# Chapitre 2

# Les mécanismes de Java

Apprentissage des mécanismes du langage : les héritages du C et les mécanismes objets
- manipulation des types
- boucles et alternatives
- héritages, associations, interfaces et réalisations

## 2.1 Espace de noms, paquetages et directive `import`

Java comporte une API complexe. Au sein de cette API, les classes sont nombreuses et le risque d'homonymie n'est pas exclu. Pour éviter ce problème et afin de faciliter la recherche d'une classe, Java propose une hiérarchie qui comprend des dizaines de paquetages (en anglais *packages*) imbriqués. Cette hiérarchie correspond physiquement à l'organisation matérielle des fichiers contenant les classes (sous forme de byte-code). En pratique, une classe correspondra toujours à un fichier comportant l'extension `.class`. Ce fichier sera placé dans l'un des nombreux répertoires créés lors de l'installation de Java ou d'une de ses extensions. Un grand nombre de ces répertoires figurent dans des fichiers d'archives compressés possédant l'extension `.jar`. En général, l'ensemble des répertoires contenant les classes de l'API et les extensions sont placés dans une variable d'environnement nommée `CLASSPATH`. Dans beaucoup de cas cependant, une application Java contient un fichier batch qui détermine et initialise cette variable.

Un grand nombre de paquetage de l'API sont placés dans le paquetage `java` et ses sous-paquetages. D'autres se trouvent dans `javax`. Pour éviter les homonymies, on conseille généralement aux développeurs de créer une hiérarchie de paquetage unique. En pratique, on emploie le nom de domaine du développeur en commençant par le domaine de niveau 1 : si je voulais distribuer des paquetages Java, je leur donnerais des noms commençant par *net.thoorens*. Des paquetages Ville de Liège commenceraient par *be.liege*.

Toute classe placée dans un paquetage reçoit le nom de ce paquetage à l'aide de la directive `package`, placée au début du fichier. Le nom du paquetage correspond à son chemin relatif, en prenant pour point de départ la racine du projet. Cette racine devra être ajoutée au CLASSPATH si on désire par la suite que d'autres classes puissent y faire référence. Implicitement toutes les classes faisant partie d'un même paquetage se voient mutuellement (à condition qu'elles soient publiques). Par contre, des classes figurant dans des paquetages différents doivent faire l'objet d'une importation explicite. Par exemple, si mon projet comporte trois paquetages `un`, `deux`, `trois`, sachant que `trois` est contenu dans `deux`, voici les directives qu'on trouvera dans les classes des trois paquetages, si elles doivent avoir accès aux classes des deux autres classes :

```
// à placer dans les classes du paquetage un
package un;
import deux.*;
import deux.trois.*;

// à placer dans les classes du paquetage deux
package deux;
import un.*;
import deux.trois.*;

// à placer dans les classes du paquetage trois
package trois;
import un.*;
import deux.*;
```

L'étoile représente toutes les classes d'un paquetage, mais n'inclut pas les sous-paquetages. Les directives des classes du paquetage un sont claires à cet égard. On n'est bien entendu pas obligé d'importer toutes les classes d'un paquetage. C'est une facilité qui ne produit pas un code plus lourd que l'indication explicite de chacune des classes importées. La solution de lister les classes une à une présente l'avantage de visualiser plus clairement les dépendances. Il est conseillé de procéder de cette manière lors du peaufinage final. Signalons enfin que si on n'utilise pas la directive `import`, il reste possible d'utiliser les classes en leur donnant leur nom pleinement qualifié. Par exemple, dans une classe de un, on aura :

```
...
deux.trois.ClasseDe3 c3 = new deux.trois.ClasseDe3();
```

Le compilateur signalera toute classe non trouvée. En général, l'interface de Netbeans est en mesure de nous donner la localisation de la classe inconnue afin d'opérer la correction. Les classes principales (celle de `java.lang`) sont automatiquement importées (c'est le cas de `String` et des classes d'emballage dont il va être question dans la section suivante).

## 2.2 Les types primitif et leurs classes d'emballage

Java distingue trois familles de types : les types dérivés de la classe `Object`, c'est-à-dire toutes les classes de l'API, les types primitifs et les tableaux. Les types primitifs représentent des données numériques, logiques et caractères. La joyeuse confusion que C réalise entre les entiers et les caractères n'a pas cours en Java.

### 2.2.1 Types primitifs

#### booléens ou logiques

Les variables et les expressions logiques (`boolean`) constituent un type séparé des entiers. Elles peuvent prendre les deux seules valeurs `true` et `false` et sont, en outre, susceptible de se combiner à l'aide des connecteurs logiques `&&`, `||` et `!` (voir plus loin). Les opérateurs de comparaison et certaines fonctions produisent des valeurs logiques. À aucun moment, Java n'autorise de remplacer un booléen par un entier.

