Université Paul Sabatier - Toulouse III

Conception Orientée Objet avec Unified Modelling Language et Java

Département EEA - Toulouse

Conception Orientée Objet avec Unified Modelling Language et Java

V Albert

Table des matières

Partie 1 : Les bases du langage Java

1 Types de données, variables et tableaux

Il est important de préciser que Java est un langage fortement typé. Toute variable et toute expression possède un type, et chaque type est rigoureusement défini. Qu'elles soient explicites ou par le biais de passage de paramètres transmis lors d'appels de méthodes toutes les affectations subissent une vérification, garantissant la compatibilité des types. Il n'existe pas de contraintes ou de conversions automatiques de types en conflit comme dans certains langages. Le compilateur Java vérifie toutes les expressions et les paramètres pour s'assurer que les types sont compatibles. Toute non-concordance génère une erreur qui doit être corrigée avant que le compilateur termine la compilation de la classe.

1.1 Les types de données primitifs

Java définit huit types primitifs (élémentaires) de données :
- le type booléen `boolean`, qui n'est pas un type entier, et qui peut prendre les valeurs `false` et `true`.
- le type caractère `char`
- les types entiers `byte`, `short`, `int` et `long`
- les types nombres flottants `float` et `double`

Ces types peuvent être utilisés tels quels, pour créer des tableaux ou pour vos propres types de classes.

Les types primitifs représentent des valeurs uniques et non des objets complexes. Bien que Java soit intégralement orientée objet, ce n'est pas le cas des types primitifs. Ils s'apparentent aux types élémentaires existant dans la plupart des autres langages non orientés objets.

Les types primitifs sont définis de manière à avoir une étendue explicite et un comportement mathématique spécifique. Dans les langages comme le C et le C++, la taille d'un entier peut varier en fonction des exigences de l'environnement d'exécution. Pour des raisons de portabilité, tous les types de données ont une étendue strictement définie. Par exemple, un `int` est toujours codé sur 32 bits, quelque soit la plate-forme.

Nom	Longueur	Étendue
long	64	$[-2^{63}, 2^{63}-1]$
int	32	$[-2^{31}, 2^{31}-1]$
short	16	$[-2^{15}, 2^{15}-1]$
byte	8	-128 à 127
double	64	$[1.7e-308, 1.7e308]$
float	32	$[3.410-38, 3.410+38]$
boolean	1	true or false

1.2 Les variables

1.2.1 Déclaration d'une variable

En Java, toutes les variables doivent être déclarées avant d'être utilisées. Sous sa forme élémentaire, une déclaration de variable se présente comme suit :

```
type identifier [ = value][, identifier [= value] ...] ;
```

Le type correspond à l'un des types primitifs de Java ou au nom d'une classe ou d'une interface. L'identificateur désigne le nom de la variable. Vous pouvez initialiser la variable au moyen d'un signe égal et d'une valeur.

```
int a, b, c;         // déclare trois int a, b, et c.
int d = 3, e, f = 5; // déclare trois autres int, en initialisant
                     // d et f.
byte z = 22;         // initialise z.
double pi = 3.14159; // déclare une valeur approximative de pi.
char x = 'x';        // donne à la variable x la valeur 'x'.
```

Java permet d'initialiser des variables dynamiquement, au moyen de n'importe quelle expression valide au moment de la déclaration de la variable.

```
class DynInit {
    public static void main(String args[ ]) {
      double a = 3.0, b = 4.0;
      // c est initialisée dynamiquement
      double c = Math.sqrt(a * a + b * b);
      System.out.println("Hypotenuse is " + c);
    }
}
```

1.3 Conversions de types et transtypage

Il est fréquent d'attribuer une valeur d'un type particulier à une variable d'un autre type. Si les deux types sont compatibles, Java effectue la conversion automatiquement. Par exemple, il est possible de donner une valeur `int` à une variable `long`. Cependant, tous les types ne sont pas compatibles, et certaines conversions ne sont pas implicitement autorisées. Par exemple, aucune conversion n'est définie pour passer d'un `double` à un `byte`.

Pour opérer une conversion entre deux types incompatibles, vous devez effectuer un *cast*, c'est à dire une conversion de type explicite. Il se présente sous la forme :

```
(type-cible) valeur
```

Ici, `type-cible` indique le type vers lequel vous désirez convertir la valeur spécifiée.

Le programme suivant illustre différents types de conversions qui requièrent des *casts* :

```
class Conversion {
  public static void main(String args[ ]) {
    byte b;
    int i = 257;
    double d = 323.142;
    System.out.println("\nConversion of int to byte.");
    b = (byte) i;
    System.out.println("i and b " + i + " " + b);
    System.out.println("\nConversion of double to int.");
    i = (int) d;
    System.out.println("d and i " + d + " " + i);
    System.out.println("\nConversion of double to byte.");
    b = (byte) d;
    System.out.println("d and b " + d + " " + b);
  }
}
```

Ce programme génère la sortie suivante :

```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67
```

1.4 Les tableaux

Un tableau Java est un objet permettant de rassembler sous un même identificateur des données de même type. Sa déclaration est la suivante :

```
Type Tableau[ ]= new Type[n] ou Type[ ] Tableau= new Type[n]
```

Les deux syntaxes sont légales. `n` est le nombre de données que peut contenir le tableau. La syntaxe `Tableau[i]` désigne la donnée n°i où i appartient à l'intervalle $[0, n-1]$. Toute référence à la donnée `Tableau[i]` où i n'appartient pas à l'intervalle $[0, n-1]$ provoquera une exception.

Un tableau à deux dimensions pourra être déclaré comme suit :

```
Type Tableau[ ][ ]= new Type[n][p] ou Type[ ][ ] Tableau= new Type[n][p]
```

La syntaxe `Tableau[i]` désigne la donnée n°i de `Tableau` où i appartient à l'intervalle $[0, n-1]$. `Tableau[i]` est lui-même un tableau : `Tableau[i][j]` désigne la donnée n°j de `Tableau[i]` où j appartient à l'intervalle $[0, p-1]$. Toute référence à une donnée de `Tableau` avec des index incorrects génère une erreur fatale.

2 Opérateurs

2.1 Opérateurs arithmétiques

Les opérateurs arithmétiques sont utilisés dans des expressions mathématiques, de la même manière qu'en algèbre. Le tableau suivant énumère les opérateurs arithmétiques :

Opérateur	Résultat
$+$	Addition
?	Soustraction (également moins unaire)
$*$	Multiplication
$/$	Division
%	Modulo
$++$	Incrémentation
$+=$	Affectation d'addition
$-=$	Affectation de soustraction
$*=$	Affectation de multiplication
$/=$	Affectation de division
$\%=$	Affectation de Modulo
$--$	Décrémentation

Les opérandes des expressions arithmétiques doivent être de type numérique. Il est impossible de les utiliser sur les types `boolean`, mais vous pouvez les utiliser sur les types `char`, dans la mesure où, en Java, ces derniers constituent un sous-ensemble du type `int`.

2.2 Opérateurs bit à bit

Java définit plusieurs opérateurs bit à bit applicables aux types `int`, `long`, `short`, `char`, et `byte`, qui agissent sur chaque bit de leurs opérandes. Ils sont résumées dans le tableau suivant :

Opérateur	Résultat
~	NON bit à bit unaire
&	ET bit à bit
|	OU bit à bit
^	OU bit à bit exclusif
>>	Décalage à droite
>>>	Décalage à droite avec remplissage de zéros
<<	Décalage à gauche
&=	Affectation de ET bit à bit
|=	Affectation de OU bit à bit
^=	Affectation de OU exclusif bit à bit
>>=	Affectation de décalage à droite
>>>=	Affectation de décalage à droite avec remplissage de zéros
<<=	Affectation de décalage à gauche

2.3 Opérateurs de comparaison

Les opérateurs de comparaison déterminent la relation qui unit un opérande à l'autre. Plus précisément, ils définissent une relation d'égalité et de rapport hiérarchique. Ils sont répertoriés dans la liste suivante :

Opérateur	Résultat
==	Égal à
! =	Différent de
>	Plus grand que
<	Plus petit que
>=	Plus grand ou égal
<=	Plus petit ou égal

2.4 Opérateurs logiques booléens

Les opérateurs booléens logiques présentés ci-dessous agissent uniquement sur des opérandes `boolean`. Tous les opérateurs logiques binaires combinent deux valeurs `boolean` pour produire une valeur finale `boolean`.

Opérateur	Résultat
`&`	Et logique
`|`	OU logique
`^`	OU logique exclusif
`||`	OR avec court circuit
`&&`	ET avec court circuit
`!`	NON logique unaire
`&=`	Affectation de ET
`|=`	Affectation de OU
`^=`	Affectation de OU exclusif
`==`	Égal à
`!=`	Différent de
`?:`	Si-alors-non ternaire

2.5 Opérateurs d'affectation

L'opérateur d'affectation correspond au signe d'égalité unique, =. Il s'utilise ainsi :

```
var = expression;
```

Ici, le type de `var` doit être compatible avec le type de `expression`. Il permet de créer une chaine d'affectation :

```
int x, y, z;
x = y = z = 100;
```

2.6 Opérateur ?

L'expression

```
expr_cond ? expr1:expr2
```

est évaluée de la façon suivante :

1. L'expression `expr_cond` est évaluée. C'est une expression conditionnelle à valeur vrai ou faux
2. Si elle est vraie, la valeur de l'expression est celle de `expr1`. `expr2` n'est pas évaluée.
3. Si elle est fausse, c'est l'inverse qui se produit : la valeur de l'expression est celle de `expr2`. `expr1` n'est pas évaluée.

Exemple

```
i=(j>4 ? j+1:j-1);
```

affectera à la variable i : $j+1$ si $j>4$, $j-1$ sinon.

C'est la même chose que d'écrire

```
if(j>4) i=j+1; else i=j-1;
```

mais c'est plus concis.

2.7 Priorité des opérateurs

```
()  [ ]  .
!   ~    ++   - -
*   /    %
+   -
<<  >>
<   <=   >    >=
==  !=
&
^
|
&&
||
?:
=   op=
```

3 Instructions de contrôle

3.1 Instructions de sélection

3.1.1 if

Une instruction `if` se présente sous la forme suivante :

```
if (condition) {actions_condition_vraie;} else {actions_condition_fausse;}
```

Il est important de noter que :
– la condition est entourée de parenthèses.
– chaque action est terminée par point-virgule.
– les accolades ne sont pas terminées par point-virgule.
– les accolades ne sont nécessaires que s'il y a plus d'une action.
– la clause `else` peut être absente.
– il n'y a pas de `then`.

Exemple :

```
if (x>0)     { nx=nx+1;sx=sx+x;} else dx=dx-x;
```

On peut imbriquer les structures de choix :

```
if(condition1)
if(condition2)
    {......}
    else
    {......}
else
{.......}
```

Il peut se poser parfois le problème suivant :

```
public static void main(void){
    int n=5;
    if(n>1)
       if(n>6)
          System.out.println("> 6");
       else System.out.println("<= 6");
}
```

Dans l'exemple précédent, le `else` se rapporte à quel `if` ? La règle est qu'un `else` se rapporte toujours au `if` le plus proche :

`if(n>6)`

dans l'exemple.

Considérons un autre exemple :

```
public static void main(void){
    int n=0;
    if(n>1)
       if(n>6)       System.out.println("> 6");
       else;       // else du if(n > 6) : rien à faire
    else System.out.println("<= 1"); // else du if(n>1)
}
```

Ici nous voulions mettre un `else` au

`if(n>1)`

et pas de `else` au

`if(n>6)`

A cause de la remarque précédente, nous sommes obligés de mettre un `else` au

`if(n>6)`

, dans lequel il n'y a aucune instruction.

3.1.2 switch

Une instruction `switch` se présente sous la forme suivante :

```
switch(expression) {
  case v1:
      actions1;
      break;
  case v2:
      actions2;
      break;
       . .. .. .. .. ..
  default: actions_sinon;
}
```

Il est important de noter que :
- La valeur de l'expression de contrôle, ne peut être qu'un entier ou un caractère.
- l'expression de contrôle est entourée de parenthèses.
- la clause `default` peut être absente.
- les valeurs `vi` sont des valeurs possibles de l'expression. Si l'expression a pour valeur `vi` , les actions derrière la clause `case` sont exécutées.
- l'instruction `break` fait sortir de la structure de cas. Si elle est absente à la fin du bloc d'instructions de la valeur `vi`, l'exécution se poursuit alors avec les instructions de la valeur `vi+1`.

Voici un exemple en algorithmique

```
selon la valeur de choix
    cas 0
      arrêt
    cas 1
      exécuter module M1
   cas 2
      exécuter module M2
   sinon
      erreur<--vrai
fin des cas
```

En Java

```
int choix, erreur;
switch(choix){
    case 0: System.exit(0);
    case 1: M1();break;
    case 2: M2();break;
    default: erreur=1;
}
```

3.2 Instructions de répétitions

3.2.1 while

Une instruction tant que `while` se présente sous la forme suivante :

```
while(condition){
      actions;
}
```

On boucle tant que la condition est vérifiée. La boucle peut ne jamais être exécutée.

Il est important de noter que :
- la condition est entourée de parenthèses.
- chaque action est terminée par point-virgule.
- l'accolade n'est nécessaire que s'il y a plus d'une action.
- l'accolade n'est pas suivie de point-virgule.

3.2.2 do-while

Une instruction tant que `do-while` se présente sous la forme suivante :

```
do{
    instructions;
}while(condition);
```

On boucle jusqu'à ce que la condition devienne fausse ou tant que la condition est vraie. Ici la boucle est faite au moins une fois.

3.2.3 for

Une instruction `for` se présente sous la forme suivante :

```
for(instructions_départ;condition;instructions_fin_boucle){
  instructions;
}
```

On boucle tant que la condition est vraie (évaluée avant chaque tour de boucle). `instructions_départ` sont effectuées avant d'entrer dans la boucle pour la première fois. `instructions_fin_boucle` sont exécutées après chaque tour de boucle. Il est important de noter :

- les 3 arguments du `for` sont à l'intérieur des parenthèses.
- les 3 arguments du `for` sont séparés par des points-virgules.
- chaque action du `for` est terminée par un point-virgule.
- l'accolade n'est nécessaire que s'il y a plus d'une action.
- l'accolade n'est pas suivie de point-virgule.
- les différentes instructions dans `instructions_depart` et `instructions_fin_boucle` sont séparées par des virgules.

