Conception Orientée Objet avec Unified Modeling Language et Java

Vincent Albert

Université Paul Sabatier

2013-2014

Programme

1. Partie I : Les bases du langage Java 1h
2. Partie II : Technologie Objet et Java 2h
3. Partie III : La bibliothèque Java 1h
4. Partie IV : Interface Graphique 2h
5. TP/BE 20h

Programme

Partie I : Les bases du langage Java

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`

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.

Types de données primitifs

Les types primitifs sont définis de manière à avoir une étendue explicite et un comportement mathématique spécifique. Pour des raisons de portabilité, tous les types de données ont une étendue strictement définie.

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

Les variables

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 initialisan
                      // 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'.
```

Les variables

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);
    }
}
```

static permet à main() d'être exécuté sans avoir à instancier une instance particulière de la classe.

Portée et durée de vie d'une variable

```
class Portee{
 public static void main(String args[ ]){
  int x; //visible pour tout le code au sein du main
  x=10;
  if(x==10){ //commence une nouvelle portee
   int y = 20; //visible uniquement pour ce bloc
   System.out.println("x et y:" + x + " " + y);
   x=y*2;
  }
  y=100; //erreur
  System.out.println("x =" + x );
 }
}
```

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.

```
int a, int b;
float resultat = a + b;
```

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.

Conversions de types et transtypage

```
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);
  }
}
```

Conversions de types et transtypage

Ce programme génère la sortie suivante :

```
Conversion of int to byte.
i and b 257 1  // valeur de l'entier plus grande que l'étendue
               // du byte => elle est réduite modulo l'étendue
               // du byte
Conversion of double to int.
d and i 323.142 323 // tronquée
Conversion of double to byte.
d and b 323.142 67 // réduite modulo 256
```

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 nom[];
```

Example

```
int mois[];
```

Bien que cette déclaration définisse *mois* comme une variable tableau d'entiers, il n'existe aucun tableau. La valeur de *mois* est fixée sur *null* qui représente un tableau sans valeur. Pour lier jour à un véritable tableau d'entiers, vous devez en définir un avec l'opérateur *new* et l'affecter à *mois*.

L'opérateur new et les tableaux

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

`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 erreur d'exécution.

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 $[0, p-1]$.

L'opérateur new et les tableaux

Opérateur pour allouer de la mémoire

```
int mois[]=new int[12];  // alloue un tableau de 12 entiers
                         // initialisés à 0 à mois
int deuxD[][]=new int[4][5];
double nom[]={10.1,11.2,12.3,13.4};

double m[][]={{0.0,1.1,2.0},
              {3.3,2.0,1.8}};
```

Arguments du programme principal

La fonction main admet comme paramètre un tableau de chaines de caractères *String[]*.
Ce tableau contient les arguments de la ligne de commande utilisée pour lancer l'application :

```
java appli arg0 arg1 ... argn
```

```
public class monappli{
  public static void main(String[] arg){
    int i;
    System.out.println("Nombre d'arg=" + arg.length);
    for(i=0;i<arg.length;i++)
      System.out.println("arg[" + i + "]=" + arg[i]);
    }
  }
}
```

```
java monappli a b c
Nombre d'arg=3
arg[0]=a
arg[1]=b
arg[2]=c
```

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

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`.

Opérateurs bit à bit

Applicables aux types `int`, `long`, `short`, `char`, et `byte`, qui agissent sur chaque bit de leurs opérandes.

Opérateurs de comparaison

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

Opérateurs logiques booléens

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;
```

Opérateur ?

L'expression

`expr_cond ? expr1:expr2`

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.

Priorité des opérateurs

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

Instruction de contrôle - 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`.

Example

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

Instruction de contrôle - 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`.

Instruction de contrôle - 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.

Instruction de contrôle - 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.

Instruction de contrôle - 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.