**entiers**

Java distingue plusieurs types d'entiers, en fonction du nombre d'octets qui les composent. La liaison entre le nom et la taille ne dépend pas de l'implémentation. Un `int` représentera toujours une valeur sur quatre octet. Le fragment de code suivant affichera donc les mêmes valeurs sur n'importe quel système :

```
System.out.println("byte␣:"+Byte.SIZE);        // 8
System.out.println("short␣:"+Short.SIZE);      // 16
System.out.println("integer␣:"+Integer.SIZE); // 32
System.out.println("long␣:"+Long.SIZE);        // 64
```

Les constantes des trois premiers types s'écrivent avec des nombres décimaux normaux. Par contre, les nombres de type `long` doivent se terminer par le suffixe `L`. Les affectations entre les différents types sont possibles tant qu'il n'y a pas de risque de perte de précision. On pourra donc affecter un `byte` dans un `short`. Par contre l'opération inverse provoque une erreur (notamment `int i=4L;` est illégal), quelle que soit la valeur effectivement contenue. On peut supprimer l'erreur de compilation en forçant le transtypage (*casting*) à l'aide d'un opérateur de cast, à condition de savoir ce que l'on fait :

```
int i= 123456;
long l = i; // accepté
byte a = i; // erreur de compilation
byte b=(byte) i; // revient à faire 123456%256
```

Les nombres entiers ne sont pas illimités et les dépassements de capacité ne sont pas détectés. La boucle suivante finira donc par afficher le message « négatif ».

```
for(int i=0, val=1;i<33;i++)
  {
      val*=2;
      if(val<0) System.out.println("négatif");
  }
```

**nombres pseudo-réels**

Java distingue deux types de pseudo-réels `float` et `double`. Le premier type fournit un intervalle de nombres plus petit que le second et offre 8 chiffres significatifs exacts contre 16.

```
System.out.println("float␣:"+Float.SIZE);        // 32
System.out.println("double␣:"+Double.SIZE);      // 64
```

Les constantes s'écrivent obligatoirement avec un point. En l'absence de suffixe `F`, le nombre est supposé `double`, ce qui peut provoquer des erreurs de compilation. Voici par exemple comment initialiser à 0 une variable de type `float`.

```
float f1 = 0.;         // Erreur
float f2 = 0.F;        // accepté
float f3 = (float)0.; // accepté
```

**caractères**

Java est un langage moderne. Il se doit donc de représenter tous les caractères possibles. Le type `char` utilise donc un codage sur 16 bits, de type Unicode. Java est insensible au contexte de la machine hôte pour la gestion des caractères spéciaux. Malheureusement, les fichiers sources ne jouissent pas de cette impartialité. Dans un contexte multi-systèmes, il est donc conseillé d'encoder les caractères spéciaux à l'aide de notations non ambiguës [1] :

```
char a_circ;
a_circ = 'â'; // peu sûr
a_circ = '\u00E2'; // toujours â
```

### 2.2.2 Les classes d'emballage

Certaines circonstances nécessitent la manipulation des nombres comme s'ils étaient des objets. On dispose pour cela d'autant de classes d'emballages qu'il y a de types primitifs (`Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float` et `Double`). Jusqu'à la version 5, ces classes se manipulaient avec beaucoup de lourdeur. Par exemple, pour créer un `Integer` de valeur 5, il fallait utiliser le constructeur de la classe. Pour additionner deux `Integer`, on devait utiliser trois méthodes.

```
// 5 + 6 en Java 2
Integer cinq = new Integer(5);
Integer six = new Integer(6);
Integer onze = Integer.valueOf(cinq.intValue()+six.intValue());
// Version moderne en Java 5
Integer cinq = 5;
Integer six= 6;
onze = cinq+six;
```

Les mécanismes de simplification, *autoboxing* et *autounboxing*, fonctionnent au niveau du code source. Le compilateur se charge de réécrire les anciennes instructions avant de les re-compiler.

Outre la création d'objets correspondant à des types primitifs, les classes d'emballages offrent une série de méthodes statiques, qui s'apparentent en fait à des fonctions et permettent diverses manipulations. Parmi celle-ci, on notera les fonctions permettant de transformer une chaîne en nombre. En voici un exemple :

```
int nd = Integer.parseInt("255");
int nh = Integer.parseInt("FF", 16);
```

Ces classes possèdent également des attributs statiques qui permettent de connaître la taille en bits de leur équivalent primitif (`SIZE`), le plus grand et le plus petit nombres (`MAX_VALUE` et `MIN_VALUE`) et d'autres constantes.