Les programmes suivants calculent tous la somme des n premiers nombres entiers.

```
for(i=1, somme=0;i<=n;i=i+1)
    somme=somme+a[i];

for (i=1, somme=0;i<=n;somme=somme+a[i], i=i+1);

i=1;somme=0;
while(i<=n)
{ somme+=i; i++; }

i=1; somme=0;
do somme+=i++;
while (i<=n);
```

Instructions de gestion de boucle sont

- `break` fait sortir de la boucle for, while, do ... while.
- `continue` fait passer à l'itération suivante des boucles for, while, do ... while

Partie 2 : Technologie Objet et Java

4 Utilisation d'UML

4.1 Objets : combinaison de services et de données

Un système orienté objet (OO) comprend un certain nombre d'objets qui interagissent pour réaliser un objectif. Les objets logiciels imitent les objets du monde réel du domaine d'application. Les objets du monde réel peuvent avoir une présence physique ou ils peuvent représenter des entités conceptuelles bien comprises de l'application. Par exemple dans une université en tant qu'application, il y aura des objets logiciels qui représentent les étudiants. De même, il y aura des objets logiciels qui représentent les programmes d'études de l'université même si ces derniers n'ont pas d'existence physique.

Les objets sont caractérisés par un *état* et un *comportement*. L'état d'un objet est l'information permettant de le caractériser. Par exemple, un objet étudiant aura un nom, une date de naissance, et un numéro de matricule universitaire. Un objet représentant un programme d'étude aura un nom, une durée et le nom du responsable de ce programme. Le comportement de l'objet décrit les actions que l'objet s'engage à réaliser. Par exemple nous pourrions demander à un objet étudiant son age, ceci impliquerait que l'objet réalisera un calcul à partir de la date de naissance et la date du jour.

Le comportement d'un objet est décrit par l'ensemble des *opérations* qu'il s'engage à réaliser. Un objet interagit avec un autre en demandant à ce dernier d'exécuter l'une de ses opérations. Cette interaction est accomplie par l'envoi d'un *message* d'un objet à un autre objet. Le premier objet est l'*appelant* (ou *client*) et le second objet est l'*appelé* (ou *serveur*). Les seuls messages qu'un objet peut recevoir sont ceux parmi l'ensemble des opérations que l'objet peut accepter. En UML, les objets et l'appel de message est communément décrit par un diagramme de séquence tel que l'illustre la figure 1. Ici, l'objet université envoie le message `getAge` à un objet étudiant pour obtenir l'age de ce dernier.

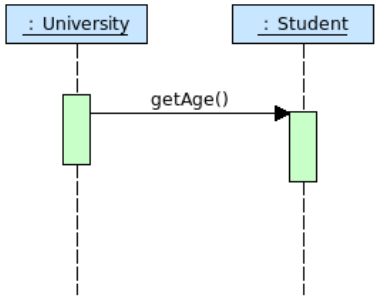

FIGURE 1 – Envoi de message dans un diagramme de séquence

Lorsque un objet reçoit un message, il exécute une action. Cette action est décrite par une *méthode*. Une méthode est le processus conduit par l'objet appelé lorsqu'il traite le message. Par exemple, si un objet tuteur envoie un message à un objet étudiant pour lui demander son nom, alors l'objet étudiant répond simplement à l'appelant par l'envoi d'une partie de son état, à savoir, le nom. Cependant, si un objet tuteur envoie le message demandant l'age de l'objet étudiant, alors la méthode que doit réaliser l'objet appelé est plus élaborée. Tout d'abord il doit obtenir la date actuelle. Ceci peut être récupéré par l'envoi d'un message à un objet calendrier. Puis l'objet étudiant doit réaliser une opération arithmétique pour déduire son age à partir de sa date de naissance et de la date du jour. Dans le diagramme de séquence ci-dessus ce processus est illustré par une *activation*, le rectangle adjacent à la flèche du message.

L'exemple de l'age illustre la *propagation de messages* et donc l'imbrication de méthodes. Lorsque un objet reçoit un message cela génère souvent l'envoi en cascade d'autres messages à d'autres objets. L'objet tuteur envoi un message à l'objet étudiant pour lui demander son age. A son tour, l'objet étudiant envoi un

message à l'objet calendrier pour connaitre la date du jour. Cet exemple montre qu'un système OO est un mélange d'interactions entre objets pour atteindre un objectif commun.

Une université aura surement un grand nombre d'étudiants. Contrairement aux étudiants réels, les objets étudiant vont tous présenter le même comportement et ils vont tous porter la même information. Nous pouvons caractériser un objet étudiant par un nom, une date de naissance et un numéro de matricule. Les valeurs des états de deux étudiants seront différentes puisque un matricule est unique. Dans un grande université nous pouvons cependant s'attendre à avoir deux ou plusieurs étudiants avec le même nom et la même date de naissance.

En revanche, ils seront tous soumis au même comportement. Si à un étudiant peut être demandé son age par un envoi de message approprié, alors ce même message peut être envoyé à tous les étudiants. Comment ? Tous les objets étudiants soutiennent une abstraction unique que nous choisirons d'appeler `Student`. D'autres abstractions pour ce même domaine d'application seront par exemple `ProgrammeOfStudy` et `Tutor`. Cet abstraction est appelé la *classe* d'un objet.

Une classe est donc un gabarit ou un modèle qui décrit complètement l'abstraction. La classe décrit les informations que contient un objet pour représenter son état. Les items d'information sont des *attributs* (ou *propriétés*). La classe définit aussi les comportements des objets en listant les opérations qu'ils peuvent effectuer (i.e. les messages qu'ils peuvent recevoir), et les effets de ces opérations décrits par un méthode pour chaque opération. La figure 2 illustre un exemple simple de diagramme de classe pour la classe `Student`.

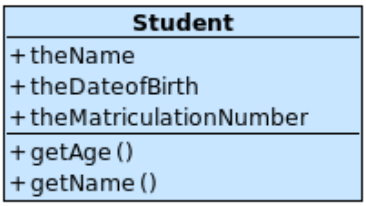

FIGURE 2 – Diagramme de classe UML pour la classe `Student`

Dans cette figure nous avons une classe `Student` avec deux opérations et trois attributs. Tout objet étudiant qui sera créé aura un état qui comprend trois valeurs pour ces attributs. De plus, à tout objet étudiant peut être envoyé des messages pour obtenir son nom ou son age.

La figure 3 illustre graphiquement comment est représentée une *instance* d'une classe en UML. La région supérieure indique le nom de la classe dont est originaire l'instance et labellise l'objet avec un identifier unique (s1). La région inférieure indique les valeurs d'attributs de l'instance qui représentent son état. Un tel élément de diagramme fait généralement partie d'un diagramme à objet (e.g. diagramme de collaboration).

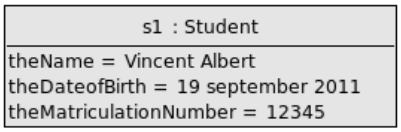

FIGURE 3 – Une instance de `Student`

Considérons maintenant la modélisation d'un compte en banque. Un compte peut avoir une panoplie de comportement tel que débiter ou créditer une somme du compte, obtenir le solde du compte. Ces comportements donnent lieu à des opérations du même nom. Une transaction de débit ou de crédit indiquera la quantité impliquée dans la transaction et modifiera le solde. Le solde d'un numéro de compte particulier doit être géré par chaque instance de compte. Chaque exemple de compte en banque porte ses propres valeurs pour ces deux attributs.

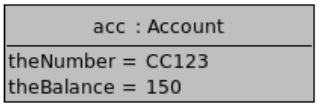

FIGURE 4 – Une instance de `Account`

Pour modéliser un compte par un objet, nous décrivons ses comportements par des opérations et ses caractéristiques par des attributs. Durant l'exécution du système, l'objet devra réaliser ses différentes opérations et changer au besoin les valeurs de ses attributs pour prendre en compte l'effet des actions. Par exemple, dans la figure 4 une opération `debit` appliquée à cet objet compte engendrera un changement de la valeur de l'attribut `theBalance`.

Certaines opérations donnent une information sur l'objet, alors que d'autres ont un effet sur l'objet. Par exemple, une transaction de débit sur un objet compte modifie l'attribut `theBalance` de l'objet. L'opération permettant de retourner l'état du solde du compte est une opération qui n'a pas d'effet sur l'état de l'objet. Collectivement les valeurs des attributs de l'objet définissent son *état*. Les opérations qui modifient la valeur d'un ou plusieurs attributs changent donc l'état de l'objet.

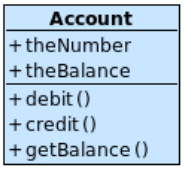

FIGURE 5 – La classe `Account`

4.2 Objets comme entités modulaires

4.3 Interactions entre objet : appels de messages

Les objets interagissent entre eux par appels de messages d'un objet appelant (client) à un objet appelé (serveur) afin que ce dernier réalise un certain nombre d'opérations. Par exemple, si un client de banque souhaite faire une transaction de crédit sur son compte, l'objet appelant client envoi un message à l'objet appelé compte. Un *message* est composé de l'identificateur de l'objet appelé et du nom de l'opération souhaitée. Ce nom de message représente une des opérations publiques de la classe dont est originaire l'objet appelé. Si l'opération nécessite des détails supplémentaires, ceux-ci sont donnés en *paramètres*.

Par exemple, pour demander à un objet compte d'engager une transaction de débit d'une somme donnée, l'objet appelé, par exemple, l'objet distributeur doit envoyer le message : `acc debit 50`

Ici, *acc* est l'identité de l'objet appelé, *debit* est l'opération requise et 50 est le paramètre informant l'objet appelée de la somme à imputer. Le diagramme de collaboration UML sur la figure 6 illustre cet appel de message. Dans ce diagramme il y a deux objets : l'objet `Account` identifié `acc` et un objet sans label originaire de la classe `ATM`. Ce dernier envoie un message à l'objet `Account` pour que celui-ci execute l'opération `debit` avec le paramètre 50.

A la réception d'un message, l'objet appelé réalise l'action correspondante. L'objet utilisera alors certaines valeurs des attributs qui représentent l'état de l'objet et les paramètres du message pour réaliser cette action. La logique associée à cette action est décrite par la `méthode`. Une méthode est un algorithme qui est déroulé lorsque l'opération est exécutée.

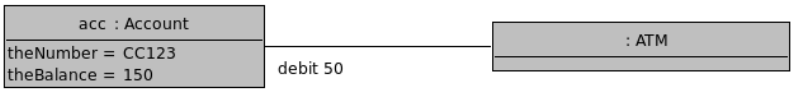

FIGURE 6 – Message à `acc` dans un diagramme de collaboration

Un message est un moyen de communication entre un client et un serveur. Parfois le client peut attendre du serveur une réponse en retour de son message sous la forme d'une *valeur de retour*.

4.4 Les classes : ensembles d'objets similaires

Il est commun d'avoir plus d'un objet de même sorte. Par exemple une banque a certainement plusieurs compte client qui chacun réalise les même actions et porte les même sortes d'information. La classe `Account` qui représente un ensemble de comptes définit des opérations (methodes) et des attributs. Les comptes qui sont représentés par les instances de la classe ont un identificateur unique (acc1, acc2,...). Chaque instance a un état particulier défini par les valeurs de ces attributs.

La figure 7 illustre la classe `Account` et deux instances de cette classe *acc1* et *acc2*. La classe `Account` définit trois opérations `debit`, `credit` et `getBalance`. Chaque instance de cette classe a ses propres valeurs des attributs `theNumber` et `theBalance`.

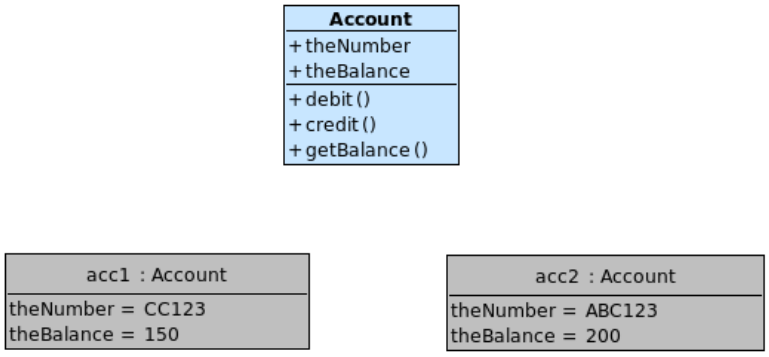

FIGURE 7 – La classe `Account` et deux instances

Les objets constituent un ensemble, deux objets sont reliés entre eux par des relations lorsqu'il doivent échanger des messages. La figure 8 illustre deux objets `Account` qui sont liés à l'objet `Bank`. Les objets `Account` ne sont pas reliés entre eux. Donc l'objet `Bank` peut envoyer des message à l'un des deux ou aux deux objets `Account` et ces derniers peuvent envoyer des messages à l'objet `Bank`. Puisque il n'y a pas de relation entre les deux objets `Account`, ils ne peuvent pas s'envoyer de messages.

Les classes et la relation correspondants sont modélisés sur la figure 9. L'annotation * indique qu'un objet `Bank` peut être lié à 0 ou plusieurs objets `Account`.

4.5 Spécialisation

Les modèles UML peuvent être conçus intelligemment afin de réduire la complexité en organisant les classes par des niveaux de hiérarchie du plus général ou plus spécifique.

