d'une personne */
   Personne mere ;      /* attribut donnant la mère d'une personne */

   /* Les méthodes */
   /* les accesseurs : pour acceder aux attributs */
   int getAge() { return age ; }
   float getTaille() { return taille ; }
   ...

   /* les setteurs */
   void setAge(int a) { age = a ; }
   void setTaille(float t) { taille = t ; }
   ...
}
```

Le même exemple en C++ demande un traitement spécifique car il n'est pas possible d'"utiliser" la classe `Personne` à l'intérieur d'elle-même. Il faut donc, comme en C, passer par des pointeurs :

```
class Personne {

   /* Les attributs */
   int age ;            /* attribut donnant l'age d'une personne */
   float taille ;       /* attribut donnant la taille d'une personne */
   bool estVivant ;     /* attribut indiquant qu'une personne est vivante ou pas */
   char sexe ;          /* attribut donnant le sexe d'une personne (M ou F) */
   string nom ;         /* attribut donnant le nom d'une personne */
   Personne * ptrPere ; /* attribut donnant l'adresse de l'objet correspondant au père */
   Personne * ptrMere ; /* attribut donnant l'adresse de l'objet correspondant à la mère */

   /* Les méthodes */
   /* les accesseurs : pour acceder aux attributs */
   int getAge() { return age ; }
   float getTaille() { return taille ; }
   ...

   /* les setteurs */
   void setAge(int a) { age = a ; }
   void setTaille(float t) { taille = t ; }
   ...
}
```

2.5 Les instances

Une classe étant l'équivalent d'un type, on peut donc s'en servir pour définir des objets de ce type-là qui seront alors appelés des "instances de cette classe".

2.5.1 Création d'une instance

Cas du Java En java, il y a deux étapes distinctes :

1. la déclaration : il s'agit du mécanisme qui consiste à déclarer des variables correspondant à une classe donnée.

   ```
   Personne dupont, durand, lagasquie
   ```

Ici 3 variables sont définies comme étant de la classe `Personne` ; il s'agit de `dupont`, `durand` et `lagasquie`. En Java, il n'y a en mémoire que la place nécessaire pour ranger ces 3 variables dont les valeurs seront soit `null`, soit une adresse mémoire ; rappelons que, si ces variables sont des attributs, l'initialisation par défaut de Java fait que ces 3 variables correspondent pour l'instant à des objets "vides" (donc leur valeur est `null`). Si on fait un parallèle avec le langage C, ces trois variables sont des pointeurs qui pour l'instant ne pointent sur rien.

2. l'instanciation : il s'agit du mécanisme qui permet de créer vraiment en mémoire un objet d'une classe donnée et d'affecter son adresse à une variable qui permettra alors d'accéder à l'objet :

   ```
   lagasquie = new Personne();
   ```

   Notons qu'il s'agit d'une création dynamique (un peu comme si on avait fait un `malloc` en C). En effet, cet objet peut continuer à exister en dehors du bloc de programme qui a vu sa création.

   A la suite de cette instanciation, une zone mémoire a été réservée pour un objet de classe `Personne` et cette zone est accessible à partir de la variable `lagasquie`. On pourra alors écrire `lagasquie.nom` ou bien `lagasquie.setTaille(1.67)` (tant que l'instanciation n'est pas faite, il est impossible d'écrire ces choses-là). Si on fait un parallèle avec le langage C, la variable `lagasquie` est un pointeur désormais initialisé avec l'adresse de la zone mémoire où vont se trouver les attributs et les méthodes correspondants.

Pour créer une instance d'une classe, il faut donc un constructeur. On peut définir un ou plusieurs constructeurs par classe. S'il y en a plusieurs, ils seront différentiés par leur signature :

```
class Personne {

   /* Les attributs */
   int age ;            /* attribut donnant l'age d'une personne */
   boolean estVivant ;  /* attribut indiquant si la personne est vivante (true) ou décédée (false) */
   String nom ;         /* attribut donnant le nom d'une personne */

   /* Les méthodes */
   /* les accesseurs : pour acceder aux attributs */
   int getAge() { return age ; }
   ...

   /* les setteurs : pour mettre à jour des attributs */
   void setAge(int a) { age = a ; }
   ...

   /* les constructeurs */
   Personne () { estVivant = true ; }
   Personne (int a, float t, String n) { age = a ; taille = t ; nom = n ; estVivant = true ; }
   Personne (String n, char s) { nom = n ; sexe = s ; estVivant = true ; }
}
```

On pourra donc écrire :

```
     Personne dupont ;                  /* la variable dupont existe mais pas l'objet */
     dupont.setAge(10) ;        /* impossible à exécuter puisque l'objet n'existe pas */
     dupont = new Personne("Dupont", 'M') ;             /* l'objet est créé avec le 2ème
                     constructeur et est désormais accessible par la variable dupont */
     dupont.setAge(10) ;                 /* exécution possible puisque l'objet existe */
     Personne durand = new Personne (20, 1.92, "Durand") ;          /* déclaration de la
variable
                        durand, création de l'objet avec un autre constructeur (le 3ème)
                     et mise à jour de la variable durand avec l'adresse de cet objet */
```

Remarquons que, par défaut en Java, chaque classe dispose aussi d'un constructeur de base implicite n'ayant aucun paramètre et qui est utilisé uniquement quand aucun autre constructeur n'est défini. Voir un exemple en section 3.3 (c'est dû au mécanisme de l'héritage dont nous reparlerons plus tard).

Cas du C++ Etudions maintenant ce qu'il se passe en C++. Dans ce langage-là, la déclaration et l'instanciation peuvent être confondues ou pas.

Déclaration et instanciation confondues : il y a alors "création statique" d'un objet qui existera, comme n'importe quelle variable, tant que le bloc de programme ayant vu sa création existe lui-aussi. Plusieurs cas sont alors possibles : soit il existe des constructeurs définis explicitement, soit il n'y en a aucun.

— Dans le cas où aucun constructeur explicite n'existe, on a, comme en Java, un constructeur implicite sans paramètre et le seul fait de déclarer la variable suffit à appeler ce constructeur par défaut et à créer l'objet :

```
class Personne {

   /* Les attributs */
   string nom ;            /* attribut donnant le nom d'une personne */
   ...

   /* Les méthodes */
   /* les accesseurs : pour acceder aux attributs */
   string getNom() { return nom ; }
   ...

   /* les setteurs */
   void setNom(string n) { nom = n ; }
   ...

   /* pas de constructeurs */
}
```

On pourra alors écrire :

```
Personne p ;                        // constructeur par défaut
p.setNom("Dupont") ;                // maj du nom de p
cout << "son nom: " << p.getNom(); // affichage du nom de p
```

— Et dans le cas où des constructeurs existent, alors l'appel de ces constructeurs peut aussi se faire lors de la déclaration (c'est la signature qui permet de choisir, comme en Java) :

```
class Personne {

   /* Les attributs */
   string nom ;            /* attribut donnant le nom d'une personne */
   ...

   /* le constructeur */
   Personne (string n) { nom = n ; estVivant = true ; }
   Personne (string n, bool etat) { nom = n ; estVivant = etat ; }
}
```

On pourra alors écrire :

```
Personne p1("Dupont") ;             // déclaration + appel constructeur 1
cout << "son nom: " << p1.getNom();  // affichage du nom de p1 (Dupont)
Personne p2("Durand", false) ;      // déclaration + appel constructeur 2
cout << "son nom: " << p2.getNom();  // affichage du nom de p2 (Durand)
```

Cela signifie donc que contrairement à ce qui se passe en Java, la création d'une instance peut se faire en C++ en même temps que la déclaration [3].

Déclaration et instanciation distinctes : En C++, comme en C, on peut aussi faire de l'"allocation dynamique" en utilisant le mot-clé `new` (comme en Java), sauf qu'il faut alors utiliser des variables pointeurs (comme en C) :

```
Personne * ptrMere = new Personne("Marguerite") ;
```

3. En effet, quand on écrit en Java `Personne durand = new Personne (20, 1.92, "Durand") ;`, il y a 3 opérations distinctes : la déclaration de la variable, la création de l'objet puis la mise à jour de la variable.

Le pointeur `ptrMere` permettra alors d'accéder à un objet de type `Personne` qui est ainsi créé dynamiquement et dont le nom est "Marguerite".

Attention, les constructeurs ne sont pas des méthodes ! Ils n'ont pas de type de retour (même pas `void`). D'autre part, les constructeurs de certains objets (tableau, chaînes de caractères) sont un peu particulier dans la manière d'être utilisés. On verra cela au fur et à mesure des exemples.

2.5.2 Suppression d'une instance

Puisqu'on peut créer des objets en instanciant des classes, on devrait aussi pouvoir en supprimer (comme ce qu'on fait par exemple en C en utilisant un `free`). Certains langages proposent effectivement des mécanismes explicites de suppression d'objets alors que d'autres utilisent des mécanismes implicites.

Cas du Java Java fait partir des langages qui utilisent des mécanismes implicites : le "garbage collector" ("ramasse-miette" en français). Cette fonctionnalité de Java permet de nettoyer/libérer la mémoire des objets qui ne sont plus référencés. Par exemple :

```
Personne dupont = new Personne("Dupont", 'M') ;
dupont = null ;
```

A ce stade, l'objet créé par la première instruction n'est plus accessible par le programme et sera donc supprimé de la mémoire sans que l'utilisateur ait quoi que ce soit à faire.

Cas du C++ En C++, on trouve les deux mécanismes de suppression, implicite et explicite.
Soit l'objet a été créé statiquement dans un bloc de programme et quand on sort du bloc l'objet disparaît (comme toute variable déclarée dans ce bloc).
Soit la création a été faite par allocation dynamique ; un objet créé dynamiquement n'est pas lié au bloc de programme dans lequel il a été créé (donc celui où le mot-clé `new` a été utilisé) ; il perdure tant qu'il n'a pas été supprimé explicitement par l'utilisateur (si l'on oublie de libérer des objets dynamiques, la quantité de mémoire allouée sera perdue pour le reste du programme !). La suppression d'un objet créé dynamiquement en C++ se fait avec le mot-clé `delete` :

```
delete ptrMere ;
```

2.6 Un exemple simple en Java

On veut représenter un arbre généalogique.

Les parents d'Alphonse s'appellent Arthur et Elise. Les grands-parents côté paternels (resp. maternels) se nomment Henri et Thérèse (resp. Edouard et Marguerite). Les grands-pères sont décédés alors qu'Alphonse, ses grands-mères et ses parents sont toujours vivants.

Imprimer et distribuer les deux pages suivantes

Exemple Java à distribuer :