1. On peut trouver une documentation complète sur Unicode sur le site spécialement conçu à cet effet : *http ://unicode.org*. Pour une explication des erreurs résultant du mélange des types d'encodage : *http://www.michelcarrare.com/multimedia/codage-car.php* . Pour une table des caractères UTF-8 du français : *http://www.irisa.fr/faqtypo/unicode/alpha-fr.pdf.*

## 2.3 Syntaxe de base : variables, alternatives et boucles

La syntaxe procédurale de Java est largement inspirée par le C et certaines extensions présentes dans le C++.

### 2.3.1 Commentaires

Le signe `//` introduit un commentaire limité à la fin de la ligne. Une zone de commentaire commence par `/*` et se termine par `*/`. Certaines zones de commentaire commencent par `/**`. Elles sont utilisées pour créer la documentation et utilisent des conventions qui seront vues plus tard.

### 2.3.2 Déclaration et portées des variables

On déclare une variable en faisant précéder son identificateur du nom du type ou de la classe à laquelle la variable est censée appartenir. On peut créer plusieurs variables sur la même ligne.

```
Animal a;
int b,c,d;
```

On peut accompagner la déclaration d'une initialisation :

```
Animal a = new Animal();
int b =14;
```

Il n'existe pas de variables dites globales en Java, c'est-à-dire des variables qui seraient accessibles depuis n'importe quel contexte [2]. On peut distinguer néanmoins quatre niveaux de variables :

1. les attributs se regroupent dans deux catégories :
   - les variables d'instance sont accessibles depuis les constructeurs et depuis les méthodes d'instances, à moins qu'elles n'y soient masquées par une variable locale. Dans ce cas, on peut utiliser `this` pour y accéder. Leur durée de vie est celle de l'objet. Si elles ne sont pas privées, elles sont également accessibles dans certains contextes, à condition de les préfixer à l'aide du nom d'un objet de leur classe.
   - les variables de classes (dites aussi statiques) sont accessibles depuis tous les constructeurs et méthodes (statiques ou non). Lorsqu'elles y sont masquées par une variable locale, on utilise le nom de la classe pour y accéder. Leur durée de vie est celle du programme [3]. Elles sont également accessibles dans certains contextes, à condition de les préfixer à l'aide du nom de la classe.

   Les attributs sont implicitement initialisés avec une valeur assimilée à 0 : 0, 0.0, false, ASCII 0 ou null.

2. Il est possible de les simuler en créant des variables statiques dans une classe publique. Une telle pratique revient néanmoins à se priver des avantages de la programmation objet. On peut aussi envisager de créer un singleton public dont les variables d'instance seraient publiques.

3. C'est le souci d'être complet qui m'amène à parler des variables statiques. Nous reviendrons sur la question dans un autre chapitre.

2. les paramètres des méthodes ou des constructeurs sont définis entre les parenthèses qui suivent le nom de la méthode. Ils reçoivent toujours une valeur initiale au moment de l'appel de la méthode ou du constructeur. Le passage se fait toujours par valeur pour les variables des types primitifs et par référence pour les objets. Java, à l'inverse du C, interdit qu'une variable locale ou de bloc soit définie avec le même nom. Leur durée de vie est celle de la méthode ou du constructeur.
3. les variables locales sont définies dans le corps d'une méthode ou d'un constructeur. Elles peuvent masquer un attribut, mais non un paramètre. Elles ont la même durée de vie que les paramètres. Il est à noter que la déclaration d'une variable étant une instruction exécutable, celle-ci peut figurer au milieu d'autres instructions. Il va de soi que seules les instructions qui les suivent sont autorisées à y accéder. Les variables locales, comme les variables de bloc, ont une valeur initiale indéterminée. Tout lecture d'une variable non initialisée entraîne une erreur de compilation.
4. les variables de bloc sont définies soit dans l'en-tête d'une instruction `for`, soit dans un bloc délimité par des accolades. Les mêmes limitations d'homonymies s'appliquent. Leur portée est limitée au bloc où elles figurent. Elles ont aussi une valeur initiale indéterminée.

```
public class Valeur {

    // variable d'instance (niveau 1)
    int nombre;
    // variable de classe (niveau 1)
    static int base=10;

     // définition d'un paramètre (niveau 2)
    void ecrire(int nombre){
        // La variable d'instance est masquée, on emploie this
        this.nombre=nombre;
        // INTERDICTION de masquer le paramètre
        //int nombre; // illégal
    }

    // définition d'un paramètre (niveau 2)
    static void changerBase(int base){

        // La variable de classe est masquée, on emploie le nom de la classe
        Valeur.base=base;
        // INTERDICTION de masquer le paramètre
        //int base; // illégal
    }

    // variables locales (niveau 3)
    void brol(){
        // il est possible de masquer une variable de classe
        int base;
        // il est possible de masquer une variable d'instance
        int nombre;
        // accès toujours possible aux attributs:
        base=Valeur.base;
        nombre=this.nombre;
```

```
    }

    // variables de bloc (niveau 4)
    void comptage(){
        for(int i=0;i<10;i++)
            System.out.println(i);
        {
            int j;
        }

    }
}
```