La généralisation décrit une relation entre une classe générale (classe de base ou classe parent) et une classe spécialisée (sous-classe). La classe spécialisée est intégralement cohérente avec la classe de base, mais

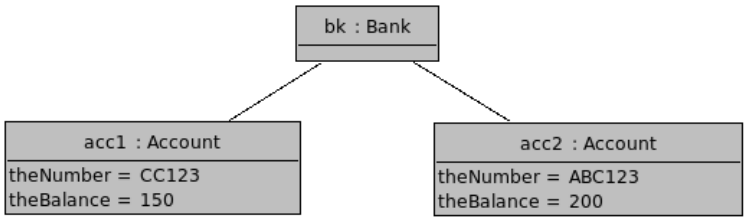

FIGURE 8 – Objets et relations

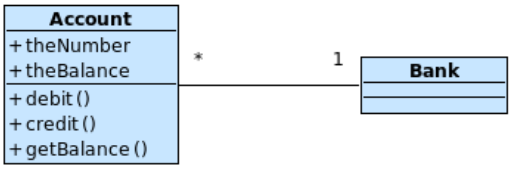

FIGURE 9 – Classes et relations

comporte des informations supplémentaires (attributs, opérations, associations). Un objet de la classe spécialisée peut être utilisé partout où un objet de la classe de base est autorisé.

Dans le langage UML, ainsi que dans la plupart des langages objet, cette relation de généralisation se traduit par le concept d'héritage. On parle également de relation d'héritage.

Par exemple, dans l'application de la banque, la classe `Account` peut être spécialisée par deux sous-classes `CurrentAccount` et `DepositAccount`. La figure 10 illustre le concept d'héritage.

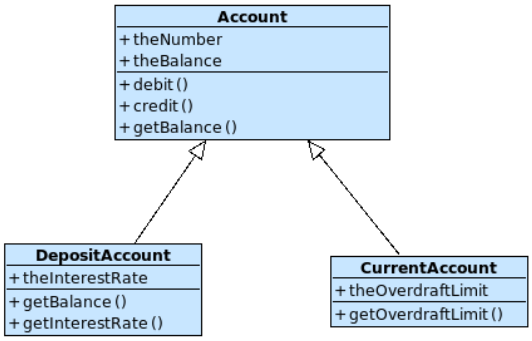

FIGURE 10 – Deux sous-classes de `Account`

On dit que l'opération `getBalance` de la classe `DepositAccount` *redéfinit* l'opération `getBalance` de la classe `Account`. La redéfinition d'une opération dans une sous-classe réalise une implémentation spécialisée de la méthode. Par exemple, l'opération `getBalance` de la classe `Account` retournera simplement la valeur de `theBalance` alors que l'opération `getBalance` de la classe `DepositAccount` indique une redéfinition qui par exemple intègre le cours du taux d'intérêt.

Les propriétés principales de l'héritage sont :
- La classe enfant possède toutes les caractéristiques des ses classes parents, mais elle ne peut accéder aux caractéristiques privées de cette dernière.
- Une classe enfant peut redéfinir (même signature) une ou plusieurs méthodes de la classe parent. Sauf indication contraire, un objet utilise les opérations les plus spécialisées dans la hiérarchie des classes.

– Toutes les associations de la classe parent s'appliquent aux classes dérivées.
– Une instance d'une classe peut être utilisée partout où une instance de sa classe parent est attendue. Par exemple, en se basant sur le diagramme de la figure 10, toute opération acceptant un objet d'une classe `Account` doit accepter un objet de la classe `DepositAccount`.
– Une classe peut avoir plusieurs parents, on parle alors d'héritage multiple. Le langage C++ est un des langages objet permettant son implémentation effective, le langage Java ne le permet pas.

4.6 Polymorphisme

D'une manière générale, le concept de polymorphisme se traduit souvent par l'expression "peut prendre plusieurs formes". Puisqu'un `DepositAccount` est aussi un `Account` avec peut être des attributs et des opérations supplémentaires, une instance de `DepositAccount` peut être utilisé alors qu'une instance de `Account` est attendue. Considérons une classe `Bank` avec un certain nombre d'objet compte qui lui sont associés. La classe `Bank` n'a pas besoin de savoir s'il s'agit d'un objet issu de la classe `CurrentAccount` ou issu de la classe `DepositAccount`.

La figure 11 représente un diagramme de classe dans laquelle une instance particulière de la classe `Bank` sera responsable de 0 ou plusieurs comptes, pouvant être des comptes courants ou des comptes de dépôt.

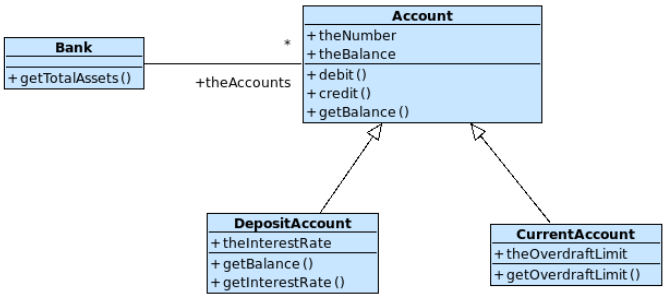

FIGURE 11 – Diagramme de classe `Bank`/`Account`

Lorsque le message `getBalance` est envoyé à chaque compte, le choix de l'opération à exécuter se fait en fonction de la classe dont est issu l'objet appelé. Si l'objet appelé est issu de la classe `DepositAccount` c'est la version redéfinie de l'opération `getBalance` qui est exécutée. Si l'objet appelé est issu de la classe `CurrentAccount` il exécute la méthode `getBalance` héritée de sa classe parent `Account`.

4.7 Association

Une association est une relation entre deux classes, qui décrit les connexions structurelles entre leurs instances. Une association entre deux classes indique donc qu'il peut y avoir co-opération par appel de message entre les instances de ces classes. En général, la relation d'association est utilisée lorsque deux objets ne sont conceptuellement pas reliés mais que, dans un contexte opérationnel, ils utilisent des opérations des uns et des autres. Par exemple `Bank` et `Account` sont associées dans le sens où un objet banque à besoin de connaitre le solde d'un objet compte.

Il existe plusieurs types d'association :
– une à une : une personne conduit une voiture.
– une à plusieurs : une banque gère plusieurs comptes.
– plusieurs à plusieurs : plusieurs professeurs enseignent à plusieurs étudiants.
– association récursive : plusieurs personnes enfants ont une personne mère.

4.8 Agrégation et composition

Une association simple entre deux classes représente une relation structurelle entre pairs, c'est à dire entre deux classes de même niveau conceptuel : aucune des deux n'est plus importante que l'autre. Lorsque l'on

souhaite modéliser une relation tout/partie où une classe constitue un élément plus grand (tout) composé d'éléments plus petit (partie), il faut utiliser l'agrégation ou la composition.

La composition est plus forte que l'agrégation. Dans une composition le tout n'existe pas sans ses parties et les parties n'ont pas de raison d'être sans leur tout. Ainsi dans une composition :
- la suppression du tout implique la suppression des parties
- il n'y a toujours qu'un seul et unique tout, i.e. les parties ne sont pas partagées avec différents tout
- un message destiné à une partie doit être envoyé au tout puis transféré par celui-ci à la partie (vrai aussi pour l'agrégation)

5 Implémentation de la technologie objet avec Java

5.1 Classes et objets

5.1.1 Définition d'une classe

Nous abordons maintenant, l'implémentation des objets avec Java. Un objet est créé selon son modèle : la classe. En java une classe est définie par

```
public class C1{
  type1 p1;          // propriété p1
  type2 p2;          // propriété p2
  ...
  type3 m3(...){ //      méthode m3
  ...
  }
  type4 m4(...){ //      méthode m4
  ...
  }
  ...
}
```

A partir de la classe C1, on peut créer de nombreux objets O1, O2,... Tous auront les propriétés p1, p2,... et les méthodes m3, m4,... Ils auront des valeurs différentes pour leurs propriétés pi ayant ainsi chacun un état qui leur est propre.

Reprenons l'exemple du compte en banque et considérons la définition de la classe `Account` figure 12.

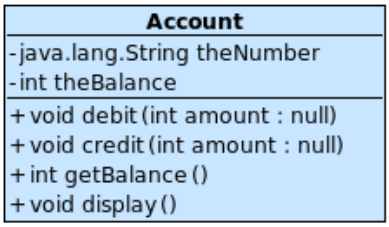

FIGURE 12 – Classe `Account` et ses attributs

Jusqu'à maintenant nous n'avons pas considéré le *niveau de visibilité* des attributs et des opérations d'une classe. Un attribut et une opération peut avoir les niveaux de visibilité suivants :
- privé (-) Un champ privé (*private*) n'est accessible que par les seules méthodes internes de la classe
- public (+) Un champ *public* est accessible par toute fonction définie ou non au sein de la classe
- protégé (#) Un champ protégé (*protected*) n'est accessible que par les seules méthodes internes de la classe ou d'un objet dérivé

En général, les données d'une classe sont déclarées privées alors que ses méthodes sont déclarées publiques. Cela signifie que le programmeur n'aura pas accès directement aux données privées de l'objet et il pourra faire appel aux méthodes publiques de l'objet et notamment à celles qui donneront accès à ses données privées.

A présent, nous considérons aussi la *signature des opérations*. La signature ajoute des paramètres à l'opération et définit le type de valeurs, s'il existe, retourné par l'opération. Les opérations `credit` et `debit` ont chacune un paramètre de type `int` pour indiquer le montant de la transaction. Elles ne retournent aucune valeur. L'opération `getBalance` est ce qu'on appelle une *méthode de lecture*. Elle ne change pas l'état de l'objet. Elle est utilisée pour lire l'attribut privé `theBalance` de la classe `Account`. Elle retourne donc une valeur de retour du type de l'attribut que l'on souhaite lire, ici, un entier `int`.

La définition de la classe Java `Account` sera la suivante :

```
public class Account{

  // attributs
  private String theNumber;
  private int theBalance;

  // operations
  public void credit(int amount){
    theBalance += ammount;
  }

  public void debit(int amount){
    if(theBalance >= amount)
      theBalance -= ammount;
  }

  public int getBalance(){
    return theBalance;
  }

  public void display(){
    System.out.println("Account number " + theNumber + " balance " + theBalance);
  }
}
```

5.1.2 Constructeurs d'une classe

Un constructeur est une méthode qui porte le nom de la classe et qui est appelée lors de la création de l'objet. On s'en sert généralement pour l'initialiser. C'est une méthode qui peut accepter des arguments mais qui ne rend aucun résultat. Son prototype ou sa définition ne sont précédés d'aucun type (même pas void).

Pour éviter des problèmes, la création d'un objet doit aussi initialiser ses attributs. Un constructeur doit donc permettre de passer en paramètres les valeurs d'initialisation des attributs de l'objet. Il est aussi nécessaire de donner un constructeur par défaut qui sera appelé si aucun paramètres ne sont donnés par l'utilisateur à la création de l'objet. La définition de la classe Java `Account` devient :

```
public class Account{

  // attributs
  private String theNumber;
  private int theBalance;

  // constructeur paramétré
  public Account(String aNumber, int aBalance){
```

```
    theNumber = aNumber;
    theBalance = aBalance;
  }

  // constructeur par défaut
  public Account(){
    this("",0);
  }

  // operations
  public void credit(int amount){
    theBalance += ammount;
  }

  public void debit(int amount){
    if(theBalance >= amount)
      theBalance -= ammount;
  }

  public int getBalance(){
    return theBalance;
  }

  public void display(){
    System.out.println("Account number " + theNumber + " balance " + theBalance);
  }
}
```

Pour créer un objet on utilise l'opérateur `new`. Les instructions suivantes indiquent que `ac1` référence un objet compte dont le nom est `ABC123` et le solde `1000` et que `ac2` référence un objet compte dont le nom est vide est le solde 0. La première instruction utilise le constructeur paramétré alors que la deuxième instruction utilise le constructeur par défaut.

```
Account ac1 = new Account("ABC123", 1000);
Account ac2 = new Account();
```

L'instruction

```
Account ac3;
```

déclare `ac3` comme une référence à un objet de type `Account`. Cet objet n'existe pas encore et donc `ac3` n'est pas initialisé. C'est comme si on écrivait :

```
Account ac3 = null;
```

5.2 Création de notre première application

Afin de construire une application nous avons besoin d'un objet qui est capable de répondre au message envoyé par l'environnement d'exécution (le système d'opération). Pour des raisons historiques cet objet doit contenir une méthode `main`.

```
//Main.java
public class Main{
  public static void main(String[ ] args){
  Application app = new Application();
  app.run();
  }
```

```
}

Application.java
public class Application{
  public void run(){
    Account acc = new Account("ABC123",1200);
    acc.credit(200); \\solde est maintenant 1400
    acc.display();
    acc.debit(900);  \\solde est maintenant 500
    acc.debit(700);  \\solde inchangé
  }
}
```

Le diagramme de séquence sur la figure 13 illustre ces suites d'instructions.

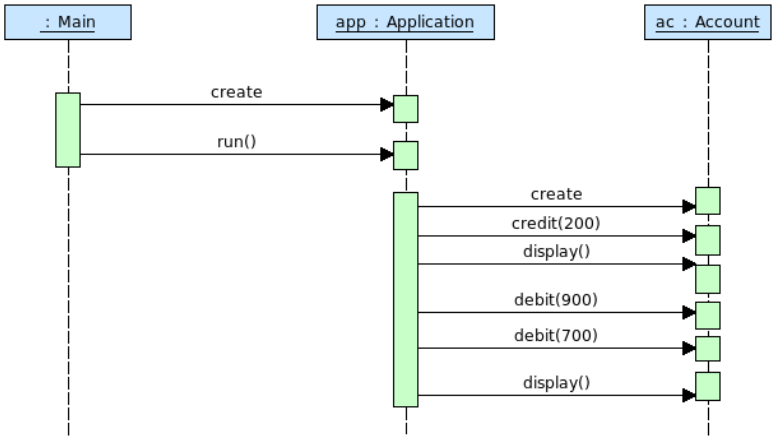

FIGURE 13 – Diagramme de séquence pour Application

5.3 Implémentation des relations architecturales

5.3.1 Collections d'objets

En Java, il existe différentes sortes de collections d'objets. Deux d'entre elles sont les *listes* et les *ensembles*. L'ensemble est similaire à celui que l'on trouve en mathématique. C'est un groupe d'éléments uniques non ordonnés. Une liste est une séquence d'éléments indexés qui autorise les doubles.