```
class Personne {

   /* Les attributs */
   String nom ;         /* attribut donnant le nom d'une personne */
   Personne pere ;      /* attribut donnant le père d'une personne */
   Personne mere ;      /* attribut donnant la mère d'une personne */
   boolean estVivant ;  /* attribut indiquant si la personne est vivante (true)
                           ou décédée (false) */

   /* Les méthodes */
   /* les accesseurs : pour acceder aux attributs */
   String getNom() { return nom ; }
   Personne getMere() { return mere ; }
   Personne getPere() { return pere ; }
   boolean vivant() { return estVivant ; }

   /* les setteurs */
   void setNom(String n) { nom = n ; }
   void setMere(Personne m) { mere = m ; }
   void setPere(Personne p) { pere = p ; }
   void decede() { estVivant = false ; }

   String toString() {
      String s = nom ;
      if (!estVivant) s = s+ " -décédé(e)- " ;
      s = s+" a pour mère ("+mere+") et pour pere ("+pere+")" ;
      return s ;  }

   /* les constructeurs */
   Personne (String n) { nom = n ; estVivant = true ; }
   Personne (String n, Personne p, Personne m) {
      nom = n ; estVivant = true ; mere = m ; pere = p ; }
}

class ArbreGen {

   /* Les attributs */
   Personne feuilleArbre ;

   /* les constructeurs */
   ArbreGen (Personne p) { feuilleArbre = p ; }

   /* Les méthodes */
   String toString() { return ("Arbre généalogique: "+feuilleArbre) ; }

   /* le programme principal */
   public static void main (String args[]) {
      Personne p = new Personne("Alphonse");
      ArbreGen a = new ArbreGen(p) ;
      a.feuilleArbre.setMere(new Personne("Elise",new Personne("Edouard"),
                                                  new Personne("Marguerite")));
      a.feuilleArbre.setPere(new Personne("Arthur",new Personne("Henri"),
                                                   new Personne("Thérèse")));
      a.feuilleArbre.getMere().getPere().decede();
      a.feuilleArbre.getPere().getPere().decede();
```

```
      System.out.println(a) ;  }
}
```

Résultat attendu (hors la mise en forme) :

```
Arbre généalogique:
Alphonse a pour mère (Elise a pour mère (Marguerite a pour mère (null)
                                                  et pour pere (null)
                                        )
                              et pour pere (Edouard -décédé(e)-  a pour mère (null)
                                                  et pour pere (null)
                                        )
                     )
        et pour pere (Arthur a pour mère (Thérèse a pour mère (null)
                                                 et pour pere (null)
                                          )
                               et pour pere (Henri -décédé(e)-  a pour mère (null)
                                                 et pour pere (null)
                                          )
                     )
```

Remarques à faire sur :

- les fichiers source (un par classe avec la contrainte de nom),
- la compilation/édition de lien puis exécution en Java (*javac* et *java*),
- la fonction `main` (il peut y avoir une dans chaque fichier source),
- l'utilisation des constructeurs (renvoie un objet même si ce n'est pas indiqué dans la signature),
- le fait que tous les constructeurs ont le même nom,
- l'accès aux attributs, aux méthodes (avec l'enchaînement des noms de variable, attribut, méthode),
- l'utilisation de la bibliothèque Java (avec `import java.lang.* ;` pour accéder à `System.out.println` et à `String`).

Puis pour finir, dire que, si on compile tel quel, on va avoir des erreurs dues à des problèmes d'accès et d'héritage :

```
ArbreGen.java:10: error: toString() in ArbreGen cannot override toString() in Object
    String toString() { return ("Arbre généalogique: "+feuilleArbre) ; }
           ^
  attempting to assign weaker access privileges; was public
```

L'héritage, on verra cela plus tard (voir la section 4.4). Par contre, les problèmes d'accès seront traités dans le chapitre suivant (voir la section 3.1).

2.7 Le même exemple simple en C++

En rouge, les différences par rapport au Java.

Imprimer et distribuer les deux pages suivantes

Exemple C++ à distribuer :

```
class Personne {

   /* Les attributs */
   int age ;            /* attribut donnant l'age d'une personne */
   float taille ;       /* attribut donnant la taille d'une personne */
   bool estVivant ;  /* attribut indiquant qu'une personne est vivante ou pas */
   char sexe ;          /* attribut donnant le sexe d'une personne (M ou F) */
   string nom ;         /* attribut donnant le nom d'une personne */
   Personne * pere = NULL ; /* attribut accédant au père d'une personne */
   Personne * mere = NULL ; /* attribut accédant à la mère d'une personne */

   /* Les méthodes */
   /* les accesseurs : pour acceder aux attributs */
   int getAge() { return age ; }
   float getTaille() { return taille ; }
   string getNom() { return nom ; }
   bool vivant() { return estVivant ; }
   Personne* getMere() { return mere ; }
   Personne* getPere() { return pere ; }

   /* les setteurs */
   void setAge(int a) { age = a ; }
   void setTaille(float t) { taille = t ; }
   void setNom(string n) { nom = n ; }
   void decede() { estVivant = false ; }
   void setMere(Personne* m) { mere = m ; }
   void setPere(Personne* p) { pere = p ; }

   string toString() {
      string s = nom ;
      if (!estVivant) s = s+ " -décédé(e)- " ;
      if (mere != NULL) s = s+" a pour mère ("+mere->toString()+")";
      if (pere != NULL) s = s +" et pour pere ("+pere->toString()+")" ;
      return s ;
   }

   /* les constructeurs */
   Personne (string n) { nom = n ; estVivant = true ; }
   Personne (string n, Personne * p, Personne * m) {
      nom = n ; estVivant = true ; mere = m ; pere = p ; }
};

class ArbreGen {

   /* Les attributs */
   Personne * feuilleArbre ;

   /* les constructeurs */
   ArbreGen (Personne * p) { feuilleArbre = p ; }

   /* Les méthodes */
   string toString() {
       return ("Arbre généalogique: "+feuilleArbre->toString()) ; }
} ;
```

```
/* le programme principal */
int main(int argc, char *argv[]) {
   Personne * p = new Personne("Alphonse");
   ArbreGen a (p) ;
   a.feuilleArbre->setMere(new Personne("Elise",new Personne("Edouard"),
                                                 new Personne("Marguerite")));
   a.feuilleArbre->setPere(new Personne("Arthur",new Personne("Henri"),
                                                  new Personne("Thérèse")));
   a.feuilleArbre->getMere()->getPere()->decede();
   a.feuilleArbre->getPere()->getPere()->decede();
   cout << a.toString() ;
}
```

Résultat attendu (hors la mise en forme) :

```
Arbre généalogique:
Alphonse a pour mère (Elise a pour mère (Marguerite)
                            et pour pere (Edouard -décédé(e)- )
                     )
        et pour pere (Arthur a pour mère (Thérèse)
                             et pour pere (Henri -décédé(e)- )
                     )
```

Remarques à faire sur :

- les deux classes `Personne` et `ArbreGen` peuvent être dans le même fichier source,
- la compilation/édition de lien avec *g++*,
- la fonction `main` (une dans tous les fichiers source),
- l'utilisation des constructeurs (soit en statique, soit en dynamique),
- le fait que tous les constructeurs ont le même nom mais des signature différentes,
- l'accès aux attributs, aux méthodes (avec l'enchaînement des noms de variable, attribut, méthode) soit avec un point soit avec une flèche,
- l'utilisation de la bibliothèque C++ avec `#include <iostream>` et `#include <string>`.

Et enfin signaler qu'il y aura des erreurs de compil dues à un problème d'accès :

```
Personne.cpp: In member function ‘std::__cxx11::string ArbreGen::toString()’:
Personne.cpp:37:11: error: ‘std::__cxx11::string Personne::toString()’ is private
    string toString() {
           ^
Personne.cpp:62:64: error: within this context
        return ("Arbre généalogique: "+feuilleArbre->toString()) ; }
```

Les problèmes d'accès seront traités dans le chapitre suivant (voir la section 3.1).

Chapitre 3

Les droits d'accès

Il s'agit ici de savoir ce qu'un objet a le droit de voir et de modifier. Bien-sûr, il peut voir et modifier tout ce qui le concerne (ses attributs, ses méthodes). Mais comme il faut aussi qu'il puisse envoyer un message à d'autres objets (appeler leurs méthodes, utiliser directement leurs attributs) et recevoir aussi des messages, il faut pour chaque composant de chaque objet définir sa "portée" (qui a le droit d'y accéder).
Les entités pour lesquelles il faut définir des "droits d'accès" sont :

- les classes,
- les attributs,
- les méthodes.

3.1 Les différentes possibilités

Les droits d'accès de chaque composant sont définis par un mot-clé, dit modifieur, placé devant chaque composant en Java [1]. De base, il en existe au moins 2 :

- soit le composant est "privé", c'est-à-dire qu'il ne doit être vu/atteint/modifié que par l'objet auquel il appartient ; dans ce cas, le mot-clé à utiliser est `private` ;
- soit le composant est "public", c'est-à-dire qu'il peut être vu/atteint/modifié par n'importe quel objet ; dans ce cas, le mot-clé à utiliser est `public`.

Notons donc que la méthode `main` définie dans un objet est donc accessible depuis n'importe quel autre objet puisque sa signature comporte le modifieur `public`.

Dans certains langages comme Java, il existe un niveau d'accès intermédiaire qui permet de gérer les droits d'accès par "paquets de classes", ce qu'on appelle des `packages` ; dans ce cas, il suffit de ne pas mettre de mot-clé et alors tous les objets du package auront accès au composant. Par contre, en C++, l'absence de modifieur signifie que le composant est `private`.

Il existe aussi un troisième mot-clé : `protected`. Ce mot-clé offre les mêmes fonctionnalités que l'absence de mot-clé avec une fonctionnalité supplémentaire utilisable en cas d'héritage [2]. Ce modifieur existe aussi en C++ avec un sens similaire (à approfondir l'an prochain).

Si on reprend l'exemple précédent, on peut mettre en Java :

Imprimer et distribuer les deux pages suivantes

1. En C++, l'organisation est différente puisque les composants sont organisés en fonction de leur droit d'accès, voir exemple ci-après.
2. Le mot-clé `protected` autorise toutes les classes dérivées à accéder au composant, ce qui n'est pas le cas de l'absence de mot-clé.

Exemple Java à distribuer :