### 2.3.3 Affectations et modification des variables

L'opérateur signe = permet d'affecter une valeur à une variable. Cette opération renvoie une valeur de retour qui peut servir à des initialisations en cascades.

```
p=1;
t=u=v=0; // initialise v, puis u puis t
```

Comme C, Java possède de nombreux autres opérateurs qui permettent de modifier la valeur d'une variable. Je cite les principaux.

| | | |
|---|---|---|
| ++ | Incrémentation de la variable | |
| −− | Décrémentation de la variable | |
| += | Ajouter une valeur à une variable | |
| −= | Soustraire une valeur d'une variable | |
| *= | Multiplier une variable par une valeur | |
| /= | Diviser un variable par une valeur | |
| %/ | Traiter la variable modulo N | |

### 2.3.4 Instructions conditionnelles et alternatives

Java connaît trois instructions conditionnelles. L'instruction conditionnelle lie l'exécution d'un bloc à la vérification d'une condition :

```
if(c==3)
{
   System.out.println("c vaut 3");
}
```

Quand il n'y a qu'une seule instruction, on peut supprimer les accolades. Java se montre plus exigeant que C dans l'expression de la condition. C considère comme vraie toute expression non nulle. Java exige une expression logique vraie. Les expressions `c` ou `c=3` qui représentent des valeurs entières sont donc illégales dans le contexte d'une condition. Par contre, si `b` et `c` sont des variables logiques, `b` ou `b=c` (au lieu de `b==c`) seront admis. La deuxième expression rendra probablement l'exécution du programme incorrecte. Java connaît six comparateurs et trois opérateurs logiques principaux.

| | | | | | |
|---|---|---|---|---|---|
| **==** | égalité | **>** | supérieur à | **&&** | ET logique |
| **!=** | différence | **<=** | inférieur ou égal à | **\|\|** | OU logique |
| **<** | inférieur à | **>=** | supérieur ou égal à | **!** | NON logique |

```
if(c>=0 && c<=10) // c est compris entre 0 et 10
```

On peut également utiliser une variante alternative de l'instruction conditionnelle :

```
if(c==3)
{
   System.out.println("c vaut 3");
}
else
{
   System.out.println("c ne vaut pas 3");
}
```

Il est évidemment possible d'imbriquer les instructions conditionnelles :

```
if(i<0)
   System.out.println("i est négatif");
else
   if(i>0)
      System.out.println("i est positif");
   else
      System.out.println("i est null");
```

L'instruction `switch` permet d'examiner une seule variable et d'envisager plusieurs cas :

```
switch(i)
{
  case 1:
  case 2:
     System.out.println("i est petit");
     break;
  case 3:
     System.out.println("i vaut 3");
  case 4:
     System.out.println("i est moyen");
     break;
  case 5:
  case 6:
     System.out.println("i est grand");
     break;
  default:
     System.out.println("i pas contenu dans [1,6]")
}
```

La valeur `default` permet de considérer tous les cas non pris en compte par les `case`. On notera la nécessité de placer des instructions `break` pour séparer les traitements. Si `i` vaut 3, nous verrons s'afficher deux messages.

### 2.3.5 Boucles

Je vais illustrer les boucles avec un simple exemple : afficher les nombres compris entre 1 et 10.

```
// boucle à test initial
int i=1;
while(i<=10)
{
   System.out.printl(i);
   i++;
}
// boucle à test final
i=1;
do
{
   System.out.printl(i);
   i++;
}
while (i<=10)
```

Ces deux boucles se distinguent par la position du test, qui autorise la première version à ne pas s'exécuter si le test se révèle directement faux. Les boucles à test final s'exécutent toujours au moins une fois.

```
// boucle for
for(int j=1;j<=10;j++)
{
   System.out.println(j);
}
```