Le tableau ci-dessous liste quatre des différentes classes de collection de Java :

Conteneur	Classe Java	Descrption
set	HashSet	Non ordonnée et pas de double, insertions/suppressions rapide
set	TreeSet	Ordonnée et pas de double, insertions/suppressions rapide
list	ArrayList	Ordonnée, indexée et double, pas pratique pour insertions/suppressions
list	LinkedList	Ordonnée, indexée et double, pratique pour insertions/suppressions

5.3.2 Création d'une collection d'objets

Pour créer une collection d'objets en Java, l'instruction est la même que pour n'importe quel objet par exemple, l'instruction

```
ArrayList accounts = new ArrayList();
```

crée un objet `ArrayList` dont la taille est fixée par défaut, alors que l'instruction

```
ArrayList accounts = new ArrayList(16);
```

crée un objet `ArrayList` de taille 16.

5.3.3 Ajout/Suppression d'objets dans une collection

Toutes les collections Java proposent une opération `add` permettant d'ajouter un objet à la collection. Puisque une collection `HashSet` n'est pas ordonnée il suffit d'appeler l'opération

```
boolean add(Object element)
```

Un élément de la collection doit découler d'une classe `Object`. Puisque toutes les classes Java descendent de la classe `Object` n'importe quel objet Java peut être ajouté à une collection. L'opération `add` retourne une valeur booléenne indiquant le succès ou l'échec de l'opération.

Considérons les déclarations suivantes :

```
HashSet accounts = new HashSet();
Account ac1 = new Account("ABC123",1200);
```

nous pouvons avoir :

```
if(accounts.add(ac1) == true)
  //actions si add ok
else
  //actions si add nok
```

ou simplement :

```
accounts.add(ac1);   //ignore la valeur boolean de retour - possible
```

ou encore en utilisant une référence d'objet anonyme :

```
accounts.add(new Account("ABC123",1200));
```

Avec une `LinkedList` les éléments peuvent être ajoutés à un endroit donné de la liste. Donc il existe diverses surcharges de l'opération `add` disponibles :

```
boolean add(Object element)   // Ajoute un élément à la fin de la liste

void add(int index, Object element)  //Insertion de l'élément à la position donnée par index

void addFirst(Object element)   //Insertion de l'élément en début de liste

void addLast(Object element)   //Insertion de l'élément en fin de liste
```

En considérant la déclaration suivante :

```
LinkedList accounts = new Linkedlist();
```

voici des exemples d'utilisation :

```
accounts.add(ac1);  //ignore la valeur boolean de retour - possible
accounts.add(new Accounts("ABC123",1200));  //objet anonyme
accounts.add(3,ac2);  //ajoute le compte ac2 à la position 4.
accounts.addFirst(ac3);
```

De la meme manière les collections offrent des opérations permettant de retirer des éléments :

```
boolean remove(Object element) // supprime l'élément element de la liste
```

5.3.4 Accéder à un objet dans une collection

Il est possible d'accéder à un objet contenu dans une `LinkedList` grâce aux opérations suivantes :

```
Object get(int index);
Object getFirst();
Object getLast();
```

dont voici des exemples d'utilisation :

```
Account ac1 = (Account)accounts.get(3);
Account ac2 = (Account)accounts.getLast();
Account ac3 = (Account)accounts.getFirst();
```

Chaque opération retourne un objet de type `Object` il faut donc les transtyper en `Account`.

Dans un `HashSet` les éléments ne sont pas ordonnées, les opérations ci-dessus ne sont donc pas disponibles. En fait il n'y a pas d'opération permettant d'avoir un accès direct à un élément. Nous devons utiliser un objet spécial appelé un *itérateur*. L'utilisation d'un objet `Iterator` est à la fois puissante, efficace et élégante.

Toutes les collections permettent l'opération :

```
Iterator iterator()  //retourne un itérateur à la collection qui reçoit ce message
```

où un itérateur est un objet capable de parcourir la collection de début à la fin. Il permet les opérations :

```
boolean hasNext()
Object next()
```

dont voici l'utilisation, si nous considérons les instructions suivantes :

```
HashSet accounts = new HashSet();

accounts.add(new Account("ABC123", 1200));
accounts.add(new Account("DEF456", 800));

Iterator firstIter = accounts.iterator();

while(firstIter.hasNext()){
   Account acc = (Account)firstIter.next();
   String number = acc.getNumber();
   System.out.println(number);
}

Iterator secondIter = accounts.iterator();

while(secondIter.hasNext()){
   Account acc = (Account)secondIter.next();
   String number = acc.getNumber();
   if(number.equals("ABC123") == true){
      acc.display();
      break;
   }
}
```

5.3.5 Notre première architecture

Reprenons l'exemple de la figure 11 où une banque est associée à un ensemble de comptes. Pour cela il est nécessaire d'implémenter une collection permettant de gérer cette relation. Nous utiliserons une collection de type `ArrayList`.

```
public class Account{
   ...
   public Account(String aNumber, int aBalance){
     theNumber = aNumber;
     theBalance = aBalance;
     theBank = null;
   }
   public void setBank(Bank aBank){
     theBank = aBank;
   }
   //RELATIONS
   private Bank theBank;
}

public class Bank{
   ...
   public void openAccount(String aNumber, int aBalance){
      Account acc = new Account(aNumber, aBalance);
      acc.setBank(this);
      theAccounts.add(acc);
   }
   //RELATIONS
   private java.util.ArrayList theAccounts;
}
```

5.3.6 Notre deuxième application

```
public class Application{

   public void run(){
      Bank bk = new Bank("The Best Bank");

      bk.openAccount("ABC123",1000);
      bk.openAccount("DEF456",1500);
      bk.openAccount("GHI789",2000);

      bk.creditAccount("ABC123",200);
      bk.creditAccount("ABC123",900);
      bk.creditAccount("ABC123",700);

      System.out.println("Solde:" + bk.getAccountBalance("ABC123"));
      System.out.println("Montant total:" + bk.getTotalAssets());
   }
}

public class Bank{
   //OPERATIONS
   public Bank(String aName){
      theName = name;
      theAccounts = new ArrayList();
```

```
}

public void openAccount(String aNumber, int aBalance){
   Account acc = new Account(aNumber, aalance);
   acc.setBank(this);
   theAccounts.add(acc);
}

public void creditAccount(String aNumber, int anAmount){
   Iterator iter = theAccounts.iterator();
   while(iter.hasNext()){
      Account acc = (Account)iter.next();
      if(aNumber.equals(acc.getNumber())){
         acc.credit(amount);
         break;
      }
   }
}

public void debitAccount(String aNumber, int anAmount){
   Iterator iter = theAccounts.iterator();
   while(iter.hasNext()){
      Account acc = (Account)iter.next();
      if(number.equals(acc.getNumber())){
         acc.debit(amount);
         break;
      }
   }
}

public int getAccountBalance(String aNumber){
   Iterator iter = theAccounts.iterator();
   while(iter.hasNext()){
      Account acc = (Account)iter.next();
      if(number.equals(acc.getNumber())){
         return acc.getBalance();
      }
   }
}

public int getTotalAssets(){
   int totalAssets = 0;
   Iterator iter = theAccounts.iterator();
   while(iter.hasNext()){
      Account acc = (Account)iter.next();
      totalAssets += acc.getBalance();
      }
   }
}

//ATTRIBUTES
String theName;

//RELATIONS
private java.util.ArrayList theAccounts;
```

5.4 Spécialisation

5.4.1 Déclaration et utilisation de l'héritage

Nous montrons ici comment la notion d'héritage est implémentée en Java. Supposons qu'on veuille créer une classe employé : un employé est une personne particulière. Il a des attributs qu'une autre personne n'aura pas : un salaire par exemple. Mais il a aussi les attributs de toute personne : prénom, nom et âge. Un employé fait donc pleinement partie de la classe personne mais a des attributs supplémentaires. Plutôt que d'écrire une classe employé en partant de rien, on préférerait reprendre l'acquis de la classe personne qu'on adapterait au caractère particulier des employés. C'est le concept d'héritage qui nous permet cela.

La figure 14 illustre un diagramme de classe qui représente une entreprise qui embauche plusieurs employés qui sont aussi des personnes.

FIGURE 14 – Relation de spécialisation

Pour exprimer que la classe `Employee` hérite des propriétés de la classe `Person`, on écrira :

```
public class Employee extends Person
```

`Person` est la classe parent et `Employee` est la classe spécialisée. L'objet employé a toutes les qualités d'un objet personne : il a les mêmes attributs et les mêmes méthodes. Ces attributs et ces méthodes hérités de la classe parent ne sont pas répétés dans la définition de la classe : on se contente d'indiquer les attributs et méthodes rajoutés par la classe spécialisée :

```
public class Person{
   //OPERATIONS
   public void setName(String aName){...}
   public void display(){...}
   public String getName(){...}
   public int getAge(){...}
   public Person(String aName, GregorianCalendar aDateofBirth){...}
   public Person(){...}
   //ATTRIBUTES
   private GregorianCalendar theDateOfBirth;
   private String theName;
}

public class Employee extends Person{
   //OPERATIONS
   public void fired(){...}
   public void hiredBy(Company aCompany){...}
   public int getSalary(){...}
   public String getCompanyName(){...}
   public Employee(String aName, GregorianCalendar aDateofBirth,
    int aSalary, int aRefNumber){...}
```

```
   public Employee(){...}
   //ATTRIBUTES
   private int theRefNumber;
   private int theSalary;
   //RELATIONS
   private Company theEmployer;
}
```

En java, toutes les classes sont dérivées de la classe `Object`. La classe `Object` est la classe de base de toutes les autres. C'est la seule classe de Java qui ne possède pas de classe parent. Tous les objets en Java, quelle que soit leur classe, découlent d'`Object`. Cela implique que tous les objets possèdent déjà à leur naissance un certain nombre d'attributs et de méthodes dérivés d'`Object`. Dans la déclaration d'une classe, si la clause `extends` n'est pas présente, la superclasse immédiatement supérieure est donc `Object`.

Typiquement nous pourrons utiliser ces classes de la manière suivante :

```
GregorianCalendar d1 = new GregorianCalendar(1976,0,15);
GregorianCalendar d2 = new GregorianCalendar(1980,11,22);

String str1 = new String("Vincent");
String str2 = new String("Jean");

Person person = new Person(str1,d1);
Employee employee = new Employee(str2,d2, 1500, 1234);

System.out.println(person.getAge()); // affiche 35
System.out.println(employee.getAge()); // affiche 30

System.out.println(employee.getSalary()); // affiche 1500
```

L'instruction suivante est impossible :

```
 person.getSalary()
```

Puisque la classe `Employee` hérite de la classe `Person` il convient d'utiliser le constructeur de cette dernière pour créer une instance de la classe `Employee`. Si le constructeur de la classe `Person` est :

```
public Person(String aName, java.util.GregorianCalendar aDateOfBirth){
  theName = aName;
  theDateOfBirth = aDateOfBirth;
}
```

alors le constructeur de la classe `Employee` est :

```
public Employee(String aName, java.util.GregorianCalendar aDateOfBirth,
int aSalary, int aRefNumber){
  super(aName,aDateOfBirth);
  theSalary = aSalary;
  theRefNumber = aRefNumber;
}
```

L'instruction `super(aName,aDateOfBirth)` est un appel au constructeur de la classe parent, ici la classe personne. On sait que ce constructeur initialise les champs nom et date de naissance de l'objet personne contenu à l'intérieur de l'objet étudiant. Notons que l'appel au constructeur de la classe parent ne crée pas d'objet personne.

Les sous-classes n'ont pas accès aux membres privés de leur classe parent. Si un attribut de la classe parent est déclaré `private`, cet attribut n'est pas accessible par les sous-classes. Seuls des objets de la même classe ont un accès direct à ces champs. Tous les autres objets, y compris des objets fils comme ici, doivent passer par des méthodes publiques pour y avoir accès. Ainsi il est impossible d'écrire :

```
public Employee(String aName, java.util.GregorianCalendar aDateOfBirth,
int aSalary, int aRefNumber){
  theName = aName;
  theDateOfBirth = a DateOfBirth;
  theSalary = aSalary;
  theRefNumber = aRefNumber;
}
```

Si on veut qu'un attribut ou une méthode soit accessible par les sous-classes, il faut le déclarer comme `protected`. Cependant, utiliser le constructeur de la classe parent est la méthode usuelle : lors de la construction d'un objet fils, on appelle d'abord le constructeur de l'objet parent puis on complète les initialisations propres cette fois à l'objet fils.

Tout constructeur d'une classe spécialisée appelle forcément l'un des constructeurs de la classe parent : si cet appel n'est pas explicite, l'appel du constructeur par défaut (sans paramètre) est effectué implicitement.

5.4.2 Redéfinition d'une méthode

Considérons que la méthode `display` de la classe `Person` permette d'afficher sur la console le nom et l'age d'une personne :

```
//class Person
public void display(){
  System.out.println("Nom:" + theName);
  System.out.println("Age:" + this.getAge());
}
```

Si l'on souhaite afficher sur la console les informations d'un employé (salaire et numéro de référence) en plus des informations qu'il a hérité de personne il faut redéfinir la méthode `display` dans la classe `Employee` :

```
//class Employee
public void display(){
  super.display();
  System.out.println("Salaire:" + theSalary);
  System.out.println("Numéro de référence:" + theRefNumber);
}
```

Notons que la méthode redéfinie doit avoir une signature identique à l'originale. Sinon il s'agit d'une surcharge de méthode. Le mot clé `this` désigne l'objet courant : l'objet receveur du message.

La méthode `display` de la classe `Employee` s'appuie sur la méthode `display` de sa classe parent (`super.display()`) pour afficher sa partie "personne" puis complète avec les champs qui lui sont propre.

En utilisant les déclarations précédentes si nous exécutons les instructions suivantes :

```
person.display();
employee.display();
```

il s'affichera sur la console :

```
Nom: Vincent
Age: 35

Nom:Jean
Age:31
Salaire:1500
Numéro de référence: 1234
```

5.4.3 Polymorphisme

Prenons l'exemple de la figure 15 pour illustrer l'utilisation du polymorphisme.