```
public class Personne {

   /* Les attributs */
   private String nom ;         /* attribut donnant le nom d'une personne */
   private Personne pere ;      /* attribut donnant le père d'une personne */
   private Personne mere ;      /* attribut donnant la mère d'une personne */
   private boolean estVivant ;  /* attribut indiquant si la personne est vivante (true)
                                        ou décédée (false) */

   /* Les méthodes */
   /* les accesseurs : pour acceder aux attributs */
   public String getNom() { return nom ; }
   public Personne getMere() { return mere ; }
   public Personne getPere() { return pere ; }
   public boolean vivant() { return estVivant ; }

   /* les setteurs */
   public void setNom(String n) { nom = n ; }
   public void setMere(Personne m) { mere = m ; }
   public void setPere(Personne p) { pere = p ; }
   public void decede() { estVivant = false ; }

   public String toString() {
      String s = nom ;
      if (!estVivant) s = s+ " -décédé(e)- " ;
      s = s+" a pour mère ("+mere+") et pour pere ("+pere+")" ;
      return s ; }

   /* les constructeurs */
   public Personne (String n) { nom = n ; estVivant = true ; }
   public Personne (String n, Personne p, Personne m) {
      nom = n ; estVivant = true ; mere = m ; pere = p ; }
}

public class ArbreGen {

   /* Les attributs */
   private Personne feuilleArbre ;

   /* les constructeurs */
   public ArbreGen (Personne p) { feuilleArbre = p ; }

   /* Les méthodes */
   public String toString() { return ("Arbre généalogique: "+feuilleArbre) ; }

   /* le programme principal */
   public static void main (String args[]) {
      Personne p = new Personne("Alphonse");
      ArbreGen a = new ArbreGen(p) ;
      a.feuilleArbre.setMere(new Personne("Elise",new Personne("Edouard"), new Personne("Marguerite")));
      a.feuilleArbre.setPere(new Personne("Arthur",new Personne("Henri"), new Personne("Thérèse")));
      a.feuilleArbre.getMere().getPere().decede();
      a.feuilleArbre.getPere().getPere().decede();
      System.out.println(a) ; }
}
```

Exemple C++ à distribuer :

```
class Personne {
   private :
   /* Les attributs */
   bool estVivant ;  /* attribut indiquant qu'une personne est vivante ou pas */
   string nom ;         /* attribut donnant le nom d'une personne */
   Personne * pere = NULL ; /* attribut accédant au père d'une personne */
   Personne * mere = NULL ; /* attribut accédant à la mère d'une personne */

   public :
   /* Les méthodes */
   string getNom() { return nom ; }
   bool vivant() { return estVivant ; }
   Personne* getMere() { return mere ; }
   Personne* getPere() { return pere ; }
   void setNom(string n) { nom = n ; }
   void decede() { estVivant = false ; }
   void setMere(Personne* m) { mere = m ; }
   void setPere(Personne* p) { pere = p ; }
   string toString() {
      string s = nom ;
      if (!estVivant) s = s+ " -décédé(e)- " ;
      if (mere != NULL) s = s+" a pour mère ("+mere->toString()+")";
      if (pere != NULL) s = s +" et pour pere ("+pere->toString()+")" ;
      return s ; }

   /* les constructeurs */
   Personne (string n) { nom = n ; estVivant = true ; }
   Personne (string n, Personne * p, Personne * m) {
      nom = n ; estVivant = true ; mere = m ; pere = p ; }
};

class ArbreGen {
   public :
   /* Les attributs */
   Personne * feuilleArbre ;
   /* les constructeurs */
   ArbreGen (Personne * p) { feuilleArbre = p ; }
   /* Les méthodes */
   string toString() {
       return ("Arbre généalogique: "+feuilleArbre->toString()) ; }
} ;

   /* le programme principal */
   int main(int argc, char *argv[]) {
      Personne * p = new Personne("Alphonse");
      ArbreGen a (p) ;
      a.feuilleArbre->setMere(new Personne("Elise",new Personne("Edouard"),
                                                   new Personne("Marguerite")));
      a.feuilleArbre->setPere(new Personne("Arthur",new Personne("Henri"),
                                                    new Personne("Thérèse")));
      a.feuilleArbre->getMere()->getPere()->decede();
      a.feuilleArbre->getPere()->getPere()->decede();
      cout << a.toString() ;
   }
```

Un autre exemple Java : à expliquer au tableau pour bien voir les différences entre les modifieurs. Dans cet exemple, on a deux packages `Pack1` et `Pack2` avec 3 sources java :

- `Essai.java` dans le package `Pack1`,
- `Essai1.java` dans le package `Pack1`,
- `Essai2.java` dans le package `Pack2`.

Question : pour chaque instruction de chaque `main` dire si cette instruction est possible ou pas et donner les résultats obtenus.

Imprimer et distribuer les deux pages suivantes

- `Essai.java` dans le package `Pack1` :

```
package Pack1 ;
import java.util.* ;
public class Essai {

    int chp0 ;
    private int chp1 ;
    protected int chp2 ;
    public int chp3 ;
    Essai e ;

    public Essai (int chp0,int chp1,int chp2,int chp3) {
       this.chp0 =  chp0;
       this.chp1 =  chp1;
       this.chp2 =  chp2;
       this.chp3 = chp3 ;
    }
    public void maj() {
       this.e = new Essai(chp0+1, chp1+1, chp2+1, chp3+1) ;
    }
    private String toStringReduit() {
       // tous les champs de e sont accessibles ici puisque
       // e et this sont de la meme classe
       return " et ("+e.chp0+", "+e.chp1+", "+e.chp2+", "+e.chp3+")" ;
    }
    public String toString() {
       return "Essai contient "+chp0+", "+chp1+", "+chp2+", "+chp3+toStringReduit()+"." ;
    }
    public static void main(String[] args) {
       Essai ess1 = new Essai(5,10,20,30);
       ess1.maj() ;
       Essai ess2  = new Essai(55,100,200,300);
       ess2.maj() ;
       System.out.println(ess1.toString());
       System.out.println(ess2.toString());
       System.out.println("Le chp0 de Essai est = "+ess1.chp0);
       System.out.println("Le chp1 de Essai est = "+ess1.chp1);
       System.out.println("Le chp2 de Essai est = "+ess1.chp2);
       System.out.println("Le chp3 de Essai est = "+ess1.chp3);
    }
}
```

- `Essai1.java` dans le package `Pack1` :

```
package Pack1 ;
import java.util.* ;
public class Essai1 {

    int chp0 ;
    private int chp1 ;
    protected int chp2 ;
    public int chp3 ;

    public Essai1 (int chp0,int chp1,int chp2,int chp3) {
       this.chp0 =  chp0;
       this.chp1 =  chp1;
       this.chp2 =  chp2;
       this.chp3 = chp3 ;
    }
    public String toString() {
       return "Essai contient "+chp0+", "+chp1+", "+chp2+", "+chp3+"." ;
```

```
    }
    public static void main(String[] args) {
       Essai ess1 = new Essai(5,10,20,30);
       ess1.maj() ;
       Essai1 ess2  = new Essai1(55,100,200,300);
       System.out.println(ess1.toString());
       System.out.println(ess2.toString());
       System.out.println("Le chp0 de Essai est = "+ess1.chp0);
       System.out.println("Le chp1 de Essai est = "+ess1.chp1);
       System.out.println("Le chp2 de Essai est = "+ess1.chp2);
       System.out.println("Le chp3 de Essai est = "+ess1.chp3);
    }
}
```

- `Essai2.java` dans le package `Pack2` :

```
package Pack2 ;
import java.util.* ;
import Pack1.* ;
public class Essai2 {

    int chp0 ;
    private int chp1 ;
    protected int chp2 ;
    public int chp3 ;

    public Essai2 (int chp0,int chp1,int chp2,int chp3) {
       this.chp0 =  chp0;
       this.chp1 =  chp1;
       this.chp2 =  chp2;
       this.chp3 = chp3 ;
    }
    public String toString() {
       return "Essai2 contient "+chp0+", "+chp1+", "+chp2+", "+chp3+"." ;
    }
    public static void main(String[] args) {
       Essai ess1 = new Essai(5,10,20,30);
       ess1.maj();
       Essai2 ess2  = new Essai2(55,100,200,300);
       System.out.println(ess1.toString());
       System.out.println(ess2.toString());
       System.out.println("Le chp0 de Essai est = "+ess1.chp0);
       System.out.println("Le chp1 de Essai est = "+ess1.chp1);
       System.out.println("Le chp2 de Essai est = "+ess1.chp2);
       System.out.println("Le chp3 de Essai est = "+ess1.chp3);
       System.out.println("Le chp0 de Essai2 est = "+ess2.chp0);
       System.out.println("Le chp1 de Essai2 est = "+ess2.chp1);
       System.out.println("Le chp2 de Essai2 est = "+ess2.chp2);
       System.out.println("Le chp3 de Essai2 est = "+ess2.chp3);
    }
}
```

Solution :

Pour `Essai.java` dans le package `Pack1`, on a :

```
package Pack1 ;
import java.util.* ;
public class Essai {

    int chp0 ;
    private int chp1 ;
    protected int chp2 ;
    public int chp3 ;
    Essai e ;
 ...
    public static void main(String[] args) {
       Essai ess1 = new Essai(5,10,20,30);
       ess1.maj() ;
       Essai ess2  = new Essai(55,100,200,300);
       ess2.maj() ;
       System.out.println(ess1.toString());
       System.out.println(ess2.toString());
       // toutes les commandes sont possibles puisqu'on est ds la classe
       System.out.println("Le chp0 de Essai est = "+ess1.chp0);
       System.out.println("Le chp1 de Essai est = "+ess1.chp1);
       System.out.println("Le chp2 de Essai est = "+ess1.chp2);
       System.out.println("Le chp3 de Essai est = "+ess1.chp3);
    }
}

/*  resultat obtenu :
Essai contient 5, 10, 20, 30 et (6, 11, 21, 31).
Essai contient 55, 100, 200, 300 et (56, 101, 201, 301).
Le chp0 de Essai est = 5
Le chp1 de Essai est = 10
Le chp2 de Essai est = 20
Le chp3 de Essai est = 30  */
```

Pour `Essai1.java` dans le package `Pack1`, on a :

```
package Pack1 ;
import java.util.* ;
public class Essai1 {

    int chp0 ;
    private int chp1 ;
    protected int chp2 ;
    public int chp3 ;
...
    public static void main(String[] args) {
       Essai ess1 = new Essai(5,10,20,30);
       ess1.maj() ;
       Essai1 ess2  = new Essai1(55,100,200,300);
       System.out.println(ess1.toString());
       System.out.println(ess2.toString());
       // commande possible puisque l'absence de mot-cle
       //    autorise l'acces a toutes les classes du meme paquetage
       System.out.println("Le chp0 de Essai est = "+ess1.chp0);
       // commande interdite puisque
```