La boucle `for` se distinguent de la boucle à test initial par la possibilité d'utiliser une variable de bloc défine dans la première clause du `for`. La deuxième clause définit le test qui est examiné à chaque itération, en début de boucle. La troisième clause est un moteur de boucle et s'exécute après la dernière des instructions situées dans les accolades. Si la variable est définie dans la clause d'initialisation, sa durée de vie se limite au corps de la boucle. Il est possible d'avoir plusieurs instructions dans chaque clause, séparées par des virgules. L'instruction suivante effectue la somme des éléments d'un tableau.

```
int somme;
for(int i=0, somme=0;i<tab.length;i++)
{
   somme+=tab[i];
}
```

Une variante de `for,` introduite dans Java 5, permet de parcourir un tableau ou une collection sans gérer d'indice ou d'itérateur.

```
int somme=0;
for(int v : tab)
{
   somme+=v;
}
```

**Remarque sur les opérateurs logiques**

Les opérateurs `&&` et `||` ont un comportement particulier qui fait qu'ils ne sont pas commutatifs. Lorsqu'une expression `A && B` est évaluée et que `A` a la valeur `false`, l'expression `B` n'est pas évaluée. Il en est de même dans `A || B` au cas où `A` est vrai. On peut en voir une application pratique dans la recherche d'une valeur dans un tableau.

```
val=...;
i=0;
while(i<tab.length && tab[i]!= val)
       i++;
```

Le but de la boucle est de tester chacune des cases du tableau. La deuxième partie de l'expression conditionnelle n'a bien sûr de sens que si l'indice `i` est contenu dans les limites définies par l'initialisation du tableau. Si l'opérateur `&&` n'avait pas le comportement décrit, on serait amené à exécuter une comparaison illégale lorsque `i` a atteint une valeur supérieure à celle du dernier indice.

## 2.4 Visibilité des classes, des attributs et des méthodes

Java dispose de modificateurs spécifiques pour déterminer la visibilité des classes, des attributs, des méthodes et des constructeurs [4].

### Visibilité paquetage

En l'absence de spécification, une classe, un attribut ou une méthode sont uniquement visibles par les autres objets ou classes du paquetage. Ce n'est pas la visibilité la plus employée.

### Visibilité `public`

Quand une classe, un attribut ou une méthode sont déclarés `public`, ils sont visibles partout dans l'application. C'est habituellement le cas pour la plupart des classes et pour les méthodes qui correspondent à des opérations correspondant aux aptitudes de la classe. On pourra instancier un objet d'une classe publique dans toute l'application. On pourra faire appel à une méthode publique pour manipuler un objet. Les constructeurs sont généralement publics. Par contre, un attribut public constitue une faille dans l'encapsulation : n'importe qui est capable d'y accéder et de le modifier sans contrôle. On préfère généralement le rendre non public et obtenir sa valeur ou la modifier au travers d'accesseurs, qui rendent possible un contrôle des modifications. L'utilisation des accesseurs permet aussi de rendre une attribut non public accessible en lecture uniquement (on ne crée qu'un accesseur *get*) ou uniquement en écriture (création d'un seul accesseur *set*). Par définition, les accesseurs ne sont jamais privés.

### Visibilité `private`

Quand un attribut ou une méthode sont déclarés `private`, ils ne sont accessibles uniquement que depuis la classe ou un objet de la classe. Cela permet d'accéder aux attributs à l'aide d'accesseurs.

4. Un constructeur peut posséder l'accès `private`, mais dans ce cas, il faut pouvoir disposer d'un autre moyen pour instancier des objets de sa classe. Ce point sera abordé plus tard.

```
public class Visibilite {

    private int valeur;
    private int valeur2;

    public Visibilite(int valeurInitiale) {
       valeur=valeurInitiale;
    }

    // attribut accessible en lecture
    public int getValeur(){
        return valeur;
    }

    // attribut accessible en écriture
    public void setValeur2(int valeur2){
        this.valeur2=valeur2;
    }
}
```

Les méthodes privées sont généralement destinées à réaliser des tâches annexes dont les utilisateurs de la classe n'ont pas besoin de connaître l'existence. Ces méthodes peuvent être modifiées, supprimées ou remplacées par d'autres sans que les utilisateurs de la classe doivent modifier leur programme.