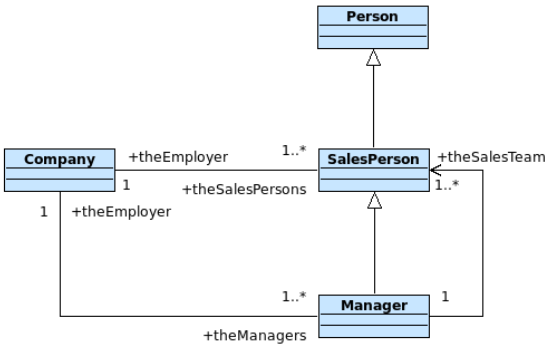

FIGURE 15 – Diagramme de classe avec spécialisations

Si l'on souhaite afficher sur la console les informations de tout les employés (commerciaux et managers) il faut parcourir les collections `theSalesPersons` et `theManagers` et pour chaque objet appartenant à ces collections il faut appeler la méthode `display` :

```
Iterator iter1 = theSalesPersons.iterator();
while(iter1.hasNext() == true){
  SalesPerson salesPerson = (SalesPerson)iter1.next();
  salesPerson.display();
}

Iterator iter2 = theManagers.iterator();
while(iter2.hasNext() == true){
  Manager manager = (Manager)iter2.next();
  manager.display();
}
```

Notons que `next()` retourne un `Object` que nous transtypons en SalesPerson.

L'utilisation du polymorphisme donne la solution illustrée par le diagramme de classe de la figure 16.

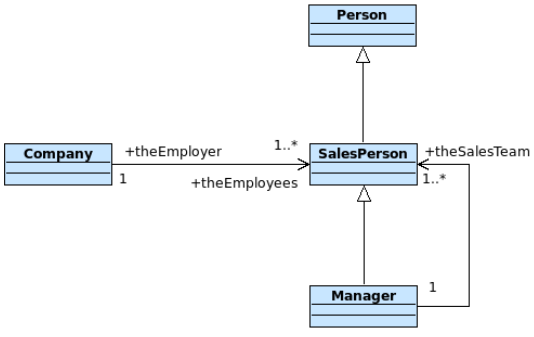

FIGURE 16 – Diagramme de classe amélioré

Une méthode de `Company` peut maintenant avoir les instructions suivantes :

```
Iterator iter = theEmployees.iterator();
while(iter.hasNext() == true){
  SalesPerson employee = (SalesPerson)iter.next();
  employee.display();
}
```

La méthode `Display` dans `Person` est dite polymorphique. La méthode `Display` dans `SalesPerson` est dite redéfinie. Imaginons maintenant que nous ne souhaitions pas qu'une opération de la classe `Person` aie un comportement polymorphe. Par exemple si nous voulons que la méthode `getName` retourne simplement le nom d'une personne, d'un commercial ou d'un manager. Nous définirons alors cette méthode comme *figée*, le mot clé correspondant est `final` :

```
public final void String getName(){...}
```

Une méthode final est une méthode qui ne peut pas être redéfinie dans les sous-classes. On peut également appliqué ce mot clé aux classes. Une classe final est une classe qui ne peut pas avoir de fils. Une classe final ne peut pas être étendue : le mécanisme d'héritage est bloqué. Mais une classe final peut évidemment être le fils d'une autre classe non bloquée.

Rappelons nous :
- une opération figée ne peut pas être redéfinie dans une classe fils
- une opération polymorphe peut être redéfinie dans une classe fils directe
- une opération redéfinie peut être redéfinie dans une classe fils directe
- une opération non figée peut être figée dans une classe fils.

Le polymorphisme permet de substituer un objet parent par un objet fils. Le choix de la méthode à exécuter (pour une méthode polymorphe) ne se fait pas statiquement à la compilation mais dynamiquement à l'exécution. A l'invocation d'une méthode, le choix de l'implémentation à exécuter ne se fait pas en fonction du type déclaré de la référence à l'objet, mais en fonction du type réel de l'objet. `employee` désigne parfois un objet qui découle de `Manager`, même si c'est une référence qui découle de `SalesManager` : donc c'est la méthode implémentée dans `Manager` qui est exécutée.

Complétons maintenant le code Java pour l'application donnée en exemple.

```
public class Person{
   //OPERATIONS
   public Person(String aName, GregorianCalendar aDateofBirth){...}
   public Person(){...}
   public void display(){
      System.out.println("Nom:" + theName + "\t" + "Age:" + this.getAge());
   }
   public final String getName(){...}
   public final int getAge(){...}

   //ATTRIBUTES
   private GregorianCalendar theDateOfBirth;
   private String theName;
}

public class SalesPerson extends Person{
   //OPERATIONS
   public SalesPerson(String aName, GregorianCalendar aDateofBirth,
    int aSalary, int aRefNumber){...}
   public SalesPerson(){...}
   public void display(){
      super.diplay();
      System.out.println("Salary:" + theSalary + "\t" + "Numero de référence:" +
```

```
          theReferenceNumber + "\t" + "Figure de vente:" + theSalesFigure);
   }

   public final void setSalesFigure(int aSalesFigure){...}
   public final int getSalesFigure(){...}

   //ATTRIBUTES
   private int theRefNumber;
   private int theSalary;
   private int theSalesFigure;
}

public class Manager extends SalesPerson{
   //OPERATIONS
   public Manager(String aName, GregorianCalendar aDateofBirth,
    int aSalary, int aRefNumber){...}
   public Manager(){...}
   public void display(){
      super.diplay();
      System.out.println("Ventes accumulées:" + this.getAccumulatedSales());
   }

   public final void addSalesPerson(SalesPerson aSalesPerson){...}
   public final int getAccumulatedSales(){...}

   //ATTRIBUTES
   private int theAccumulatedSales;

   //RELATIONS
   private java.util.HashSet theSalesTeam;

public final class Company{
   //OPERATION
   public Company(String aName){...}
   public Company(){...}
   public final String getName(){...}
   public final void displayEmployees(){
      System.out.println("Nom de l'entreprise:" + theName);
      Iterator iter = theEmployees.iterator
      while(iter.hasNext() == true){
        SalesPerson salesPerson = (SalesPerson)iter.next();
        salesPerson.display();
      }
   }

   public final void addEmployee(SalesPerson aSalesPerson, Manager aManager){
      if(aManager!=null)
         aManager.addSalesPerson(aSalesPerson);
      theEmployees.add(aSalesPerson);
   }

   //ATTRIBUTES
   private String theName;

   //RELATIONS
```

```
   private java.util.HashSet theEmployees;
}

//Application class
public void final run(){
   //crée quelques objets
   Company c1 = new Company("the Best Company");
   SalesPerson s1 = new SalesPerson("Vincent", new GregorianCalandar(1976,0,15),1500,1234);
   s1.setSalesFigure(1000);
   SalesPerson s2 = new SalesPerson("Jean", new GregorianCalandar(1980,11,22),2000,5678);
   s2.setSalesFigure(1500);
   Manager m1 = new Manager("Chris", new GregorianCalandar(1982,3,7),3000,9123);
   m1.setSalesFigure(2500);

   //configuration
   c1.addEmployee(m1,null);
   c1.addEmployee(s1,m1);
   c1.addEmployee(s2,m1);

   //demonstration du polymorphisme
   c1.displayEmployees();
}
```

donne la sortie :

```
Nom de l'entreprise: theBest Company
Nom:Chris Age:29
Salaire:3000 Numero de référence: 9123 Figure de Vente 2500
Ventes accumulées: 5000
Nom:Jean Age:30
Salaire:2000 Numero de référence: 5678 Figure de Vente 1500
Nom:Vincent Age:35
Salaire:1500 Numero de référence: 1234 Figure de Vente 1000
```

5.5 Classes abstraites

Dans certain cas, vous voudrez définir une superclasse qui déclare la structure d'une abstraction particulière sans fournir de mise en oeuvre complète de chaque méthode. En d'autre termes vous souhaiterez créer une superclasse qui définisse uniquement une forme générale partagée par toutes ses sous-classes, en laissant à ces dernières le soin de spécifier les détails. Une telle classe appelée *classe abstraite* détermine uniquement la nature des méthodes que doivent mettre en oeuvre les sous-classes. On ne peut pas l'instancier.

Par exemple considérons une classe `Employee` qui est une spécialisation de `Person`. Considérons qu'il n'y aura jamais d'instance d'`Employee` puisqu'un objet `Employee` sera toujours soit un `Manager` soit un `SalesPerson` et jamais qu'un `Employee`. Cependant nous souhaitons que tous les employés de l'entreprise partagent des opérations communes comme `display`, `getSalesFigure`, `setSalesFigure`, `getBonus` et `getSalary`. Alors nous choisissons de stéréotyper la class `Employee` comme abstraite.

La classe abstraite `Employee` se déclare de la manière suivante :

```
public abstract class Employee extends Person
```

Ainsi, si nous considérons que la classe `SalesPerson` est une classe dérivée de `Employee` nous pouvons avoir :

```
Employee theEmployee = new SalesPerson(...);
```

mais jamais

```
Employee theEmployee = new Employee(...);
```

Ci-dessous le code Java pour la classe `Employee` :

```
public abstract class Employee extends Person{
//OPERATIONS
public abstract double getBonus();
public final int getSalesFigure(){...}
public final void setSalesFigure(int aSalesFigure){...}
public void display(){...}
public Employee(String aName,java.util.GregorianCalendar aDateOfBirth,
int aSalary, int aReferenceNumber(){...}
public Employee(){...}
//ATTRIBUTES
private int theSalesFigure;
private int theReferenceNumber;
private int theSalary;
}
```

De la même manière qu'une classe abstraite ne peut pas être instanciée on peut aussi définir une opération abstraite qui n'a pas de méthode.

L'instruction

```
Employee theEmployee = new SalesPerson(...);
```

est valide seulement si `SalesPerson` est une classe dérivée de `Employee`.

La figure 17 illustre la nouvelle version du diagramme de classe précédent prenant en compte cette nouvelle spécification.

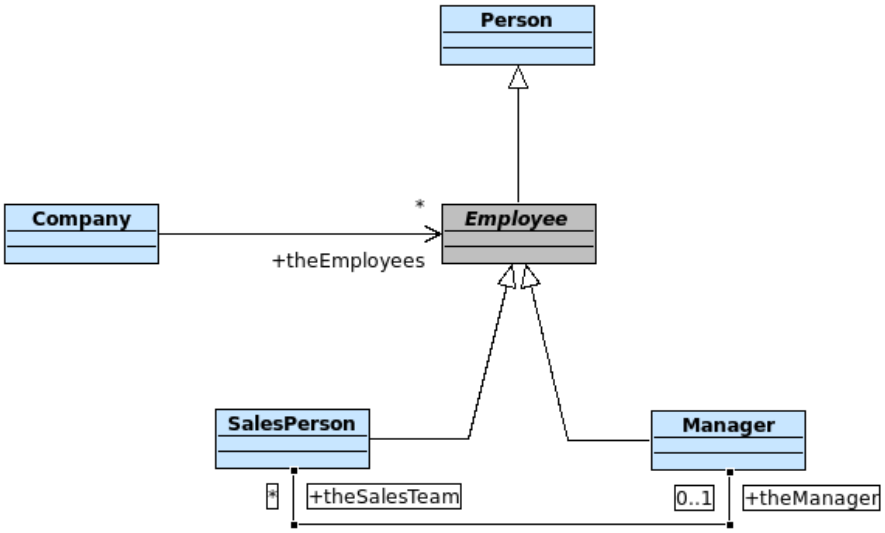

FIGURE 17 – Diagramme de classe avec classe abstraite

Les classes dérivées `SalesPerson` et `Manager` définissent une méthode pour l'opération redéfinie `getBonus` :

```
//SalesPerson class
public final double getBonus(){
return this.getSalesFigure() * 0.1;
}

//Manager class
public final double getBonus(){
return this.getAccumulatedSales() * 0.1;
}
```

Une méthode `run` valide pour scénariser ce nouveau modèle est :

```
//Application class
public void final run(){
   //crée quelques objets.. comme avant

   //les configurer
   c1.addEmployee(m1);
   c1.addEmployee(s1,m1);
   c1.addEmployee(s2,m1);

   //demonstration du polymorphisme
   c1.displayEmployees();

   //Determine le bonus de chacun
   System.out.println("Bonus pour " + s1.getName() + ":" + s1.getBonus() + "euros");
   System.out.println("Bonus pour " + s2.getName() + ":" + s2.getBonus() + "euros");
   System.out.println("Bonus pour " + m1.getName() + ":" + m1.getBonus() + "euros");
}
```

donne la sortie :

```
//Comme avant
Bonus pour Chris: 500
Bonus pour Jean: 150
Bonus pour Vincent: 100
```

Notez que la méthode `addEmployee` de la classe `Company` devient :

```
public final void addEmployee(Employee anEmployee){
   theEmployees.add(anEmployee);
}

public final void addEmployee(SalesPerson aSalesPerson, Manager aManager){
   aManager.addSalesPerson(aSalesPerson);
   aSalesPerson.setManager(aManager);
   this.addEmployee(aSalesPerson);
}
```

5.6 Interfaces

Une interface est un ensemble de prototypes de méthodes ou de propriétés qui forme un contrat. Une classe qui décide d'implémenter une interface s'engage à fournir une implémentation de toutes les méthodes définies dans l'interface. C'est le compilateur qui vérifie cette implémentation.

Considérons notre application. Nous pourrions insister sur le fait que tous les employés de l'entreprise doivent implémenter les opérations `setReferenceNumber` et `display`. En d'autres termes, la classe de laquelle

est originaire un employé doit avoir au moins ces opérations dans son interface publique. Cependant, il n'y a aucune contrainte sur la hiérarchie de spécialisation de la classe correspondante. Une classe qui implémente une interface peut dérivée de n'importe quelle classe. Deux classes peuvent implémenter la même interface mais ne pas appartenir à la même hiérarchie de spécialisation.

Nous pouvons modéliser cette situation avec une *interface* Java tel que l'illustre la figure 18.