```
        //      chp1 has private access in Essai
        // ==> System.out.println("Le chp1 de Essai est = "+ess1.chp1);
        // commande possible puisque le mot-cle protected
        //    autorise l'acces a toutes les classes du meme paquetage
        System.out.println("Le chp2 de Essai est = "+ess1.chp2);
        System.out.println("Le chp3 de Essai est = "+ess1.chp3);
    }
}

/*  resultat obtenu :
Essai contient 5, 10, 20, 30 et (6, 11, 21, 31).
Essai contient 55, 100, 200, 300.
Le chp0 de Essai est = 5
Le chp2 de Essai est = 20
Le chp3 de Essai est = 30  */
```

Pour `Essai2.java` dans le package `Pack2`, on a :

```
package Pack2 ;
import java.util.* ;
import Pack1.* ;
public class Essai2 {

    int chp0 ;
    private int chp1 ;
    protected int chp2 ;
    public int chp3 ;
...
    public static void main(String[] args) {
        Essai ess1 = new Essai(5,10,20,30);
        ess1.maj();
        Essai2 ess2  = new Essai2(55,100,200,300);
        System.out.println(ess1.toString());
        System.out.println(ess2.toString());
        // commande interdite puisque
        //         chp0 is not public in Pack1.Essai; cannot be accessed from outside package
        // ==> System.out.println("Le chp0 de Essai est = "+ess1.chp0);
        // commande interdite puisque
        //         chp1 has private access in Essai
        // ==> System.out.println("Le chp1 de Essai est = "+ess1.chp1);
        // commande interdite puisque
        //         chp2 has protected access in Pack1.Essai
        // ==> System.out.println("Le chp2 de Essai est = "+ess1.chp2);
        System.out.println("Le chp3 de Essai est = "+ess1.chp3);
        System.out.println("Le chp0 de Essai2 est = "+ess2.chp0);
        System.out.println("Le chp1 de Essai2 est = "+ess2.chp1);
        System.out.println("Le chp2 de Essai2 est = "+ess2.chp2);
        System.out.println("Le chp3 de Essai2 est = "+ess2.chp3);
    }
}

/* resultat obtenu :
Essai contient 5, 10, 20, 30 et (6, 11, 21, 31).
Essai2 contient 55, 100, 200, 300.
Le chp3 de Essai est = 30
Le chp0 de Essai2 est = 55
Le chp1 de Essai2 est = 100
Le chp2 de Essai2 est = 200
```

```
      Le chp3 de Essai2 est = 300  */
```

3.2 Notion de constante

En Java, la notion de constante (classique dans la plupart des langages de programmation) est explicitée à l'aide d'un mot-clé modifieur `final` et peut être vue en programmation objet de différentes manières :

- Pour un attribut ou une variable locale à une méthode, l'utilisation de ce mot-clé signifie que cet élément devient immuable dès qu'il a été initialisé (l'initialisation par défaut n'étant pas comptabilisée) ; quand cet élément est un type primitif, cela veut dire que la valeur ne peut plus changer ; quand cet élément est une référence (l'adresse d'un objet), cela signifie que cette adresse restera inchangée même si le contenu de l'objet évolue.
- Pour une méthode, l'utilisation de ce mot-clé indique que cette méthode ne peut pas être modifiée dans une classe dérivée. Cela est en lien avec la notion d'héritage que nous verrons plus tard.
- Pour une classe, l'utilisation de ce mot-clé indique que cette classe ne peut pas avoir de sous-classe. Cela est aussi en lien avec la notion d'héritage que nous verrons plus tard.

Par exemple :

```
public class Animal {

   /* attribut */
   private final int nbPattes ;

   /* Les méthodes */
   /* les accesseurs : pour acceder aux attributs */
   public int getNbPattes() {return nbPattes ;}

   /* les setteurs */
   public void setNbPattes(int nbp) { nbPattes = nbp ;}

   /* constructeur */
   public Animal (int nbp) {
      nbPattes = nbp ; } /* à partir de maintenant le nb de pattes ne pourra plus évoluer */
}
```

Si on compile ce bout de code, on obtient l'erreur suivante :

```
Animal.java:11: error: cannot assign a value to final variable nbPattes
   public void setNbPattes(int nbp) { nbPattes = nbp ;}
```

Une remarque : le compilateur Java, contrairement à celui utilisé pour le langage C, est très peu permissif.

En C++, la définition d'une valeur constante se fait comme en C avec le mot-clé `const`. Notons que le mot-clé `final` existe aussi en C++ (il est réservé aux méthodes et sera explicité dans le cours de l'an prochain).

3.3 Appartenance à une classe ou une instance

Pour un type d'objet (donc une classe), il peut arriver que certains composants soient les mêmes quelles que soient les instances qui les utilisent. Du coup, il paraît intéressant en terme d'économie de mémoire de ne pas les dupliquer pour chaque instance et de les rattacher directement à la classe. Cette fonctionnalité est mise en place en Java grâce au mot-clé modifieur `static` :

- Pour une méthode, le modifieur `static` indique qu'elle peut être appelée sans instancier sa classe (on écrira juste `NomClasse.methode()`).
- Pour un attribut, le modifieur `static` indique qu'il s'agit d'un attribut de classe, et que sa valeur est donc partagée entre les différentes instances de sa classe.
- on peut même déclarer dans une classe un bloc d'initialisation statique, qui est un bloc d'instruction précédé du modifieur `static`.

Notons que la méthode `main` (dont la signature contient le mot-clé modifieur `static`) est un exemple de méthode de classe. Elle est commune à toutes les instances de la classe dans laquelle elle est définie (pas de duplication de code).
Un exemple :

```
public class Chat {

   /* attribut */
   private static int esperanceDeVie = 15 ; /* l'espérance de vie est la même pour tous les chats */

   /* Les méthodes static */
   public static void setEsperanceDeVie (int e) { esperanceDeVie = e ; }
   public static int getEsperanceDeVie () { return(esperanceDeVie) ; }

   /* Les méthodes non static */
   public void setEsperanceDeVie2 (int e) { esperanceDeVie = e ; }
   public int getEsperanceDeVie2 () { return(esperanceDeVie) ; }

   public static void main (String args[]) {
      Chat tom = new Chat() ;
      Chat grosminet = new Chat() ;
      Chat garfield = new Chat() ;
      System.out.println(Chat.getEsperanceDeVie()) ;
      System.out.println(tom.getEsperanceDeVie2()) ;
      System.out.println(grosminet.getEsperanceDeVie2()) ;
      System.out.println(garfield.getEsperanceDeVie2()) ;
      /* les 3 objets tom, grosminet et garfield partagent la même espérance de vie */

      Chat.setEsperanceDeVie(20) ;
      /* l'espérance de vie des 3 chats a été modifiée */
      System.out.println(Chat.getEsperanceDeVie()) ;
      System.out.println(tom.getEsperanceDeVie2()) ;
      System.out.println(grosminet.getEsperanceDeVie2()) ;
      System.out.println(garfield.getEsperanceDeVie2()) ;

      tom.setEsperanceDeVie2(30) ;
      /* l'espérance de vie des 3 chats a été modifiée */
      System.out.println(Chat.getEsperanceDeVie()) ;
      System.out.println(tom.getEsperanceDeVie2()) ;
      System.out.println(grosminet.getEsperanceDeVie2()) ;
      System.out.println(garfield.getEsperanceDeVie2()) ;

      grosminet.setEsperanceDeVie2(40) ;
      /* l'espérance de vie des 3 chats a été modifiée */
      System.out.println(Chat.getEsperanceDeVie()) ;
      System.out.println(tom.getEsperanceDeVie2()) ;
      System.out.println(grosminet.getEsperanceDeVie2()) ;
      System.out.println(garfield.getEsperanceDeVie2()) ;

      garfield.setEsperanceDeVie2(50) ;
      /* l'espérance de vie des 3 chats a été modifiée */
      System.out.println(Chat.getEsperanceDeVie()) ;
      System.out.println(tom.getEsperanceDeVie2()) ;
      System.out.println(grosminet.getEsperanceDeVie2()) ;
      System.out.println(garfield.getEsperanceDeVie2()) ;
    }
}
```

On obtient le résultat suivant (hors mise en forme) :

```
15
15
15
15

20
20
20
20

30
30
30
30

40
40
40
40

50
50
50
50
```

Remarquons que l'accès (lecture/écriture) à un attribut déclaré `static` peut donc se faire soit par la classe, soit par un des objets de la classe.
Remarquons aussi ici que le constructeur n'a pas été déclaré. Par défaut, chaque classe dispose d'un constructeur de base n'ayant aucun paramètre.
Et enfin, 3ème remarque : l'écriture `System.out.println(...)` signifie donc que la classe `System` définie dans la bibliothèque de Java contient un attribut de classe `out` qui est une instance d'une classe contenant la méthode `println` qui est une méthode d'instance.

Notons que le mot-clé `static` existe aussi en C++ avec un effet similaire à celui obtenu en Java (voir le cours de l'an prochain).

Chapitre 4

Interactions entre objets : appel de méthodes

Nous avons vu qu'un moyen de faire agir un objet sur un autre objet est l'envoi de messages. Cet envoi de message s'implémente sous la forme de l'utilisation de méthodes.

Or pour qu'un objet o_1 de la classe `O1` puisse utiliser/appeler la méthode d'un autre objet o_2 de la classe `O2` , il faut que o_1 "connaisse" o_2. Cette "connaissance" lie deux classes et peut être définie de plusieurs manières (cela rejoint le cours UML vu en début d'année) :

- association : il y a association (dirigée ou pas) entre deux classes [1], lorsqu'une des deux classes sert de type d'attribut à l'autre, et donc qu'apparaît dans le code cette dernière l'envoi d'un message vers la première (par l'appel d'une méthode) ;
- dépendance : il s'agit d'une association affaiblie ; une classe `O1` connaît la classe `O2` mais uniquement durant un temps d'exécution court (par exemple lors de l'appel d'une des méthodes de `O1` ;
- agrégation et composition : il s'agit de représenter les liens existant entre un "tout" et "ses parties" avec une spécificité pour la composition puisque, dans ce cas-là, l'existence "des parties" est liée à l'existence du "tout" ;
- héritage : il s'agit ici de permettre à une classe (dite "classe fille") de récupérer des attributs, des constructeurs et des méthodes d'une autre classe (dite "classe mère").

4.1 Exemple d'association

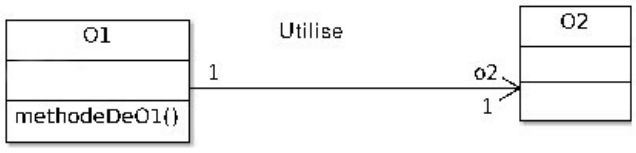
Représentation UML : un trait plein avec ou sans flèche simple au bout entre les deux classes.