Notons qu'un objet peut accéder aux attributs et méthodes privées d'un autre objet de la classe. La méthode suivante donne un moyen détourné de modifier un attribut privé d'un objet de la classe qui la contient.

```
public void incrementerCompagnon(Visibilite compagnon){
        compagnon.valeur++;
    }
```

Voici un exemple de code qui montre l'impossibilité de modification directe et son contournement [5].

```
Visibilite v1 = new Visibilite(1);
 Visibilite v2 = new Visibilite(150);
 // Accès à l'attribut interdit
 //v1.valeur++;
 System.out.println("Valeur de v1: "+v1.getValeur()); //1
 v2.incrementerCompagnon(v1);
 System.out.println("Valeur de v1: "+v1.getValeur()); //2
```

### Visibilité `protected`

L'une des conséquences de l'encapsulation est que les attributs et les méthodes privées ne sont pas exposées pour les autres objets ou classes. Seule la classe et les objets de la même

5. On suppose ici que le code présenté ici fait partie d'une autre classe. Si nous mettions ces lignes dans une méthode `main` contenue dans la classe `Visibilite`, aucune restriction d'accès ne serait de mise.

classe pourront y accéder. Java entend même classe dans un sens strict, puisque que les objets qui font partie de classes dérivées n'auront pas accès aux éléments privés de la classe mère. La visibilité `protected` contourne l'obstacle en autorisant l'accès à ces attributs et méthodes soit par une classe dérivée ou un objet construit par elle ou par les classes et objets du même paquetage [6].

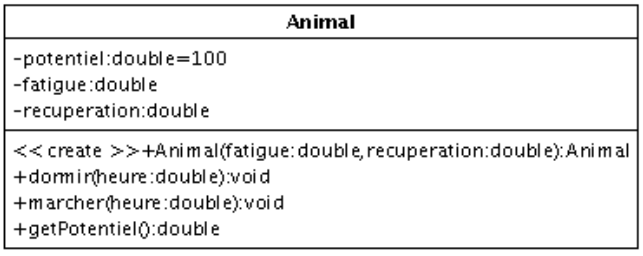
## 2.5 Mécanisme de l'héritage

Une première approche de la programmation objet permet de se rendre compte que le confort et la sécurité résultant de l'utilisation des classes ne suffisent pas à compenser la somme de travail investie dans la création d'une classe. Pour que cela devienne rentable, il faut que la classe puisse servir plusieurs fois. Le mécanisme de l'héritage permet de généraliser la réutilisation des classes en jouant sur deux tableaux : le programmeur va créer de nouvelles classes qui dériveront de classes qu'il a déjà écrite et il va également se servir des classes écrites par d'autres, telles quelles ou en en dérivant ses propres classes. Notons que dans cette deuxième hypothèse, il ne devra pas, du moins en Java, disposer du code source de ces classes « étrangères ».

Pour illustrer notre propos, nous allons développer un exemple basé sur l'observation des animaux. Nous allons constater qu'un animal dispose d'un potentiel d'énergie (accessible via le message `getPotentiel`). Lors de sa création, chaque animal disposera d'un potentiel maximal, exprimé en pourcents. Nous allons faire marcher nos animaux en leur spécifiant le nombre d'heures de leur déambulation. Cette marche va diminuer leur potentiel. Nous considérerons que le potentiel d'un animal ne peut aller plus bas que 0, qui indique qu'il est virtuellement mort. Pour récupérer leur potentiel, les animaux peuvent dormir. Comme un chat a besoin de beaucoup plus de sommeil qu'un chien, nous allons spécifier lors de la création de chaque animal, les coefficients de fatigue et de récupération qui les caractérisent. Par exemple, un chat va user 25% de son potentiel pour chaque heure de marche et en récupérer 5% en une heure de sommeil. Le chien va user deux fois moins et récupérer deux fois plus vite.

```
Animal chat = new Animal(25,5);
Animal chien = new Animal(12.5,10);
```

La définition de la classe animal comportera un créateur permettant d'initialiser les attributs `fatigue` et `recuperation`, un accesseur en lecture sur l'attribut `potentiel` et deux méthodes `marcher` et `dormir`.




Voici le code correspondant à ce premier modèle :

6. Insistons sur la distinction entre visibilité `protected`, qui donne accès aux sous-classes et aux classes du même paquetage, et la visibilité par défaut, ou paquetage, qui donne accès aux seules classes du même paquetage. Les deux notions ne recouvrent pas, puisqu'une classe dérivée peut figurer dans un autre paquetage que sa classe mère.

```
public class Animal {

    private double potentiel = 100.;
    private double fatigue;
    private double recuperation;

    public Animal(double fatigue, double recuperation) {
        this.fatigue=fatigue;
        this.recuperation=recuperation;
    }

    public void dormir(double heure){
        double nouveauPotentiel=potentiel+heure*recuperation;
        if(nouveauPotentiel<=100)
            potentiel=nouveauPotentiel;
        else
            potentiel=100;
    }

    public boolean marcher(double heure){
        double nouveauPotentiel=potentiel-heure*fatigue;
        if(nouveauPotentiel<0){
            potentiel=0;
            return false;
        }
        else{
            potentiel=nouveauPotentiel;
            return true;
        }
    }

    public double getPotentiel() {
        return potentiel;
    }
```

Les méthodes `dormir` et `marcher` prennent soin de ne pas dépasser les limites du potentiel (entre 0 et 100). La valeur renvoyée par `marcher` permet de vérifier que l'animal est encore susceptible d'une quelconque activité.