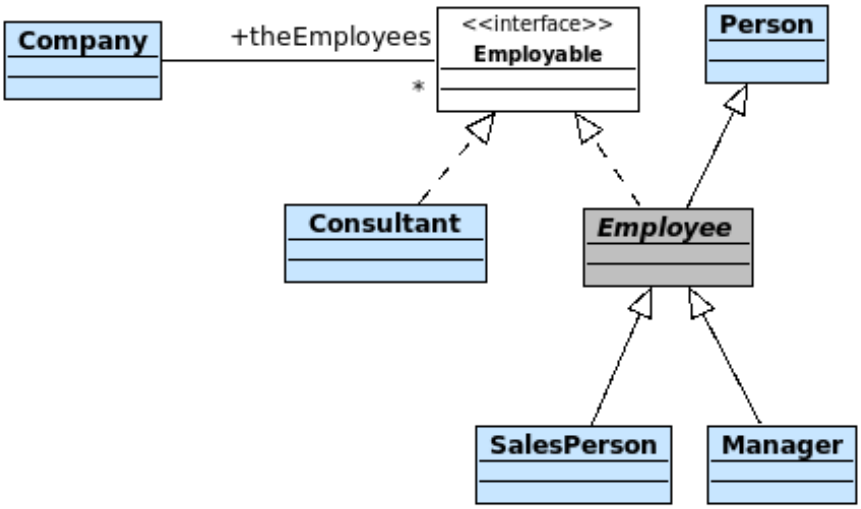

FIGURE 18 – Diagramme de classe avec une interface

Maintenant une `Company` est associée à des objets `Employable` qui peuvent être issus de n'importe quelle hiérarchie de classes. Le code Java pour l'interface `Employable` est :

```
public interface Employable{
  public abstract void setReferenceNumber(int aReferenceNumber);
  public abstract void display();
}
```

Le code pour `Employee`, `SalesPerson` et `Manager` devient :

```
public abstract class Employee extends Person implements Employable{
  public final void setReferenceNumber(int aReferenceNumber){...}
  public void display(){...}
  //...
  private int theReferenceNumber;
  private int theSalary;
}

public final class SalesPerson extends Employee{
  public void display(){...}
  //...
}

public final class Manager extends Employee{
  public void display(){...}
  //...
}
```

Pour la classe `Consultant` nous avons :

```
public final class Consultant implements Employable{
  public final void setReferenceNumber(int aReferenceNumber){...}
  public void display(){...}
  \\...
  private String theName;
```

Et pour la classe `Company` nous avons :

```
public final class Company{
   //OPERATION
   public final void setAllReferenceNumbers(){
     int referenceNumber = 1;

     Iterator iter = theEmployees.iterator
     while(iter.hasNext() == true){
       Employable employee = (Employable)iter.next();
       employee.setReferenceNumber(referenceNumber);
       referenceNumber++;
     }
   }

   public final void displayEmployees(){
      System.out.println("Nom de l'entreprise:" + theName);
      Iterator iter = theEmployees.iterator
      while(iter.hasNext() == true){
        Employable employee = (Employable)iter.next();
        employee.display();
      }
   }
   //...
}
```

Partie 3 : La bibliothèque Java

Nous présentons dans ce chapitre un certain nombre de classes Java d'usage courant. Celles-ci ont de nombreux attributs, méthodes et constructeurs. A chaque fois, nous ne présentons qu'une faible partie des classes. Le détail de celles-ci est disponible dans l'API de Java que nous verrons en TP.

6 Gestion des chaines

6.1 Constructeurs String

La classe String admet plusieurs constructeurs. Pour créer un objet `String` vide, lancez le constructeur par défaut. Par exemple :

```
String = new String();
```

La plupart du temps, vous aurez besoin de créer une chaîne ayant une valeur initiale. Pour créer une chaîne initialisée par un tableau de caractères, utilisez le constructeur suivant :

```
String (caractères char [ ])
```

exemple :

```
char cars[ ] = {'a','b','c'};
String s = new String(cars);
```

Vous pouvez spécifier une plage dans tableau de caractères pour initialiser l'objet `String` utilisant le constructeur :

```
String (caractères char [], int startIndex, int numChars)
```

exemple :

```
caractères char [] = {'a', 'b', 'c', 'd', 'e', 'f'};
String s = new String (char, 2, 3);
```

Ce constructeur initialise `s` avec la chaine "cde".

6.2 Méthodes de String

La classe String est riche d'attributs et méthodes. En voici quelques-uns :

Methode	Exemple	Description
public char charAt(int i)	String("cheval").charAt(3)	donne le caractère i de la chaîne
public int compareTo(chaine2)	chaine1.compareTo(chaine2)	compare chaine1 à chaine2 et rend 0 si chaine1 = chaine2, 1 si chaine1 > chaine2, -1 si chaine1 < chaine2
public boolean equals(Object anObject)	chaine1.equals(chaine2)	rend vrai si chaine1=chaine2, faux sinon
int length()		nombre de caractères de la chaîne
public String substring(int beginIndex, int endIndex)	String("chapeau").subString(2,4)	rend la chaîne "ape"
public char[] toCharArray()		permet de mettre les caractères de la chaîne dans un tableau de caractères
int indexOf(String chaine2)		rend la première position de chaine2 dans la chaîne courante ou -1 si chaine2 n'est pas présente
static String valueOf(float f)		Rend le nombre réel f sous forme de chaine

7 Exploration de java.lang

7.1 Encapsulation des types simples

Java utilise des types simples, tels que int et char, pour des raisons de performance. Ces types de données ne font pas partie de la hiérarchie des objets. Ils sont passés aux méthodes par valeur et ne peuvent pas être fournis directement par référence. De ce fait, il n'existe aucun moyen pour deux méthodes distinctes de faire référence à la même instance d'un int. Pour enregistrer un type simple dans une de ces classes, vous devrez l'encapsuler dans une classe. Les classes suivantes encapsulent les types simples à l'intérieur d'une classe : Integer, Boolean, Double, Float, Short, Byte, Long, Character.

8 Exploration de java.util

Je vous invite à découvrir la librairie java.util de la documentation Java. Cette librairie contient les classes permettant de gérer les collections. Nous en avons déjà vu certaines d'entre elles.

9 Entrées/sorties : exploration de java.io

9.1 Écriture sur écran

La syntaxe de l'instruction d'écriture sur l'écran est la suivante :

```
System.out.println(expression) ou System.err.println(expression)
```

où `expression` est tout type de donnée qui puisse être converti en chaîne de caractères pour être affiché à l'écran. `System.out` écrit dans un fichier texte qui est par défaut l'écran. Il en est de même pour `System.err`. Ces fichiers portent un numéro (ou descripteur) respectivement 1 et 2. Le flux d'entrée du clavier (System.in) est également considéré comme un fichier texte, de descripteur 0. Dos comme Unix supportent le tubage (pipe) de commandes :

```
commande1 | commande2
```

Tout ce que `commande1` écrit avec `System.out` est tubé (redirigé) vers l'entrée `System.in` de `commande2`. Dit autrement, `commande2` lit avec `System.in`, les données produites par `commande2` avec `System.out` qui ne sont donc plus affichées à l'écran. Ce système est très utilisé sous Unix. Dans ce tubage, le flux `System.err` n'est lui pas redirigé : il écrit sur l'écran. C'est pourquoi il est utilisé pour écrire les messages d'erreurs (d'où son nom err) : on est assuré que lors d'un tubage de commandes, les messages d'erreur continueront à s'afficher à l'écran. On prendra donc l'habitude d'écrire les messages d'erreur à l'écran avec le flux `System.err` plutôt qu'avec le flux `System.out`.

9.2 Lecture de données tapées au clavier

Le flux de données provenant du clavier est désigné par l'objet `System.in` de type `InputStream`. Ce type d'objets permet de lire des données caractère par caractère. C'est au programmeur de retrouver ensuite dans ce flux de caractères les informations qui l'intéressent. Le type `InputStream` ne permet pas de lire d'un seul coup une ligne de texte. Le type `BufferedReader` le permet avec la méthode `readLine`.

Afin de pouvoir lire des lignes de texte tapées au clavier, on crée à partir du flux d'entrée `System.in` de type `InputStream`, un autre flux d'entrée de type `BufferedReader` cette fois :

```
BufferedReader IN=new BufferedReader(new InputStreamReader(System.in));
```

Nous n'expliquerons pas ici les détails de cette instruction qui fait intervenir la notion de constructions d'objets. Nous l'utiliserons telle-quelle. La construction d'un flux peut échouer : une erreur fatale, appelée exception en Java, est alors générée. A chaque fois qu'une méthode est susceptible de générer une exception, le compilateur Java exige qu'elle soit gérée par le programmeur. Aussi, pour créer le flux d'entrée précédent, il faudra en réalité écrire :

```
BufferedReader IN = null;
try{
  IN=new BufferedReader(new InputStreamReader(System.in));
} catch (Exception e){
    System.err.println("Erreur " +e);
    System.exit(1);
}
```

De nouveau, on ne cherchera pas à expliquer ici la gestion des exceptions. Une fois le flux `IN` précédent construit, on peut lire une ligne de texte par l'instruction :

```
String ligne;
ligne=IN.readLine();
```

La ligne tapée au clavier est rangée dans la variable `ligne` et peut ensuite être exploitée par le programme.

9.3 Les fichiers texte

9.3.1 Écrire

Pour écrire dans un fichier, il faut disposer d'un flux d'écriture. On peut utiliser pour cela la classe `FileWriter`. Les constructeurs souvent utilisés sont les suivants :

`FileWriter(String fileName)`

crée le fichier de nom `fileName` - on peut ensuite écrire dedans - un éventuel fichier de même nom est écrasé

`FileWriter(String fileName,boolean append)`

idem - un éventuel fichier de même nom peut être utilisé en l'ouvrant en mode ajout (append = true).

La classe `FileWriter` offre un certain nombre de méthodes pour écrire dans un fichier, méthodes héritées de la classe `Writer`. Pour écrire dans un fichier texte, il est préférable d'utiliser la classe `PrintWriter` dont les constructeurs souvent utilisés sont les suivants :

`PrintWriter(Writer out)`

l'argument est de type `Writer`, c.a.d. un flux d'écriture (dans un fichier, sur le réseau, ...)

`PrintWriter(Writer out, boolean autoflush)`

idem. Le second argument gère la bufferisation des lignes. Lorsqu'il est à faux (son défaut), les lignes écrites sur le fichier transitent par un buffer en mémoire. Lorsque celui-ci est plein, il est écrit dans le fichier. Cela améliore les accès disque. Ceci-dit quelquefois, ce comportement est indésirable, notamment lorsqu'on écrit sur le réseau.

Les méthodes utiles de la classe `PrintWriter` sont les suivantes :

void print(Type T)	écrit la donnée T (String, int, ...)
void println(Type T)	idem en terminant par une marque de fin de ligne
void flush()	vide le buffer si on n'est pas en mode autoflush
void close()	ferme le flux d'écriture

Voici un programme qui écrit quelques lignes dans un fichier texte :

```
import java.io.*;
public class ecrire{
  public static void main(String[] arg){
    // ouverture du fichier
    PrintWriter fic=null;
    try{
      fic=new PrintWriter(new FileWriter("out"));
    } catch (Exception e){
      Erreur(e,1);
    }
    // écriture dans le fichier
    try{
      fic.println("Jean,Dupont,27");
      fic.println("Pauline,Garcia,24");
      fic.println("Gilles,Dumond,56");
    } catch (Exception e){
```

```
      Erreur(e,3);
    }
    // fermeture du fichier
    try{
      fic.close();
    } catch (Exception e){
      Erreur(e,2);
    }
  }// fin main
  private static void Erreur(Exception e, int code){
    System.err.println("Erreur : "+e);
    System.exit(code);
  }//Erreur
}//classe
```

9.3.2 Lire

Pour lire le contenu d'un fichier, il faut disposer d'un flux de lecture associé au fichier. On peut utiliser pour cela la classe `FileReader` et le constructeur suivant :

`FileReader(String nomFichier)`

ouvre un flux de lecture à partir du fichier indiqué. Lance une exception si l'opération échoue.

La classe `FileReader` possède un certain nombre de méthodes pour lire dans un fichier, méthodes héritées de la classe `Reader`. Pour lire des lignes de texte dans un fichier texte, il est préférable d'utiliser la classe `BufferedReader` avec le constructeur suivant :

`BufferedReader(Reader in)`

ouvre un flux de lecture bufferisé à partir d'un flux d'entrée in. Ce flux de type `Reader` peut provenir du clavier, d'un fichier, du réseau,...