```
class O1 {
    O2 o2 ;
    void methodeDeO1() {
        ...
        o2.methodeDeO2() ;  /* envoi message vers O2 */
        ...
    }
    ...
}
```

Les objets de la classe `O1` connaissent la classe `O2` durant toute leur vie.

Le diagramme UML :

1. La notion de "classe d'association" ne sera pas détaillée ici.



4.2 Exemple de dépendance

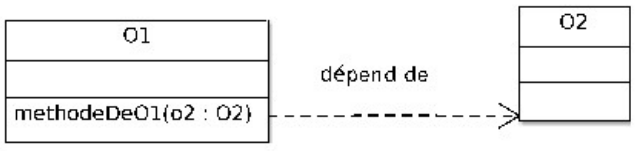
Représentation UML : un trait pointillé avec ou sans flèche simple au bout entre les deux classes.
En général, cela se produit lorsqu'un objet de la classe `O2` est utilisé en tant que paramètre lors de l'appel d'une méthode de la classe `O1`.

```
class O1 {
    ...
    void methodeDeO1(O2 o2) {
        ...
        o2.methodeDeO2() ;  /* envoi message vers O2 */
        ...
    } /* o2 disparaît */
    ...
}
```

Ou bien :

```
class O1 {
    ...
    void methodeDeO1() {
        O2 o2 ;
        ...
        o2.methodeDeO2() ;  /* envoi message vers O2 */
        ...
    } /* o2 disparaît */
    ...
}
```

Les objets de la classe `O1` connaissent la classe `O2` uniquement durant l'exécution de `methodeDeO1`.
Le diagramme UML :



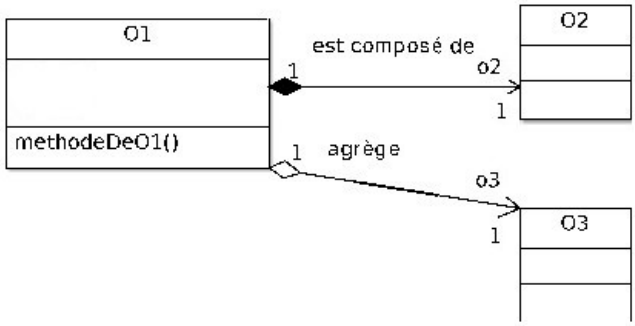
4.3 Exemple d'agrégation et de composition

Représentation UML d'une agrégation : un trait plein avec un losange vide du côté de la classe contenante.
Représentation UML d'une composition : un trait plein avec un losange plein du côté de la classe contenante.

D'autre part, on a aussi une flèche du côté de la classe contenue puisqu'il y a association (la classe contenante utilisant la classe contenue peut donc lui envoyer des messages).

```
class O1 {
   private O2 o2 ;
   private O3 o3 ;
   public O1(O3 o) { /* dépendance avec la classe O3 */
      o2 = new O2() ; /* o2 est créé par O1 : il disparaîtra avec O1 */
                      /* c'est donc une composition */
      o3 = o ;    /* o3 est mis à jour avec o créé ailleurs */
                  /* c'est donc une agrégation */
                  /* qd O1 disparaîtra, o pourrait tjrs exister */
      ...
   }
   void methodeDeO1() {
      o2.methodeDeO2() ;   /* envoi message vers O2 donc association */
      o3.methodeDeO3() ;   /* envoi message vers O3 donc association */
   }
   ...
}
```

Remarquons que la différence entre association et agrégation est purement sémantique. Elle est bien souvent ignorée.
Remarquons aussi que l'association entre `O1` et `O3` cache la dépendance qui existe aussi entre ces deux classes.
Le diagramme UML :



Il existe aussi une notion de composition particulièrement forte qui peut se traduire par la définition d'une classe interne à une autre classe (dans ce cas bien-sûr, les objets correspondant à la classe interne n'existent pas en dehors de la classe contenante) :

```
class O {
   DansO oDansO ;

   class DansO {  /* classe interne donc composition */
      int att ;
      DansO (int i) { att = i ; }
      void methodeDeDansO(int i) { att = i ; }
   }
   public O() { oDansO = new DansO(10) ; }
   void methodeDeO() {
      oDansO.methodeDeDansO(20) ;   /* envoi message vers DansO donc association */
   }
```

```
   public static void main (String args[]) {
      O o = new O() ;
      System.out.println("apres construction : "+o.oDansO.att) ;
      o.methodeDeO() ;
      System.out.println("apres methode : "+o.oDansO.att) ;
    }
}
```

En UML, il n'y a pas vraiment de notation spécifique pour ce type de cas.

4.4 Exemple d'héritage

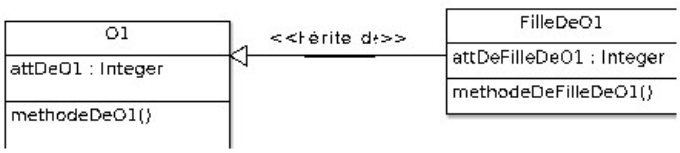
Représentation UML : un trait plein avec une flèche complète vide du côté de la classe mère.

```
class O1 {
   int attDeO1 ;
   ...
   public O1() {
      attDeO1 = 10 ;
   }
   void methodeDeO1() {
      attDeO1 = 20 ;
   }
   ...
}

class FilleDeO1 extends O1 {
   float attDeFilleDeO1 ;
   ...
   void methodeDeFilleDeO1() {
       methodeDeO1() ;
       attDeFilleDeO1 = 15 ;
   }
   ...

   public static void main (String args[]) {
      FilleDeO1 fo1 = new FilleDeO1() ;
      System.out.println("attDeO1 = "+fo1.attDeO1+", attDeFilleDeO1 = "+fo1.attDeFilleDeO1) ;
      fo1.methodeDeFilleDeO1() ;
      System.out.println("attDeO1 = "+fo1.attDeO1+", attDeFilleDeO1 = "+fo1.attDeFilleDeO1) ;
   }
}
```

Ce bout de programme va renvoyer le résultat suivant qui permet de constater qu'à partir de la classe `FilleDeO1`, on a bien accès aux attributs, aux constructeurs et aux méthodes de la classe mère `O1` :

```
attDeO1 = 10, attDeFilleDeO1 = 0.0
attDeO1 = 20, attDeFilleDeO1 = 15.0
```

Le diagramme UML :



Dès qu'on commence à utiliser l'héritage il peut devenir indispensable de distinguer la classe mère de la classe courante. On utilisera alors deux variables spécifiques :

— la variable `super` permet d'accéder à la classe mère de la classe courante (et donc aux attributs, constructeurs et méthodes de la classe mère) ;
— la variable `this` permet d'accéder à la classe courante (et donc aux attributs, constructeurs et méthodes de la classe courante en les distinguant bien de ceux de la classe mère).

Des exemples d'utilisation de ces deux variables seront donnés dans le chapitre sur le polymorphisme (chapitre 6).

Retour sur les droits d'accès : en cas d'héritage, il est important de bien voir l'action des modifieurs. On reprend donc l'exemple traité en section 3.1, avec les deux packages `Pack1` et `Pack2`, et en rajoutant une classe fille :

- `Essai.java` dans le package `Pack1` :

```
package Pack1 ;
import java.util.* ;
public class Essai {

    int chp0 ;
    private int chp1 ;
    protected int chp2 ;
    public int chp3 ;
    Essai e ;

    public Essai (int chp0,int chp1,int chp2,int chp3) {
       this.chp0 =  chp0;
       this.chp1 =  chp1;
       this.chp2 =  chp2;
       this.chp3 = chp3 ;
    }

    public void maj() {
       this.e = new Essai(chp0+1, chp1+1, chp2+1, chp3+1) ;
    }

    private String toStringReduit() {
       // tous les champs de e sont accessibles ici puisque
       // e et this sont de la meme classe
       return " et ("+e.chp0+", "+e.chp1+", "+e.chp2+", "+e.chp3+")" ;
    }

    public String toString() {
       return "Essai contient "+chp0+", "+chp1+", "+chp2+", "+chp3+toStringReduit()+"." ;
    }

    public static void main(String[] args) {
       Essai ess1 = new Essai(5,10,20,30);
       ess1.maj() ;
       Essai ess2  = new Essai(55,100,200,300);
       ess2.maj() ;
       System.out.println(ess1.toString());
       System.out.println(ess2.toString());
       // toutes les commandes sont possibles puisqu'on est ds la classe
       System.out.println("Le chp0 de Essai est = "+ess1.chp0);
       System.out.println("Le chp1 de Essai est = "+ess1.chp1);
       System.out.println("Le chp2 de Essai est = "+ess1.chp2);
```

```
        System.out.println("Le chp3 de Essai est = "+ess1.chp3);
    }
}

/*  resultat obtenu :
Essai contient 5, 10, 20, 30 et (6, 11, 21, 31).
Essai contient 55, 100, 200, 300 et (56, 101, 201, 301).
Le chp0 de Essai est = 5
Le chp1 de Essai est = 10
Le chp2 de Essai est = 20
Le chp3 de Essai est = 30
*/
```

- `Essai2Fille.java` dans le package `Pack2` :

```
package Pack2 ;
import java.util.* ;
import Pack1.* ;

public class Essai2Fille extends Essai {

    public Essai2Fille (int chp0,int chp1,int chp2,int chp3) {
        super(chp0,chp1,chp2,chp3) ;
    }

    public String toString() {
        return super.toString() ;
    }

    public static void main(String[] args) {
        Essai2Fille ess1 = new Essai2Fille(5,10,20,30);
        ess1.maj() ;
        System.out.println(ess1.toString());
        // commande interdite puisque
        //         chp0 is not public in Pack1.Essai; cannot be accessed from outside package
        // l'absence de mot-cle autorise l'acces aux classes du meme paquetage mais pas
        // aux classes filles si elles sont dans un autre paquetage
        // ==> System.out.println("Le chp0 de Essai2Fille est = "+ess1.chp0);
        // commande interdite puisque
        //         chp1 has private access in Essai
        // ==> System.out.println("Le chp1 de Essai2Fille est = "+ess1.chp1);
        // commande possible puisque le mot-cle protected
        //    autorise l'acces a toutes les classes du meme paquetage
        //    et a toutes les classes filles
        System.out.println("Le chp2 de Essai2Fille est = "+ess1.chp2);
        System.out.println("Le chp3 de Essai2Fille est = "+ess1.chp3);
    }
}