Voici un exemple d'exécution qui permet de constater qu'un chien qui marche 8 heures récupère son potentiel en 10 heures de sommeil, alors qu'il faut 20 heures de sommeil à un chat pour récupérer 4 heures d'activité [7].

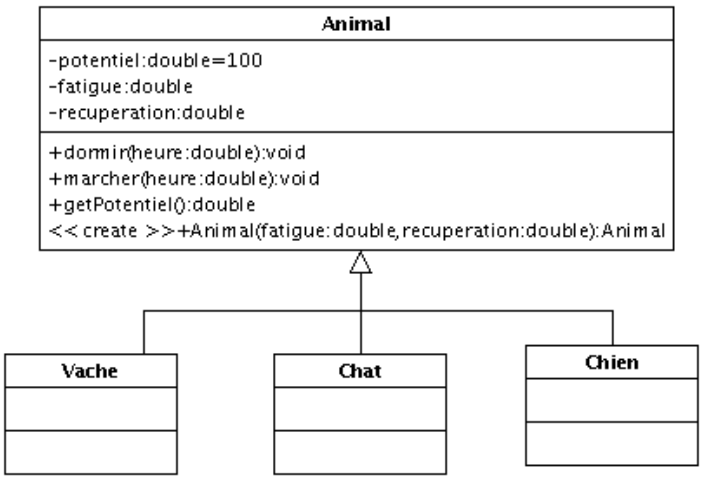
```
Animal chien = new Animal(12.5,10);
chien.marcher(8);
chien.dormir(10);
System.out.println("Potentiel␣du␣chien␣:"+chien.getPotentiel());

Animal chat = new Animal(25,5);
chat.marcher(4);
```

7. Je remercie Vodka, Hermione et Hébus pour leur assistance active dans la détermination des paramètres propres aux chats. Les valeurs propres aux chiens sont malheureusement purement inventées.

```
chat.dormir(20);
System.out.println("Potentiel␣du␣chat␣:"+chat.getPotentiel());
```

Un programme complexe qui devrait faire intervenir de nombreux chats, chiens, vaches et canaris ne pourra se satisfaire d'une telle modélisation. Nous allons créer d'autres classes qui permettront de gérer de manière cohérentes tous les chats, tous les chiens et toutes les vaches.




Nous pourrons dire que la classe *Animal* est la **classe-mère**, tandis que *Vache*, Chat et *Chien* seront des classes **dérivées** ou **classes-filles**. Toutes les instances de chat, de chien et de vache seront *de facto* des instances d'Animal et auront ainsi les mêmes attributs et les mêmes méthodes. En fait, un chat sera un animal et pourra fonctionner de la même façon qu'un animal quelconque.

La définition de ces trois classes va se faire très simplement : nous allons tout simplement définir nos trois classes en disant qu'elles étendent la classe animal. Examinons la classe *Chat :*

```
// première ébauche
public Class Chat extends Animal{
}
```

Cette première ébauche pose deux problèmes :

- l'initialisation des variables de fatigue et de récupération ne semble pas prévue
- le compilateur refuse notre classe en prétextant que le constructeur Animal() n'existe pas.

Une définition comme celle que nous venons de voir ne peut fonctionner que si la classe mère ne possède pas de constructeur ou possède un constructeur sans paramètre. Dans tous les autres cas, nous devrons spécifier au moins un constructeur explicite.

Pour instancier un chat, nous n'avons besoin d'aucun paramètre, il suffit d'appeler le constructeur de la classe mère avec les bons paramètres [8]. L'appel du constructeur se fait à l'aide de `super`.