Instruction de contrôle - for

Les programmes suivants calculent tous la somme des n premiers nombres entiers d'un tableau a.

```
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+=a[i]; i++; }
```

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

Instructions de gestion de boucle

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

Instructions de gestion de boucle

Quitter une boucle avec break

```
class BreakBoucle{
  public static void main(String[] arg){
    for(int i=0;i<100;i++){
      if(i==10) break;  //Termine la boucle si i vaut 10
      System.out.println("i=" + i);
    }
    System.out.println("boucle terminée");
  }
}
```

Instructions de gestion de boucle

Illustre l'emploi de continue

```
class Continue{
  public static void main(String[] arg){
    for(int i=0;i<10;i++){
      System.out.println(i + " ");
      if(i%2) continue;
      System.out.println("");
    }
  }
}
```

Programme

Partie II :Technologie Objet et Java

Objet

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 de l'application.

Example

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.

Objet

Les objets sont décrit par

- un *état* : information permettant de le caractériser
- un *comportement* :décrit les actions que l'objet s'engage à réaliser

Example

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.

Example

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.

Objet - Message

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*).

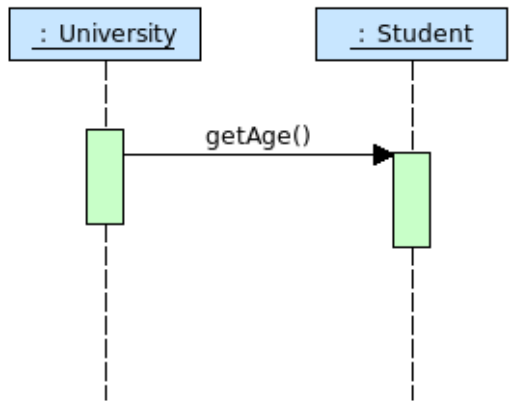


Figure: Envoi de message dans un diagramme de séquence

Objet - Méthode

Lorsque un objet reçoit un message, il exécute une action. Cette action est décrite par une *méthode*.

Example

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.

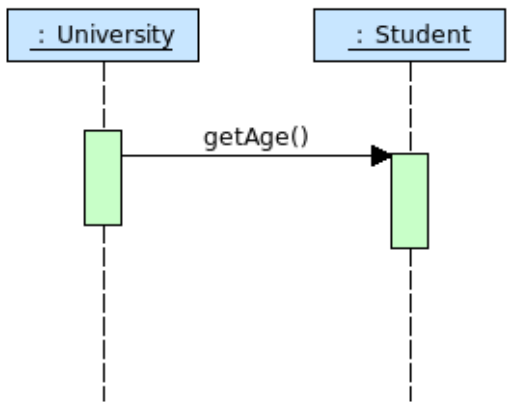


Figure: Activation / Execution

Système Orienté Objet

Un système OO est un mélange d'interactions entre objets pour atteindre un objectif commun.

Vers le concept de *classe*

Contrairement aux étudiants réels, les objets étudiant vont tous présenter le même comportement et ils vont tous porter la même information. Mais les valeurs des états de deux étudiants seront différentes puisque un matricule est unique.

$\rightsquigarrow$ Il faut que tous les objets soutiennent une abstraction unique

Classe

Cette abstraction est appelée 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 *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).

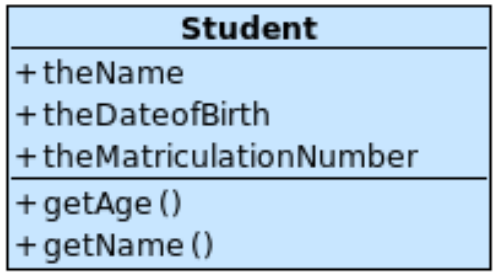


Figure: Diagramme de classe UML pour la classe `Student`

Instance

Un objet est alors l'*instance* d'une classe.