/* resultat obtenu :
Essai contient 5, 10, 20, 30 et (6, 11, 21, 31).
Le chp2 de Essai2Fille est = 20
Le chp3 de Essai2Fille est = 30
*/
```

Chapitre 5

Gestion des erreurs

Des erreurs peuvent survenir lors de l'exécution d'un programme :
- division par 0,
- fichier introuvable,
- accès à une case d'un tableau qui n'existe pas,
- accès à un objet non instancié,
- conversion d'une chaîne de caractères en nombre impossible,
- etc.

Afin de gérer au mieux ces erreurs, plusieurs langages de programmation utilisent des *exceptions*, y compris le java.
Les exceptions permettent d'interrompre le déroulement du programme et de rattraper l'erreur.

5.1 Rattraper une exception

```
int methodeDivision(int numerateur, int denominateur) {
   return numerateur / denominateur;
}
```

Si on appelle cette méthode avec comme paramètres `methodeDivision(2,0)`, il va y avoir une division par zéro. Une exception va automatiquement être levée. Le programme va alors s'arrêter et l'erreur suivante va être affichée :
`Exception in thread "main" java.lang.ArithmeticException: / by zero`
Pour éviter que le programme ne s'arrête, on peut rattraper l'erreur avec un `try/catch` :

```
int methodeDivision(int numerateur, int denominateur) {
      try {
         return numerateur / denominateur;
      } catch(ArithmeticException e) {
            System.out.println("Division par 0 interdite");
            return 0;
      }
   }
```

Ce code, au lieu de s'arrêter, affiche le message d'erreur et continue : l'exception a été rattrapée.
On a utilisé pour cela un `try/catch` qui se construit de la manière suivante :

```
try {
  /*
      bloc de code
      exécution normale
  */
```

```
} catch(ExceptionARattraper e) {
   /*
    bloc de code
    exécution en cas d'exception
   */
}
```

Le fonctionnement de `try/catch` est le suivant : dans le bloc `try`, on place le code susceptible de produire une exception.
Si une exception est produite dans ce bloc, alors le code passe automatiquement et directement dans le bloc `catch` qui correspond à l'exception qui a été levée ; la suite du `try` n'est pas exécutée.
Dans le cas précédent, comme l'exception levée est une `ArithmeticException`, on a fait un `catch(ArithmeticException)`.
Si aucun bloc `catch` ne correspond à l'exception qui a été levée, alors l'exception n'est pas rattrapée.
On peut également rattraper plusieurs exceptions de différents types :

```
try {
  /*
      bloc de code
      exécution normal
  */
} catch(ExceptionType1 e) {
   /*
    bloc de code
    exécution en cas d'exception de type 1
   */
} catch(ExceptionType2 e) {
   /*
    bloc de code
    exécution en cas d'exception de type 2
   */
} catch(ExceptionType3 e) {
   /*
    bloc de code
    exécution en cas d'exception de type 3
   */
}
```

5.2 Lever une exception

L'exception `ArithmeticException` a été levée automatiquement, mais on peut aussi créer soi-même une exception.
Si par exemple vous faites une méthode `factorielle` qui a pour signature `int factorielle(int i)`, que faire si `i` est négatif ?

```
int factorielle(int i) {
   if(i == 0)
      return 1;
   else
      return i*factorielle(i-1);
    }
```

On pourrait retourner une valeur spéciale (-1), mais ce ne serait pas propre. On peut à la place lever une exception :

```
int factorielle(int i) throws ArithmeticException {
   if(i < 0)
      throw new ArithmeticException();
```

```
   else if(i == 0)
      return 1;
   else
      return i*factorielle(i-1);
    }
```

A présent, une exception est levée si le paramètre est négatif. Pour avertir les utilisateurs éventuels de cette méthode qu'elle est susceptible de lever une exception `ArithmeticException`, on rajoute à côté de la déclaration de la méthode `throws ArithmeticException`.
On peut alors rattraper cette exception comme précédemment :

```
try {
   factorielle(-1);
} catch(ArithmeticException except) {
   System.out.println("Le paramètre de la fonction factorielle doit être positif");
}
```

5.3 Faire remonter une exception

On revient au premier code :

```
int methodeDivision(int numerateur, int denominateur) {
      try {
         return numerateur / denominateur;
      } catch(ArithmeticException except) {
            System.out.println("Division par 0 interdite");
            return 0;
      }
   }
```

Le problème ici est, qu'en cas d'erreur, on renvoie 0. Il faudrait peut-être rattraper l'erreur autre part (par exemple dans la fonction appelante) afin de la rattraper de manière plus intelligente.
Et c'est possible de ne pas rattraper une exception pour la passer à la méthode appelante (on dit que l'exception "remonte") :

```
int methodeDivision(int numerateur, int denominateur) throws ArithmeticException {
   return numerateur / denominateur;
}

boolean redemande;
do {
   int i = demandeValeurALUtilisateur();
   int j = demandeValeurALUtilisateur();

   redemande = false;

   try {
      methodeDivision(i, j)
   } catch(ArithmeticException e) {
      redemande = true;
   }
} while(redemande)
```

A présent, on redemande à l'utilisateur deux nombres jusqu'à ce que la division soit possible.

5.4 Le bloc `finally`

Le bloc `finally` peut être ajouté à la fin de blocs `try/catch`.
Qu'il y ait une exception ou pas, le bloc `finally` sera exécuté. C'est pratique pour y placer le code qui libère la mémoire, ferme les fichiers, etc. Cela évite de dupliquer du code dans le `try` et dans le `catch`.

```
// ouverture d'un fichier
try {
   // écriture dans le fichier
} catch(Exception e) {
   // gestion d'une exception
} finally {
   // fermeture du fichier, qu'il y ait eu une exception ou non
}
```

5.5 Créer ses propres exceptions

Une exception est une classe qui hérite de la classe `Exception` (classe fournie par la bibliothèque java).
Par exemple :

```
public class ExceptionStudentNotFound extends Exception
{
...
}
```

5.6 Exceptions contrôlées et non contrôlées

Dès lors qu'une méthode peut lever une exception (ou appelle une méthode qui peut en faire lever une), il faut soit :

- la rattraper avec un `try/catch`,
- la relancer avec `throws` dans le prototype.

Néanmoins, certaines exceptions parmi les plus courantes (`NullPointerException`, `RuntimeException`, ...) sont dites non-contrôlées (*unchecked* en anglais) et si vous ne les rattrapez pas, il n'est pas nécessaire (ni recommandé) de les ajouter au `throws`.
Ces exceptions non-contrôlées sont souvent des erreurs de programmation qu'il faut corriger et pas une mauvaise utilisation d'une méthode.

Chapitre 6

Le polymorphisme

6.1 Les différentes formes de polymorphisme

En C, lorsqu'on appelle une fonction `void ajoute(int)`, on sait exactement de quelle fonction il s'agit car elle est unique dans tout le code source.
En Java, ce n'est pas toujours le cas, le même nom peut appeler du code différent : c'est ce qu'on appelle en général le *polymorphisme*[1].
Ceci peut survenir dans plusieurs situations.

6.2 Le polymorphisme *ad hoc* (surcharge)

En C, si on a écrit une fonction `void ajoute(int)`, on ne peut pas écrire une autre fonction `void ajoute(float)` car elles ont le même nom.
En Java, c'est possible : deux méthodes peuvent avoir le même nom et être différenciées par le type de leurs arguments (et pas par le type de retour, ni par l'existence ou pas d'une levée d'exception). Il s'agit qu'un polymorphisme dit *ad hoc*, aussi appelé *surcharge*.
Exemple :

```
public String ajoute(String a, String b)
{
   return a.concat(b);
}

public int ajoute(int a, int b)
{
   return a + b;
}
```

Lorsqu'on appelle la méthode `ajoute`, ses paramètres permettent de lever l'ambiguïté.
Attention : on ne peut pas différencier deux méthodes uniquement par leur type de retour !

6.3 Polymorphisme d'héritage (redéfinition)

Supposons qu'on ait deux classes, `Animal` et `Chien`. Comme un chien est un animal, `Chien` sera une classe fille d'`Animal`.
`Animal` pourra avoir une méthode `description` qui affichera "Je suis un animal".

1. Attention au cas particulier suivant : deux fonctions ayant une même signature `void ajoute(int)` dans deux classes qui n'ont aucun rapport. Dans ce cas, il ne s'agit pas de polymorphisme. Par exemple, la méthode `void ajoute(int)` de la classe `CompteBancaire` va ajouter de l'argent sur le compte bancaire alors que, la méthode `void ajoute(int)` de la classe `Magicien` va ajouter des points de vie aux magiciens.

```
public class Animal {

   public void description() {
         System.out.println("Je suis un animal");
   }
}
```

Comme `Chien` est en particulier un `Animal`, il va hériter de cette méthode. Mais il a le droit de redéfinir cette méthode, c'est-à-dire de réécrire son contenu (sans modifier sa signature).