Les méthodes utiles de la classe `BufferedReader` sont les suivantes :

int read()	lit un caractère
String readLine()	lit une ligne de texte
int read(char[] buffer, int offset, int taille)	lit taille caractères dans le fichier et les met dans le tableau buffer à partir de la position offset
void close()	ferme le flux de lecture

Voici un programme qui lit le contenu du fichier créé précédemment :

```
// classes importées
import java.util.*;
import java.io.*;
public class lire{
  public static void main(String[] arg){
    personne p=null;
    // ouverture du fichier
    BufferedReader IN=null;
    try{
      IN=new BufferedReader(new FileReader("out"));
    } catch (Exception e){
      Erreur(e,1);
```

```
    }
    // données
    String ligne=null;
    String[] champs=null;
    String prenom=null;
    String nom=null;
    int age=0;
    // gestion des éventuelles erreurs
    try{
      while((ligne=IN.readLine())!=null){
        champs=ligne.split(",");
        prenom=champs[0];
        nom=champs[1];
        age=Integer.parseInt(champs[2]);
        System.out.println(""+new personne(prenom,nom,age));
      }// fin while
    } catch (Exception e){
      Erreur(e,2);
    }
    // fermeture fichier
    try{
      IN.close();
    } catch (Exception e){
      Erreur(e,3);
    }
  }// fin main
  // Erreur
  public static void Erreur(Exception e, int code){
    System.err.println("Erreur : "+e);
    System.exit(code);
  }
}// fin classe
```

10 Gestion des exceptions

De nombreuses fonctions Java sont susceptibles de générer des exceptions, c'est à dire des erreurs. Nous avons déjà rencontré une telle fonction, la fonction `readLine`.

```
String ligne=null;
try{
try
      ligne=IN.readLine();
      System.out.println("ligne="+ligne);
    } catch (Exception e){
      affiche(e);
      System.exit(2);
}// try
```

Lorsqu'une fonction est susceptible de générer une exception, le compilateur Java oblige le programmeur à gérer celle-ci dans le but d'obtenir des programmes plus résistants aux erreurs : il faut toujours éviter le "plantage" sauvage d'une application. Ici, la fonction `readLine` génère une exception s'il n'y a rien à lire parce que par exemple le flux d'entrée a été fermé. La gestion d'une exception se fait selon le schéma suivant :

```
try{
  appel de la fonction susceptible de générer l'exception
} catch (Exception e){
```

```
  traiter l'exception e
}
instruction suivante
```

Si la fonction ne génère pas d'exception, on passe alors à `instruction suivante`, sinon on passe dans le corps de la clause `catch` puis à `instruction suivante`. Notons les points suivants :

- `e` est un objet dérivé du type `Exception`. On peut être plus précis en utilisant des types tels que `IOException`, `SecurityException`, `ArithmeticException`, etc... : il existe une vingtaine de types d'exceptions. En écrivant `catch (Exception e)`, on indique qu'on veut gérer toutes les types d'exceptions. Si le code de la clause `try` est susceptible de générer plusieurs types d'exceptions, on peut vouloir être plus précis en gérant l'exception avec plusieurs clauses `catch` :
  ```
  try{
    appel de la fonction susceptible de générer l'exception
  } catch (IOException e){
    traiter l'exception e
  }
  } catch (ArrayIndexOutOfBoundsException e){
    traiter l'exception e
  }
  } catch (RunTimeException e){
    traiter l'exception e
  }
  instruction suivante
  ```
- On peut ajouter aux clauses `try/catch`, une clause `finally` :
  ```
  try{
    appel de la fonction susceptible de générer l'exception
  } catch (Exception e){
    traiter l'exception e
  }
  finally{
    code exécuté après try ou catch
  }
  instruction suivante
  ```
  Ici, qu'il y ait exception ou pas, le code de la clause `finally` sera toujours exécuté.
- La classe `Exception` a une méthode `getMessage()` qui rend un message détaillant l'erreur qui s'est produite. Ainsi si on veut afficher celui-ci, on écrira :
  ```
  catch (Exception ex){
     System.err.println("L'erreur suivante s'est produite : "+ex.getMessage());
     ...
  }//catch
  ```
- La classe `Exception` a une méthode `toString()` qui rend une chaîne de caractères indiquant le type de l'exception ainsi que la valeur de la propriété `Message`. On pourra ainsi écrire :
  ```
  catch (Exception ex){
     System.err.println ("L'erreur suivante s'est produite : "+ex.toString());
     ...
  }//catch
  ```
  On peut écrire aussi :
  ```
  catch (Exception ex){
     System.err.println ("L'erreur suivante s'est produite : "+ex);
     ...
  }//catch
  ```
  Nous avons ici une opération string + Exception qui va être automatiquement transformée en string + Exception.toString() par le compilateur afin de faire la concaténation de deux chaînes de caractères.

Partie 4 : Interfaces graphiques

11 Vue d'ensemble de Swing

Les classes de la librairie Swing reprennent les composants les plus familiers d'une interface graphique comme les boutons, les menus, les champs de text... Ce cours a pour objectif de vous apprendre à spécialiser ces classes de composants, pour les besoins de votre application, et à y associer des objets *gestionnaires*, permettant de définir les actions à réaliser lorsque un événement survient. Nous ne voyons pas toutes les classes de la librairie Swing. Nous voyons les essentielles, il y a beaucoup de similitudes entre toutes ces classes.

Beaucoup de classes sont des spécialisations de la classe `JComponent`. Cette classe abstraite porte la plupart des comportements communs aux composants graphiques. Par exemple cette classe définit si un composant est visible ou non. Il implémente aussi la notion d'agrégation. Cette classe est donc la racine d'une hiérarchie de spécialisation qui s'étend à divers type de composants concrets qui peuvent être utilisés dans une application graphique. Par exemple, une fenêtre de dialogue peut contenir un champ textuel dans lequel l'utilisateur entre une valeur. La classe `JTextField` représente un champ textuel. A coté de ca, on pourra trouver un label permettant de documenter ce que doit contenir le champ textuel. Un label est obtenu en utilisant la classe `JLabel` qui est une classe immédiatement dérivée de `JComponent` (voir figure 19).

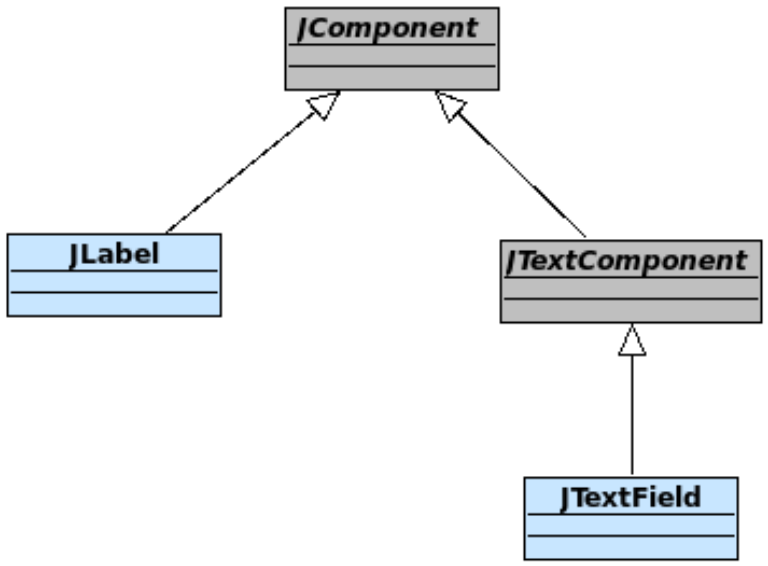

FIGURE 19 – Hierarchie des classes `JLabel` et `JTextField`

La plupart des composants graphiques sont développés de cette manière, par spécialisation directe ou indirecte. Le tableau ci-dessous donne la description des composants les plus utilisés.

JButton	Boutton poussoir pouvant être décoré par du texte ou un icone graphique.
JFrame	Fenêtre de base avec des boutons de fermeture, agrandissement/réduction, une taille ajustable...
JLabel	Zone d'affichage pour du texte
JMenuBar	Menu-bar au haut de la fenêtre
JPanel	Conteneur générique utiliser en général pour grouper des composants entre eux
JTextArea	Zone de texte multi-ligne
JTextField	Composant permettant l'édition d'une ligne de texte

12 Première application graphique

Une application graphique dérive en général de la classe de base `JFrame` (fenêtre de base). Le constructeur paramétré de cette classe prend une chaine comme argument qui est utilisée comme titre de l'application, inscrite dans la barre de légende. Une fois créée, la fenêtre doit être rendu visible grâce à la méthode `setVisible`. La méthode `setBounds` permet de définir la position et la taille de la fenêtre. La version ci-dessous ouvre la fenêtre en haut à gauche de l'écran :

```
import javax.swing.*;

public class Main{
  public static void main(String[ ] args){
    JFrame frame = new JFrame("Company");
    frame.setBounds(0,0,400,300);
    frame.setVisible(true);
  }
}
```

Dans cette deuxième version, nous avons implémenté l'application graphique dans une classe dédiée `CompanyFrame`. Cette classe hérite de `JFrame`. De plus, la fenêtre de l'application, à son démarrage, est positionnée au centre de l'écran. La classe `ToolKit` a une méthode permettant de retourner la taille de l'écran. La valeur `Dimension` retournée par cette méthode encapsule la largeur et la hauteur du composant. Nous pouvons accéder aux valeurs largeur et hauteur soit directement ou en utilisant les méthodes `getWidth` et `getHeight`.

```
// class Main
import companysubsystem.CompanyFrame;

public class Main{
  public static void main(String[ ] args){
    CompanyFrame frame = new CompanyFrame("Company");
  }
}

//class CompanyFrame
package companysubsystem;
```

```
import javax.swing.*;
import java.awt.*;

public class CompanyFrame extends JFrame{
  public CompanyFrame(String caption){
    super(caption);
    Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
    int width = screen.width * 3/4;
    int height = screen.height * 3/4;
    this.setBounds(screen.width/8,screen.height/8,width,height);
    this.setVisible(true);
  }
}
```

Pour terminer cet exemple, on notera les paquetages importés :
- javax.swing pour la classe JFrame
- java.awt pour la classe Dimension

13 Les événements

Les applications graphiques sont orientées *événement*. Elles sont en attente d'un évènement utilisateur pour réaliser une action et servir cet événement. Quand l'utilisateur bouge la souris ou selectionne un menu, un événement survient, stimule l'application, et change éventuellement l'état du système. Les évènements sont représentés par des objets de différentes classes évènement. Par exemple, les mouvements de souris sont représentés par des objets issus de la classe `MouseEvent`. La selection d'un item d'un menu est représentée par un objet issu de la classe `ActionEvent`. Un objet évènement dispose de différentes propriétés décrivant les aspects de l'évènement. Par exemple, un objet issu de `MouseEvent` possède les coordonnées de la position de la souris au moment où est survenu l'évènement. La figure 20 illustre un extrait de la hierarchie de classe des évènements.

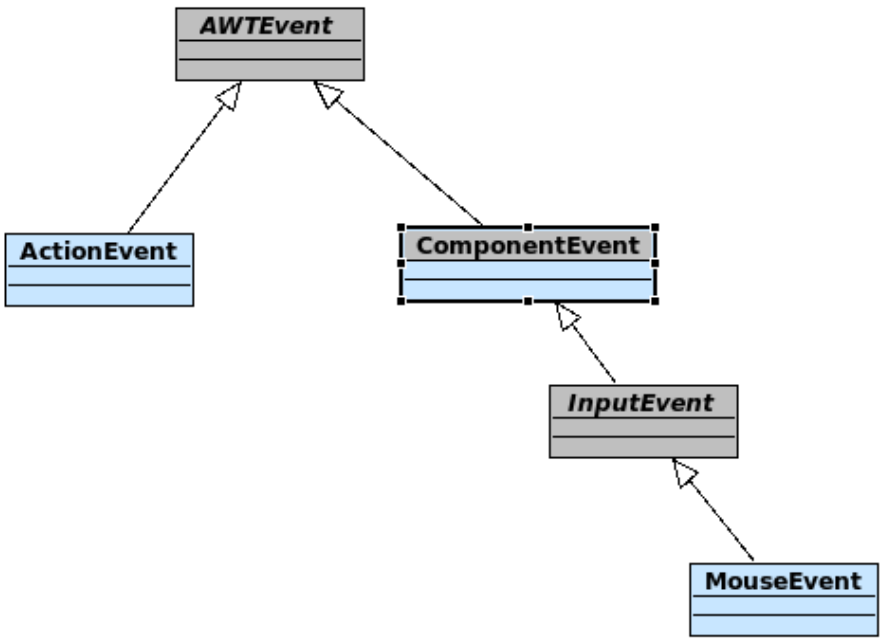

FIGURE 20 – Hierarchie de classe des évènements

Les évènements Java sont basés sur la notion de *event listener* ou gestionnaire d'évènements. Un listener est un objet qui "écoute" et détecte les événements qui se produisent sur un composant graphique. Il existe différents types de gestionnaires pour les différents événements qui peuvent se produire sur les composants d'une interface graphique. Par exemple, pour le composant `JFrame`, le listener s'appelle `WindowListener`. Lorsqu'un évènement est généré, le composant source de cet évènement informe tous ces objets listener par un appel de message en lui passant en paramètre l'objet évènement. Un listener implémente donc une méthode particulière permettant de recevoir ce message, tel que le définit l'interface qu'il implémente. Par exemple une classe `XXXListener` qui représente un listener pour un ensemble d'`ActionEvent`, doit implémenter l'interface `ActionListener` (voir figure 21). Pour implémenter cette interface, la sous-classe doit avoir une définition de la méthode `actionPerformed`.

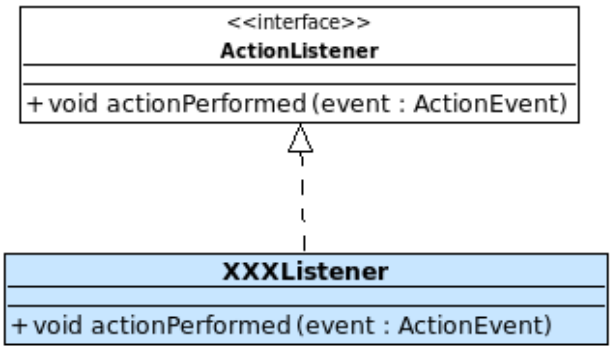

FIGURE 21 – `ActionListener`

On dira qu'un `JComponent` *enregistre* un listener. La méthode `addActionListener` est appelée *méthode d'enregistrement*. L'ensemble des listeners enregistrés à un objet source peut être dynamiquement mis à jour grâce aux méthodes ajouter et retirer listeners. Par exemple la classe `JComponent` offre les méthodes `addMouseListener(MouseListener listener)` et `removeMouseListener(MouseListener listener)`.

Le diagramme de séquence sur la figure 22 illustre comment est pris en charge une action utilisateur. Considérons que nous avons une classe `XXXListener` qui représente un listener. L'objet `cf` issu de la classe `CompanyFrame` crée une instance de la classe listener et l'enregistre en appelant la méthode `addActionListener(xl)`. Lorsque l'utilisateur fait une requête, l'objet `cf` prévient le listener `xl` de l'évènement à travers la méthode `actionPerformed`.