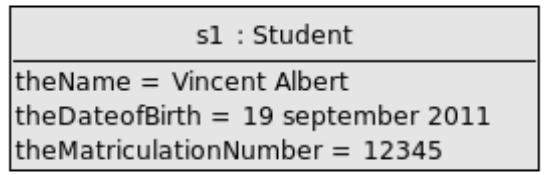


Figure: Une instance de `Student`

Exemple compte en banque

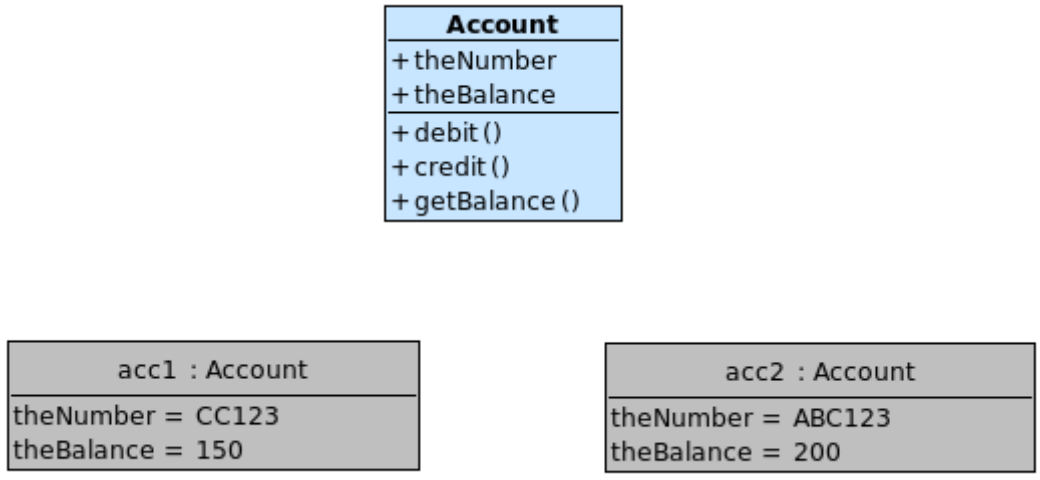


Figure: La classe `Account` et deux instances

Objets et relations

Les objets constituent un ensemble, deux objets sont reliés entre eux par des relations lorsqu'il doivent échanger des messages.

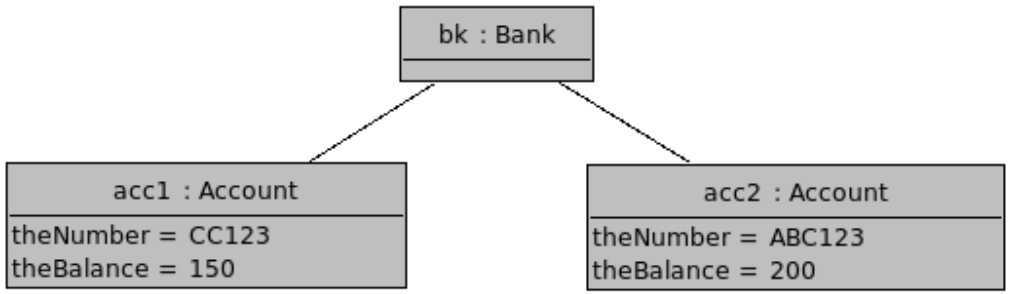


Figure: Objets et relations

Classes et relations

Figure: Classes et relations

Spécialisation

Concevoir 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 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.

Cette relation de généralisation se traduit par le concept d'*héritage*.

Spécialisation

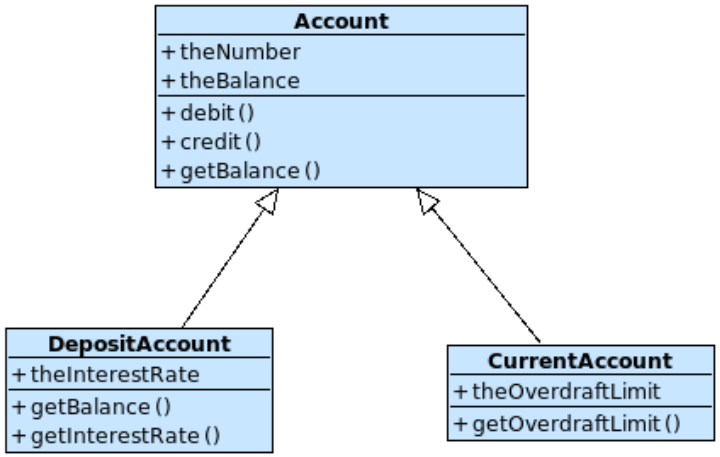


Figure: Deux sous-classes de `Account`

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.