```
public class Chien extends Animal {

   public void description() {
         System.out.println("Je suis un chien");
   }
}
```

Il y a à présent deux méthodes qui portent le même nom et ont la même signature : `void description()` dans `Animal` et `void description()` dans `Chien`.
Exécutons le code suivant :

```
Animal animal = new Animal();
Chien chien = new Chien();

animal.description();
chien.description();
```

Il va afficher :

```
Je suis un animal
Je suis un chien
```

Ceci est du polymorphisme *d'héritage*, aussi appelé *redéfinition* : le comportement de la classe `Chien` a été spécialisé par rapport à la classe `Animal`.

6.4 Le polymorphisme paramétrique

Alors que le polymorphisme *ad hoc* permet de créer plusieurs méthodes de même nom, l'idée du polymorphisme paramétrique est le contraire : avoir des paramètres les plus généraux possibles.
Pour cela, nous introduisons des notions supplémentaires : les notions de typage statique et dynamique.

6.4.1 Type statique et dynamique

Comme un `Chien` est un `Animal`, on peut écrire :

```
Animal chien2 = new Chien();
```

En effet, on met dans une variable de classe `Animal` un `Chien`, qui est un animal (c'est comme mettre une valeur entière dans un float : un entier est un cas particulier d'un float).
Quelques définitions essentielles :

- Le *type statique* est celui utilisé à la déclaration de la variable.
- Le *type dynamique* est celui de l'objet référencé par cette variable (c'est-à-dire résultant de l'instanciation).

Donc ici `chien2` a pour type statique `Animal` et pour type dynamique `Chien`.

6.4.2 Définition et exemple d'un polymorphisme paramétrique

Si on utilise la déclaration :

```
Animal chien2 = new Chien();
```

Que se passe-t-il alors si on exécute `chien2.description()` ?

La réponse à cette question n'est pas évidente puisque `chien2` a deux types différents (`Animal` et `Chien`). Il existe toutefois une régle essentielle en java :

la méthode appelée est **toujours celle du type dynamique**.

C'est-à-dire que `chien2.description()` renvoie `Je suis un chien`.

Déclarer `Animal chien2 = new Chien();` ne semble pas très malin ; c'est se compliquer la vie pour rien. Il y a par contre certains cas où c'est pratique, notamment avec les tableaux.
Supposons qu'on veuille faire un zoo : on aurait plein d'animaux et chaque case du tableau pourrait être un animal différent.
Par exemple, si on crée la classe `Chat` :

```
public class Chat extends Animal {

   public void description() {
         System.out.println("Je suis un chat");
   }
}
```

Alors, on pourra créer le tableau suivant :

```
Animal[] zoo = new Animal[2];
zoo[0] = new Chien();
zoo[1] = new Chat();
```

Si on souhaite décrire tous les animaux du zoo, on peut le faire avec une boucle :

```
for(int i = 0; i < 2; i++)
   zoo[i].description();
```

Sera alors affiché :

```
Je suis un chien.
Je suis un chat.
```

On peut également écrire une fonction polymorphique :

```
public void decrisTout(Animal[] zoo)
{
   for(int i = 0; i < zoo.length; i++)
      zoo[i].description();
}
```

Cette fonction est polymorphique car elle peut être appelée avec un tableau de `Chien`, de `Chat`, etc. Une même méthode pourra ainsi fonctionner avec plusieurs classes.

6.5 Un exemple de polymorphisme pour savoir si vous avez tout compris

Soit deux classes Java A et B, B dérivant de A :

```
class A
{
    int f(A a)
    {
        return 1;
    }
}

class B extends A
{
    int f(A a)
    {
        return 2;
    }

    int f(B b)
    {
        return 3;
    }
}
```

Il faut maintenant calculer le retour de chacun de ces appels :

```
public static void main(String[] astrArgs)
{
    A a = new A();
    A ab = new B();
    B b = new B();

    // Partie a
    System.out.println( a.f(a) );
    System.out.println( a.f(ab) );
    System.out.println( a.f(b) );

    // Partie ab
    System.out.println( ab.f(a) );
    System.out.println( ab.f(ab) );
    System.out.println( ab.f(b) );

    // Partie b
    System.out.println( b.f(a) );
    System.out.println( b.f(ab) );
    System.out.println( b.f(b) );
}
```

Les résultats attendus sont les suivants :

```
-- Partie a --
1
1
1
```

```
-- Partie ab --
2
2
2

-- Partie b --
2
2
3
```

Ces résultats sont dûs à la manière dont le code a été "compilé" :

— Suite à l'analyse de la définition de la classe `A`, le compilateur a créé (entre autres) la table des méthodes de `A` :

Table des méthodes de `A`
`f(A)` { `return 1;` }

— Suite à l'analyse de la définition de la classe `B`, le compilateur a créé (entre autres) la table des méthodes de `B` :

Table des méthodes de `B`
`f(A)` { `return 2;` } // Redéfinition, on modifie donc le pointeur
`f(B)` { `return 3;` } // Surcharge, c'est donc une nouvelle méthode

— Ensuite, pour chaque variable du programme, le compilateur définit son type statique et en déduit la méthode à appeler pour chaque instruction du programme (en exploitant que les types statiques puisqu'il ne connait que cela) :

Table de typage des variables		
Variable	type statique	type dynamique
`a`	`A`	pour l'instant inconnu
`ab`	`A`	pour l'instant inconnu
`b`	`B`	pour l'instant inconnu

Table de correspondance des appels de méthodes	
Appel	méthode que le compilateur prévoit d'appeler
`a.f(a)`	`A.f(A)`
`a.f(ab)`	`A.f(A)` // c'est le type statique de ab qui est utilisé
`a.f(b)`	`A.f(A)` // pas de méthode f(B) dans la table de A ! Mais les B sont des A !
`ab.f(a)`	`A.f(A)`
`ab.f(ab)`	`A.f(A)` // c'est le type statique de ab qui est utilisé
`ab.f(b)`	`A.f(A)` // pas de méthode f(B) dans la table de A ! Mais les B sont des A !
`b.f(a)`	`B.f(A)`
`b.f(ab)`	`B.f(A)` // c'est le type statique de ab qui est utilisé
`b.f(b)`	`B.f(B)`

— Et enfin, lors de l'exécution, le type dynamique de chaque variable du programme est défini et affecte le choix de la méthode à appeler pour chaque instruction du programme :

Table de typage des variables		
Variable	type statique	type dynamique
`a`	`A`	`A`
`ab`	`A`	`B`
`b`	`B`	`B`

Table de correspondance des appels de méthodes		
Appel	méthode prévue	méthode vraiment appelée à l'exécution
`a.f(a)`	`A.f(A)`	`A.f(A)`
`a.f(ab)`	`A.f(A)`	`A.f(A)`
`a.f(b)`	`A.f(A)`	`A.f(A)`
`ab.f(a)`	`A.f(A)`	`B.f(A)` // suite au typage dynamique
`ab.f(ab)`	`A.f(A)`	`B.f(A)` // suite au typage dynamique
`ab.f(b)`	`A.f(A)`	`B.f(A)` // suite au typage dynamique
`b.f(a)`	`B.f(A)`	`B.f(A)`
`b.f(ab)`	`B.f(A)`	`B.f(A)`
`b.f(b)`	`B.f(B)`	`B.f(B)`

Remarquons que le typage dynamique a affecté le choix de la méthode mais pas intégralement : la méthode choisie est celle correspondant à la classe du type dynamique mais dans le respect de la méthode choisie au préalable par le compilateur (donc en suivant les types statiques des paramètres). Voir les cas de `ab.f(ab)` et `ab.f(b)` qui correspondent lors de l'exécution à l'appel de la méthode `B.f(A)` et pas `B.f(B)` puisque le compilateur avait déjà sélectionné `f(A)`.

Chapitre 7

Les énumérations

Une énumération est un type de données particulier, dans lequel une variable ne peut prendre qu'un nombre restreint de valeurs. Ces valeurs sont des constantes. Par exemple on peut définir une énumération `JourSemaine` qui pourra prendre les valeurs `LUNDI`, `MARDI`, `MERCREDI`, `JEUDI`, `VENDREDI`, `SAMEDI`, `DIMANCHE`.

7.1 Déclaration d'une énumération

La façon la plus simple de déclarer une énumération consiste à écrire :

```
enum JourSemaine {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE} ;
```

Notons qu'une énumération ne peut pas être locale (c'est-à-dire déclarée dans une méthode).
Ce type de déclaration ne permet pas d'utiliser l'énumération `JourSemaine` ailleurs que dans la classe où elle est définie. Il est donc souhaitable de déclarer une énumération dans un fichier séparé, comme une classe, en remplaçant simplement le mot-clé `class` par le mot-clé `enum`.

```
public enum JourSemaine {  // dans le fichier JourSemaine.java
      LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE
}
```

Notons qu'une énumération est en fait une classe (c'est pour cela qu'on parle de "classe énumération"). Cette classe hérite de la classe `Enum`, qui elle-même hérite de la classe `Object`, comme toutes les classes en Java.

7.2 Les constructeurs

Attention, il est impossible de définir des constructeurs publics dans une classe énumération. En fait, les valeurs d'une énumération sont les seules instances possibles de cette classe. Dans notre exemple, `JourSemaine` comporte uniquement 7 instances : `LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE`. Il n'y aura donc pas besoin d'utiliser le mot-clé `new` pour créer une instance d'une énumération. Par exemple, pour créer une instance de `JourSemaine`, il suffit d'écrire :

```
JourSemaine jour = JourSemaine.JEUDI ;
```

Par contre, il est possible de définir des constructeurs privés pour chacune des valeurs énumérées.
Supposons par exemple que nous souhaitions associer à chacune de nos valeurs énumérées `LUNDI`, `MARDI`, `MERCREDI`, `JEUDI`, `VENDREDI`, `SAMEDI`, `DIMANCHE`, une abréviation : `L`, `Ma`, `Me`, `J`, `V`, `S`, `D`.
Formellement, cela signifie qu'à chaque instance de la classe énumération `JourSemaine`, on souhaite associer une propriété, abréviation, qui prend une valeur particulière pour chacune de ces instances. Cela se fait en utilisant le mécanisme des constructeurs privés (c'est-à-dire un constructeur associé à une instance). Dans le cas de notre exemple, cela va se faire de la manière suivante :

```
public enum JourSemaine {  // dans le fichier JourSemaine.java

     LUNDI("L"),  // def. de la valeur en utilisant le constructeur privé
     MARDI("Ma"),
     MERCREDI("Me"),
     JEUDI("J"),
     VENDREDI("V"),
     SAMEDI("S"),
     DIMANCHE("D");

     // attribut privé (propre à chq instance)
     private String abreviation ;

     // constructeur privé (propre à chq instance)
     private JourSemaine(String abreviation) {
         this.abreviation = abreviation ;
    }

     // accesseur pour accéder à l'attribut privé
     public String getAbreviation() {
         return  this.abreviation ;
    }
}
```

La création de chacun des éléments énumérés va appeler le constructeur privé, ce qui provoquera la mise à jour du champ abreviation pour chacune des instances de notre classe. Et donc, le code suivant :

```
JourSemaine jour = JourSemaine.JEUDI ;
System.out.println("Jour : " + jour + " [" + jour.getAbreviation() + "]") ;
```

produira l'affichage suivant :

```
Jour : JEUDI [J]
```

7.3 Les méthodes

Elles sont définies dans la classe `Enum`.

7.3.1 Méthode `toString()`

La méthode `toString()` de cette classe énumération est surchargée : elle retourne une chaîne de caractères qui porte le nom de la constante considérée. Dans notre exemple, le code :

```
System.out.println("Jour : " + JourSemaine.JEUDI) ;
```

produira l'affichage suivant :

```
Jour : JEUDI
```

7.3.2 Méthode `valueOf()`

La méthode `valueOf(String)` retourne la valeur énumérée à partir de sa chaîne de caractères.
Il existe deux versions de cette méthode, toutes les deux statiques (donc utilisables depuis la classe et non depuis une instance). La première est définie dans la classe énumération `JourSemaine`. Le code :

```
JourSemaine jour = JourSemaine.valueOf("SAMEDI") ;
```

aura pour effet la mise à jour de `jour` avec la valeur `JourSemaine.SAMEDI`.
La deuxième est définie dans la classe `Enum`, et prend le nom de l'énumération en paramètre. Le code :

```
JourSemaine jour = Enum.valueOf(JourSemaine.class, "SAMEDI") ;
```

aura aussi pour effet la mise à jour de `jour` avec la valeur `JourSemaine.SAMEDI`.

7.3.3 Méthode `values()`

La méthode statique `values()` retourne un tableau de toutes les valeurs énumérées disponibles.