8. Une autre méthode, qui initialiserait directement les variables d'instance ne fonctionnera pas :

```
public class Chat extends Animal{
   fatigue=20;
   recuperation=5;
}
```

En effet, les deux variables ont une visibilité `private` qui n'autorise pas les classes filles à y accéder directement. Il aurait pour cela fallu préciser une visibilité `protected`.

```
public class Chat extends Animal
   public Chat() {
        super(20, 5);
    }
}
```

Pour définir nos chats et nos chiens, nous allons tout simplement définir des constructeurs avec des appels de `super` adaptés à chaque classe d'animal. Le bout de code suivant permettra de se rendre compte que chaque animal se fatigue selon ses caractéristiques propres.

```
Chat chat = new Chat();
chat.marcher(1);
System.out.println(chat.getPotentiel) // renvoie 75
Animal chien = new Chien();
chien.marcher(2);
System.out.println(chien.getPotentiel) // renvoie 75
```

Le chien se fatigue deux fois moins vite que le chat. On notera aussi que si chat est une variable de type Chat, chien est de type Animal. Tout animal est susceptible de répondre au message `getPotentiel()`. Ce qui importe, c'est que l'objet se comporte en fonction de la classe dont il est une instance. L'objet chien, bien que manipulé au moyen d'une variable de type Animal est bien un Chien.

## 2.6 *Polymorphisme* : surcharges et réécritures

Supposons que nous décidions de faire crier nos animaux. Il va de soi qu'un animal va crier selon l'espèce à laquelle il appartient. Nous ne pouvons donc pas prévoir dans la classe *Animal* le comportement précis de chacun des animaux que nous serons amenés à manipuler. Nous écrirons une méthode `crier()` un rien sommaire.

```
public void crier(){
        System.out.println("Cri␣de␣"+this.getClass().getName());
    }
```

Elle se contente d'afficher « Cri de bestiaire.Chat » si l'animal est un chat. Si nous voulons faire crier nos chats, il s'avère nécessaire de définir une méthode propre au chat. En voici le code :

```
public void crier() {
   System.out.println("miaou");
}
```

On pourra de la même façon faire crier les chiens et les vaches de manière spécifique. Nous avons déjà un procédé qui nous permettait d'avoir plusieurs méthodes homonymes au sein d'une même classe : la surcharge. La surcharge d'une méthode se servait de la signature de la méthode appelée pour déterminer quel code devait exécuter. Ici, la signature est identique :

```
Animal animal = new Animal(10,10);
Chat chat = new Chat();
animal.crier();
chat.crier();
```

On parlera donc de *réécriture* ou de *redéfinition* (en anglais *overriding*). Par ce procédé, un appel similaire de méthode va déclencher dans un cas l'exécution de la méthode spécifique (définie dans la classe Chat) pour tout objet-chat et de la méthode générique (définie dans la classe Animal) pour tout objet-animal ou tout objet d'une classe dérivée de Animal pour laquelle nous n'aurions pas réécrit la méthode `crier()`. La performance du système est déjà remarquable puisque deux appels d'une « même » méthode déclenche l'exécution de deux codes différents. Voici un exemple beaucoup plus impressionnant. Nous allons définir un tableau d'animaux et y placer un chat, un chien et une vache, puis les faire crier.

```
Animal[] animaux= new Animal[3];
animaux[0] = new Chat();
animaux[1] = new Chien();
animaux[2] = new Vache();
for(Animal a:animaux){
     a.crier();
}
```

Ici la même instruction, `a.crier()`, va appeler trois méthodes différentes selon la classe à laquelle appartient l'animal. Si la notion est simple à concevoir pour l'analyste, elle est extrêmement difficile à réaliser en pratique. Java réalise cela de manière transparente, alors que d'autres langages laissent une partie du travail à charge du programmeur (notamment en C++).

Pour parfaire notre système animalier, nous pourrions trouver désastreux qu'on puisse créer un sous-classe d'animaux sans prendre la peine de redéfinir la méthode crier. Il existe un moyen d'imposer cette redéfinition : il faut rendre cette méthode abstraite :

```
//la méthode crier dans la classe Animal
public abstract void crier();
```

Le prix à payer est malgré tout lourd. La présence de cette méthode abstraite nous oblige à rendre la classe elle-même abstraite :

```
public abstract class Animal{ ...
```

Cela nous empêchera dorénavant d'instancier des animaux. Il reste possible de définir des variables de types Animaux, mais nous ne pouvons plus y ranger que des références de chat, de chien ou de vache.

```
Animal animal = new Animal(); // illégal
Animal c = new Chat(); // possible
```

## Exercices

Repartir de la notion de compte bancaire. En fait, il existe deux types de comptes bancaires. Le compte courant permet des retraits et des dépôts, avec un découvert autorisé qui varie selon le client. Par contre, un compte d'épargne produit un intérêt, ne peut pas devenir négatif et, du moins en Belgique, ne peut être utilisé pour effectuer un virement que sur le compte de son titulaire, dans la même banque. Nous allons essayer de remodéliser la notion de compte sur base de trois classes en donnant à chacune des classes filles et à la classe mère, les responsabilités qui sont les siennes.