Example

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.

Spécialisation

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, toute opération acceptant un objet d'une classe `Account` doit accepter un objet de la classe `DepositAccount`.

Niveaux de visibilité

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

Polymorphisme

Le concept de polymorphisme se traduit souvent par l'expression "peut prendre plusieurs formes".

Example

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.

Polymorphisme

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`.

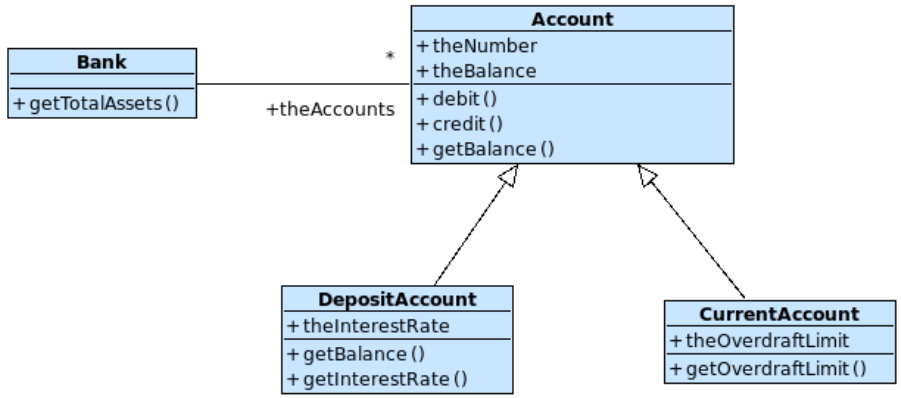


Figure: Diagramme de classe `Bank/Account`

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`.

Association

Relation entre deux classes, qui décrit les connexions structurelles entre leurs instances. 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.

Example

`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.

Agrégation et composition

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)

Définition d'une classe en Java

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
  ...
  }
  ...
}
```

Définition d'une classe en Java

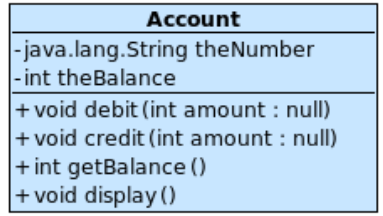


Figure: Classe `Account` et ses attributs

La *signature* ajoute des paramètres à l'opération et définit le type de valeurs, s'il existe, retourné par l'opération.

```
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);
  }
```

Constructeur d'une classe

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).

Constructeur d'une classe

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.

```
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);
  }
  ....
```

Surcharge d'opération

La surcharge de méthode consiste à garder le nom d'une méthode et à changer la liste ou le type de ses paramètres.

```
int a = 0, b= 0;
float c=1.5, d=1.5;
addition(a,b);
addition(c,d);
float resultat = addition(a,b);

public int addition(int x, int y)
{
  System.out.println("additionne des int");
  return x + y;
}

public float addition(float x, float y)
{
  System.out.println("additionne des float");
  return x + y;
}
```

Instancier un objet

Pour créer un objet on utilise l'opérateur `new`.

```
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;
```

ac3.theNumber générera une erreur d'exécution

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
  }
}
```