```
JourSemaine[] jours = JourSemaine.values() ;
```

7.3.4 Méthode `ordinal()`

La méthode `ordinal()` permet de retrouver le numéro d'ordre d'un élément énuméré, dans la liste de tous les éléments d'une énumération. Le premier numéro d'ordre est 0 (donc dans le cas de l'exemple de l'énumération `JourSemaine`, c'est `LUNDI` qui est en position 0).

```
JourSemaine jour1 = JourSemaine.MARDI ;
System.out.println("Jour1 : " + jour1 + " [" + jour1.ordinal() + "]") ;
JourSemaine jour2 = JourSemaine.JEUDI ;
System.out.println("Jour2 : " + jour2 + " [" + jour2.ordinal() + "]") ;

Jour1 : MARDI [1]
Jour2 : JEUDI [3]
```

7.3.5 Méthode `compareTo()`

La méthode `compareTo()` compare les numéros d'ordre de deux éléments énumérés :

- si les deux éléments sont les mêmes alors l'entier retourné est 0,
- sinon, si le premier élément est placé avant le deuxième dans la liste, alors l'entier retourné est négatif,
- sinon il est positif.

Par exemple, le code :

```
JourSemaine jour1 = JourSemaine.JEUDI ;
JourSemaine jour2 = JourSemaine.SAMEDI ;
System.out.println(jour1.compareTo(jour2)) ;
System.out.println(jour2.compareTo(jour1)) ;
System.out.println(jour1.compareTo(jour1)) ;
```

renverra les valeurs :

```
-2
2
0
```

Remarque : on peut aussi comparer les instances à l'aide d'un == ou avec la méthode `equals()` :

```
JourSemaine jour1 = JourSemaine.JEUDI ;
JourSemaine jour2 = JourSemaine.SAMEDI ;
System.out.println(jour1 == jour2) ;
System.out.println(jour1 == jour1) ;
System.out.println(jour1.equals(jour2)) ;
System.out.println(jour1.equals(jour1)) ;
```

renverra les valeurs :

```
false
true
false
true
```

7.4 Collections d'objets énumérés

Il existe au moins deux classes permettant de faire des collections d'objets énumérés : `EnumSet` et `EnumMap`.

Par exemple, la classe `EnumSet` est une classe abstraite fournie dans le but de créer des `Set` dont les éléments sont des objets énumérés. Elle propose un jeu de méthodes statiques, qui retournent toutes des instances de `EnumSet<E>`, où `E` désigne un type énuméré. Par exemple, le code suivant affiche 3 `Set` différents (un contenant l'ensemble des éléments de l'énumération `JourSemaine`, un autre ne contenant que les jours ouvrés et un dernier correspondant aux jours de weed-end) :

```
EnumSet<JourSemaine> laSemaine = EnumSet.allOf(JourSemaine.class) ;
EnumSet<JourSemaine> joursOuvres = EnumSet.of(JourSemaine.LUNDI,
                                              JourSemaine.MARDI,
                                              JourSemaine.MERCREDI,
                                              JourSemaine.JEUDI,
                                              JourSemaine.VENDREDI) ;
EnumSet<JourSemaine> we = EnumSet.of(JourSemaine.SAMEDI,
                                     JourSemaine.DIMANCHE) ;
System.out.println(laSemaine) ;
System.out.println(joursOuvres) ;
System.out.println(we) ;
```

Les affichages obtenus sont :

```
[LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE]
[LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI]
[SAMEDI, DIMANCHE]
```

Il est conseillé d'aller voir la documentation Java pour avoir plus d'information sur ces classes.

Chapitre 8

Exemple final

Ce exemple correspond à la modélisation d'une bataille navale. Les principes sont les suivants :

- le jeu est composé de 3 équipes et d'un plateau de jeu (qui comprendra en particulier un "dessin" pour visualiser l'état du jeu et le déplacement des navires) ;
- chaque équipe est composée d'une liste de navires et peut être soit un bataillon (statut militaire), soit des pêcheurs (statut neutre) ; on considérera qu'il y a deux équipes militaires et une neutre ;
- les navires sont donc soit des navires de guerre, soit des bateaux de pêche ; il y a 2 types de navire de guerre (les destroyers et les sous-marins) et 1 seul type de bateau de pêche (les chalutiers) ; les destroyers et les chalutiers sont des navires de surface alors que les sous-marins sont des navires de profondeur ;
- une des trois équipes sera jouée par l'ordinateur, alors que les deux autres seront jouées par des humains ; l'attribution des rôles sera faite par tirage au sort lors du lancement de la partie ;
- en cours de partie, chaque équipe devra choisir à tour de rôle un navire dans sa liste et l'action à réaliser (déplacement, tir, pêche) ; les choix des équipes humaines seront faits par interaction avec les joueurs (entrée au clavier) alors que ceux de l'équipe artificielle seront faits par tirage aléatoire ;
- chaque navire est susceptible d'être coulé par un tir provenant d'un navire de guerre ; et les sous-marins sont susceptibles d'être endommagés par les chalutiers (hélices prises dans le chalut) ; dans ce cas, ils ne sont pas coulés mais ils ne peuvent plus se déplacer ;
- les capacités de déplacement et de tir dépendent de chaque navire ; dans chaque case du dessin, il ne peut y avoir au plus qu'un navire de surface et un navire de profondeur ; par souci de simplification, on considérera qu'un navire n'occupe qu'une seule case et donc qu'un seul tir au but suffit à le couler ; de même, pour qu'un sous-marin soit endommagé, il suffit qu'il se trouve sous un chalutier lorsque celui-ci pêche ;
- la fin de la partie se produit dès que l'une des 3 équipes n'a plus de navire ; le gagnant est l'équipe à qui il reste le plus de navires valides (donc non coulés et non endommagés) ; cela peut être l'équipe des pêcheurs.

Du point de vue de l'architecture Java, on trouve les éléments suivants :

- `Jeu` : gestion du Plateau et des 3 Equipes ; répartition initiale du type des joueurs ; enchaînement des coups, récupération des choix de chaque équipe et répercution sur le Plateau pour la visualisation et sur les équipes pour la suite du jeu ; détection et arrêt de la partie ;
- `Plateau` : gestion d'une matrice de CasePlateaux et visualisation de l'impact de chaque coup ;
- `CasePlateau` : détection et gestion de l'occupation (vide ou pas, occupée par quoi) ;
- `Equipe` : gestion d'un ensemble de navires, définition d'une commande suivant le type de l'équipe (si humain alors interface de saisie des commandes, si IA alors tirage au sort des commandes) ;
- `EqBataillon` : Equipe avec capacité de tir (impact sur le type de commande disponible) ;
- `EqPecheur` : Equipe avec capacité de pêche (impact sur le type de commande disponible) ;
- `Commande` : l'ensemble des informations nécessaires à une étape de jeu : l'équipe qui agit, le navire concerné, l'action à faire (déplacement, Tir, Pêche), la direction dans laquelle faire cette action (en cas de déplacement ou de tir) ;
- `Navire` : capacité de déplacement et accès à l'état courant ;
- `NavireSurface` : Navire avec niveau d'occupation en surface ;
- `NavireProfondeur` : Navire avec niveau d'occupation en profondeur ;

— `Destroyer` : NavireSurface avec affinement des capacités de déplacement et de tir (portée), état courant étant soit valide, soit coulé ;
— `SousMarin` : NavireProfondeur avec affinement des capacités de déplacement et de tir (portée), état courant étant soit valide, soit coulé, soit endommagé ;
— `Chalutier` : NavireSurface avec affinement de la capacité de déplacement, état courant étant soit valide, soit coulé.

La répartition entre interfaces, classes et enum pourrait être la suivante :

interfaces : `Joueur`, `JHumain` (hérite de `Joueur`), `JIA` (hérite de `Joueur`) ;

classes : `Jeu`, `Plateau`, `CasePlateau`, `Equipe` (implémente `JHumain` et `JIA`), `EqBataillon` (hérite de `Equipe`), `EqPecheur` (hérite de `Equipe`), `Commande`, `Navire`, `NavireSurface` (hérite de `Navire`), `NavireProfondeur` (hérite de `Navire`), `Destroyer` (hérite de `NavireSurface`), `SousMarin` (hérite de `NavireProfondeur`), `Chalutier` (hérite de `NavireSurface`) ;

enum : `Statut` (militaire ou neutre), `Nature` (humain ou IA), `TypeNav` (chalutier, destroyer ou sous-marin), `Action` (déplacement ou tir ou pêche), `Direction` (les 4 points cardinaux)

A noter aussi au moins deux types d'exception :
— `OccupException` : pour traiter le cas où un déplacement ne serait pas possible car il y a déjà trop de navires dans la case ciblée,
— `LimiteException` : pour traiter le cas où un déplacement ou un tir ne seraient pas possibles car la position visée serait hors du plateau.

Exercice 1 : Proposer un diagramme de classe UML correspondant à ce jeu.

Exercice 2 : Ecrire en Java
— une des exceptions : `OccupException`,
— une des énumérations : `Direction`,
— deux des interfaces : `Joueur` et `JHumain`,
— deux classes : `Equipe` et `EqBataillon`.

Bibliographie

[Bar09] Barbier (Franck). – *Conception orientée objet en Java et C++. Une approche comparative.* – Pearson Education, 2009.

[Ber10] Bersini (Huges). – *La programmation orientée objet. Cours et exercices en UML2.* – Eyrolles, 2010.

[CD01] Cardon (Alain) et Dabancourt (Christophe). – *Initiation à l'algorithmique objet.* – Eyrolles, 2001.

Index