Pour le composant `JFrame`, le listener s'appelle `WindowListener` et est une interface définissant les méthodes suivantes :

```
void windowActivated(WindowEvent e) //La fenêtre devient la fenêtre active
void windowClosed(WindowEvent e) //La fenêtre a été fermée
void windowClosing(WindowEvent e) // L'utilisateur ou le programme a demandé la fermeture de la fenêtre
void windowDeactivated(WindowEvent e) //La fenêtre n'est plus la fenêtre active
void windowDeiconified(WindowEvent e) //L'utilisateur ou le programme a demandé la fermeture de la fenê
void windowIconified(WindowEvent e) //La fenêtre n'est plus la fenêtre active
void windowOpened(WindowEvent e) //La fenêtre passe de l'état réduit à l'état normal
```

Il y a donc sept événements qui peuvent être gérés. Les gestionnaires reçoivent tous en paramètre un objet de type `WindowEvent`. L'événement qui nous intéresse ici est la fermeture de la fenêtre, événement qui devra être traité par la méthode `windowClosing`. Notre gestionnaire d'événements implémentant l'interface `WindowListener` doit définir les sept méthodes de cette interface. Au lieu d'utiliser l'interface `WindowListener` on peut utiliser la classe `WindowAdapter`. Celle-ci implémente l'interface `WindowListener`,

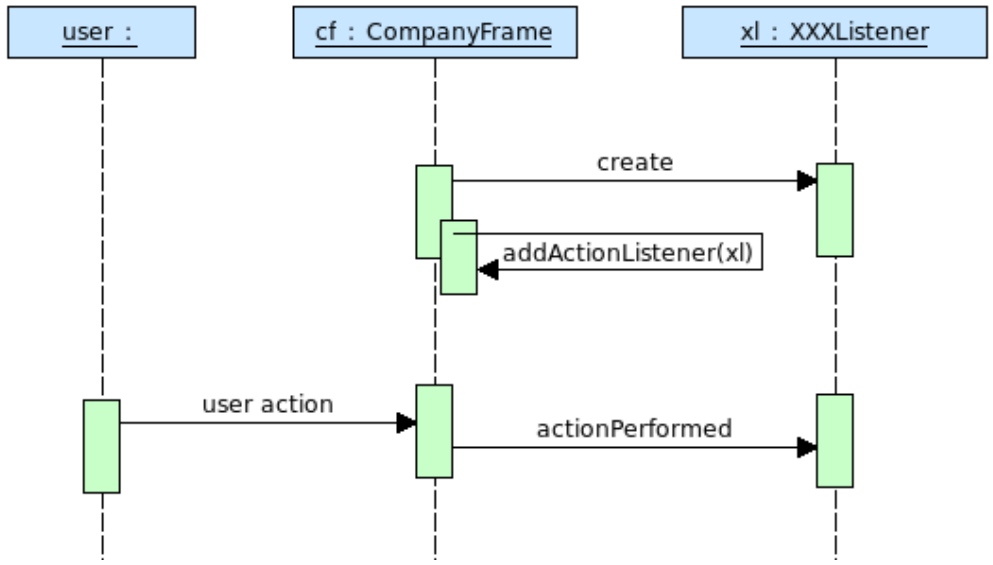

FIGURE 22 – `Création et association à un listener`

avec sept méthodes vides. En dérivant la classe `WindowAdapter` et en redéfinissant les seules méthodes qui nous intéressent, nous arrivons au même résultat qu'avec l'interface `WindowListener` mais sans avoir besoin de définir les méthodes qui ne nous intéressent pas (voir figure 23).

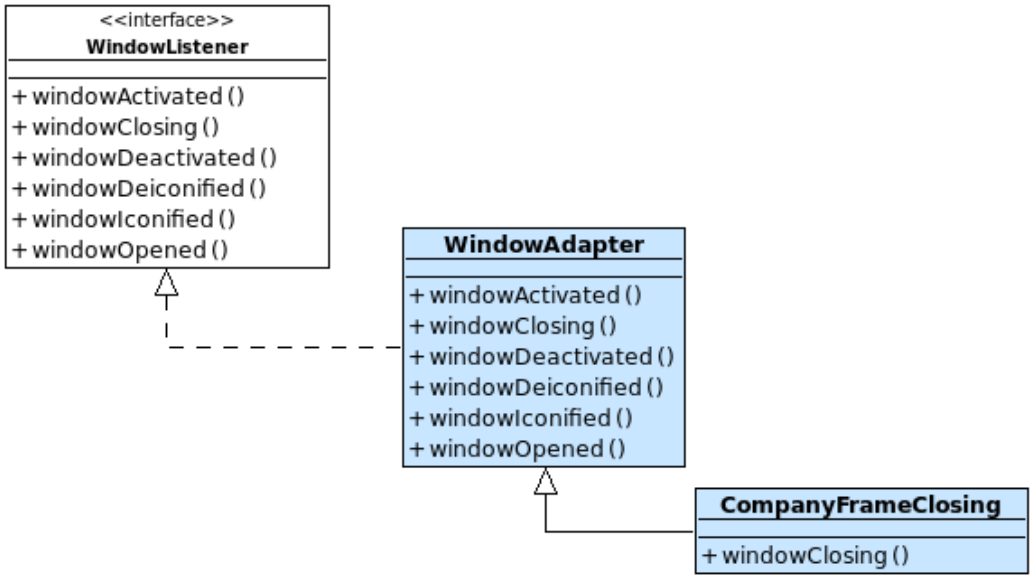

FIGURE 23 – `Notre gestionnaire d'événement`

Le code permettant de gérer la fermeture de la fenêtre de notre application `company` devient :

```
//class CompanyFrame
package companysubsystem;

import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;

public class CompanyFrame extends JFrame{
  public CompanyFrame(String caption){
    super(caption);
    Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
    int width = screen.width * 3/4;
    int height = screen.height * 3/4;
    this.setBounds(screen.width/8,screen.height/8,width,height);
    this.setVisible(true);

    this.addWindowListener(new CompanyFrameClosing());
  }

  // Classe interne

  public class CompanyFrameClosing extends WindowAdapter{
     public void windowClosing(WindowEvent event){
        System.exit(0);
     }
  }
}
```

Ici, l'objet issu de `CompanyFrame` s'envoie un message à lui-même (`this`). Ce message est `addWindowListener` et le paramètre est une instance de la classe `CompanyFrameClosing`, notre classe gestionnaire d'évènement.

14 Les barres de menu

Un menu de base est assemblé à partir des classes `JMenuBar`, `JMenu` et `JMenuItem`.

Création d'une barre de menu avec :

```
theMenuBar = new JMenuBar();
```

que l'on relie ensuite à notre fenêtre principale dérivée de `JFrame` avec :

```
this.setJMenuBar(theMenuBar);
```

Les menus et les items sont préparés comme ceci :

```
JMenu fileMenu = new JMenu("File");
fileMenu.setMnemonic('F');
theFileExitAction = ... voir après ...
JMenuItem fileExit = fileMenu.add(theFileExitAction);
fileExit.setMnemonic('x');
theMenuBar.add(fileMenu);
this.setJMenuBar(theMenuBar);
```

Pour implémenter la prise en compte d'un évènement sur un item, nous devons enregistrer un objet listener à cet item tel que nous l'avons décrit précédemment. D'abord nous devons créer un objet issu d'une classe qui implémente `ActionListener`, l'interface listener pour ce type de composant. Ensuite on enregistre cet objet avec l'item de menu en appelant la méthode d'enregistrement `addActionListener`. La classe `JMenu` offre une alternative. Plutôt que d'ajouter une instance de `JMenuItem` à un objet `JMenu`, nous pouvons ajouter une instance d'une classe qui implémente l'interface `ActionListener` en utilisant la méthode `add`. Cette opération retourne un JMenuItem nouvellement créé.

```
//class CompanyFrame
package companysubsystem;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class CompanyFrame extends JFrame{
  //OPERATIONS
  public CompanyFrame(String caption){
    super(caption);
    this.assembleMenuBar();

    Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
    int width = screen.width * 3/4;
    int height = screen.height * 3/4;
    this.setBounds(screen.width/8,screen.height/8,width,height);
    this.setVisible(true);

    this.addWindowListener(new CompanyFrameClosing());
  }

  private void assembleMenuBar(){
     JMenu fileMenu = new JMenu("File");
     fileMenu.setMnemonic('F');
     JMenuItem fileExit = fileMenu.add(theFileExitAction);
     fileExit.setMnemonic('x');
     theMenuBar.add(fileMenu);
     this.setJMenuBar(theMenuBar);
  }

  //ATTRIBUTES
  private JMenuBar theMenuBar = new JMenuBar();

  private FileExitAction theFileExitAction = new FileExitAction("Exit");

  // Classes internes

  public class FileExitAction extends AbstractAction{

     public FileExitAction(String label){
        super(label);
     }

     public void actionPerformed(ActionEvent event){
        System.exit(0);
     }
  }

  public class CompanyFrameClosing extends WindowAdapter{
     public void windowClosing(WindowEvent event){
        CompanyFrame.this.theFileExitAction.actionPerformed(null);
     }
  }
}
```

Notons ici une notion clé du concept de classe interne. Un objet issu d'une classe interne a un lien effectif avec la classes de plus haut niveau (ici `CompanyFrame`). La classe interne a accès à tout les attributs et les opérations de sa classe partenaire. Donc l'instruction :

```
CompanyFrame.this.theFileExitAction.actionPerformed(null);
```

référence l'attribut `FileExitAction` de l'instance de la classe `CompanyFrame`. Ici, `CompanyFrame.this` n'est pas nécessaire.

Les tableaux suivants donnent une liste de quelques gestionnaires d'événements et les événements auxquels ils sont liés.

Gestionnaire	Composant(s)	Méthode d'enregistrement	Évènement
ActionListener	JButton , JCheck-box,JButtonRadio, JMenu-Item,JTextField	public void addActionLis-tener(ActionListener)	clic sur le bouton, la case à cocher,le bouton radio, l'élé-ment de menu, l'utilisateur a tapé [Entrée] dans la zone de saisie
ItemListener	JComboBox, JList	public void addItemLis-tener(ItemListener)	L'élément sélectionné a changé
InputMethodListener	JTextField, JTextArea	public void ad-dMethodInputLis-tener(InputMethodListener)	le texte de la zone de saisie a changé ou le curseur de saisie a changé de position
CaretListener	JTextField, JTextArea	public void addcaretLis-tener(CaretListener)	Le curseur de saisie a changé de position
AdjustmentListener	JScrollBar	public void ad-dAdjustmentLis-tener(AdjustmentListener)	la valeur du variateur a changé
MouseMotionListener		public void ad-dMouseMotionLis-tener(MouseMotionListener)	la souris a bougé
WindowListener	JFrame	public void addWindowlis-tener(WindowListener)	événement fenêtre
MouseListener		public void addMouselis-tener(MouseListener)	événements souris (clic, en-trée/sortie du domaine d'un composant, bouton pressé, relâche)
KeyListener		public void addKeylis-tener(KeyListener)	événement clavier (touche tapée, pressée, relachée)

15 Séparation du modèle d'application et des mécanismes d'entrées/sorties

Les classes `Company`, `Employee`, `SalesPerson`, `Manager`, `Person` fournissent des méthodes permettant d'avoir accès aux états des objets. Aucune de ces classes n'implémente aucun mécanisme d'entrées/sorties. Toutes ces mécanismes doivent être fournis par les méthodes de la class `CompanyFrame`. Le patron de conception *Model-View-Controller* (MVC) préconise la séparation entre le modèle du domaine d'application et les mécanismes d'entrées/sorties de l'application. Le modèle représente l'état du système. La vue gère les visualisation des données du système et le contrôleur gère les actions de l'utilisateur sur le système. La figure 24 illustre ce patron de conception pour notre application.

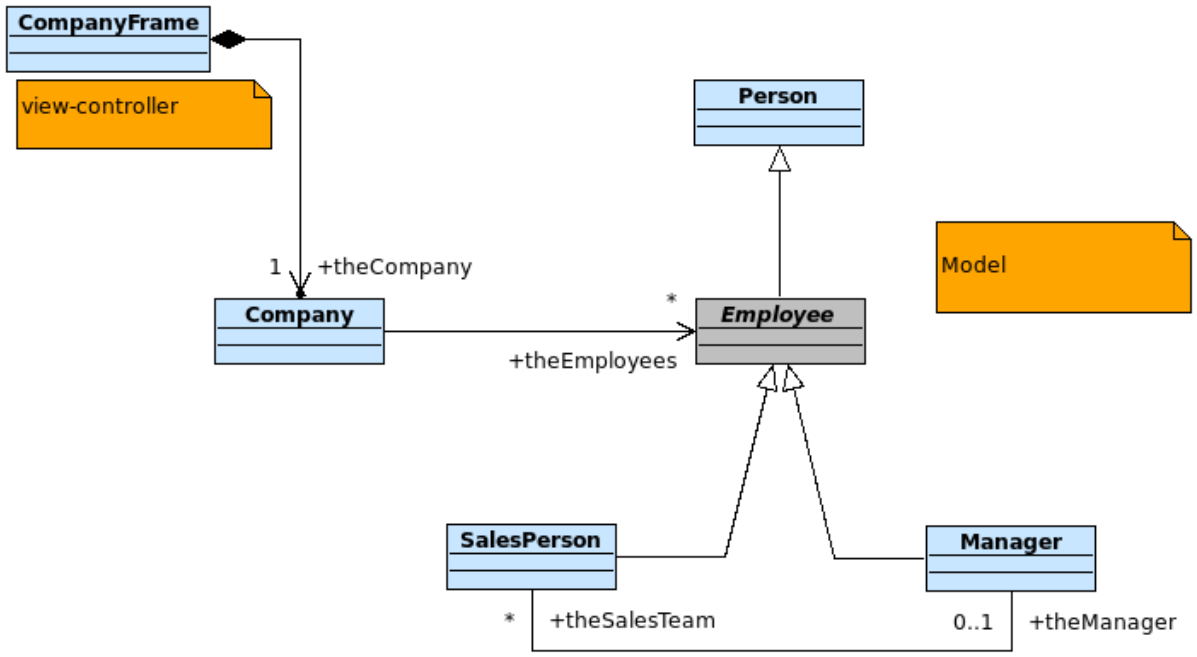

FIGURE 24 – Patron de conception MVC