Notre première application

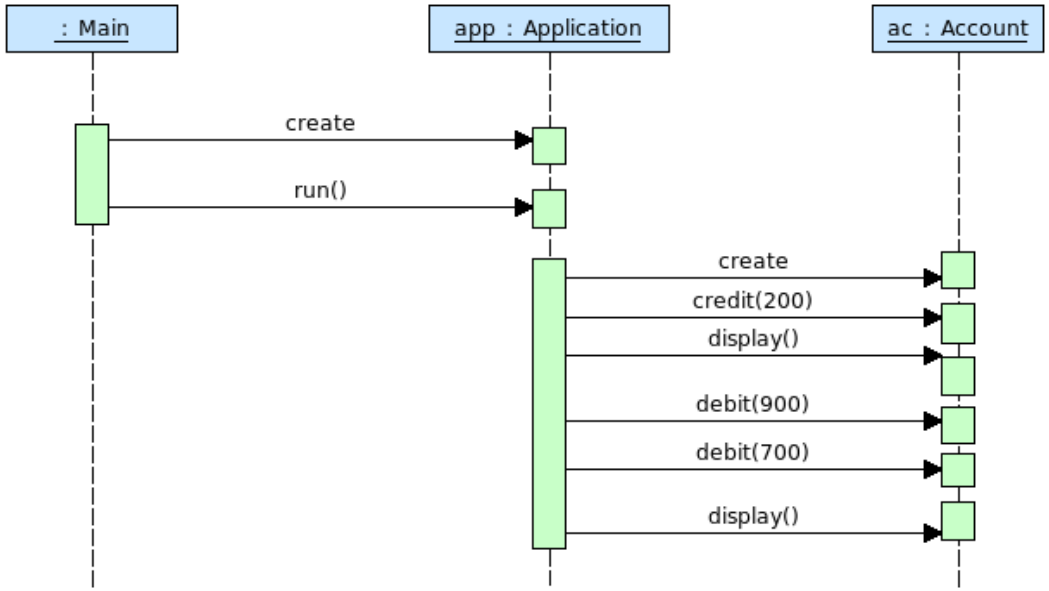


Figure: Diagramme de séquence pour Application

Collection d'objets

Une collection est un groupe d'objets. 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 unique 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

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.

On peut aussi dire quel sera le type d'objet dans une collection :

```
ArrayList<Account> accounts = new ArrayList();
```

Ajout d'objets dans une collection

Toutes les collections Java proposent une opération `add` permettant d'ajouter un objet à la collection.

```
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.

Ajout d'objets dans un HashSet

```
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, **ceci est autorisé** :

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

ou encore en utilisant une référence d'objet anonyme :

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

Ajout d'objets dans un 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 en fin de liste

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

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

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
accounts.add(new Accounts("ABC123",1200));  //objet anonyme
accounts.add(3,ac2);  //ajoute le compte ac2 à la position 4.
accounts.addFirst(ac3);
```

Suppression d'objets dans une collection

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

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

Accéder à un objet dans une LinkedList

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`.

```
for(int i=0; i<accounts.size();i++)
   Account ac1 = (Account)accounts.get(i);
```

Accéder à un objet dans un HashSet

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. Il nous faut pouvoir donc itérer à travers la collection.

- *Iterator*
- for-each loop construction depuis Java 5 (valable aussi pour les listes)

Itérateur

L'opération *iterator()* retourne un *Iterator* à la collection qui reçoit ce message

```
Iterator iterator()
```

un *Iterator* est un objet capable de parcourir la collection du début à la fin et permet les opérations :

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

Itérateur

si nous considérons les instructions suivantes :

```
HashSet<Account> accounts = new HashSet();
accounts.add(new Account("ABC123", 1200));
accounts.add(new Account("DEF456", 800));
```

voici l'utilisation de l'itérateur :

```
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;
   }
}
```

for-each loop construction

si nous considérons les instructions suivantes :

```
HashSet<Account> accounts = new HashSet();
accounts.add(new Account("ABC123", 1200));
accounts.add(new Account("DEF456", 800));
```

voici l'utilisation de la boucle for-each :

```
for(Account acc : accounts){
    String number = acc.getNumber();
    System.out.println(number);
}
```

Clonage de collections

Ceci ne fait pas un clonage de theAccounts :

```
ArrayList<Account> theAccounts = new ArrayList();
ArrayList<Account> anAccounts = theAccounts;
```

Ceci non plus :

```
ArrayList<Account> theAccounts = new ArrayList();
ArrayList<Account> anAccounts = new ArrayList();
anAccounts.addAll(theAccounts);
```

Notre première architecture

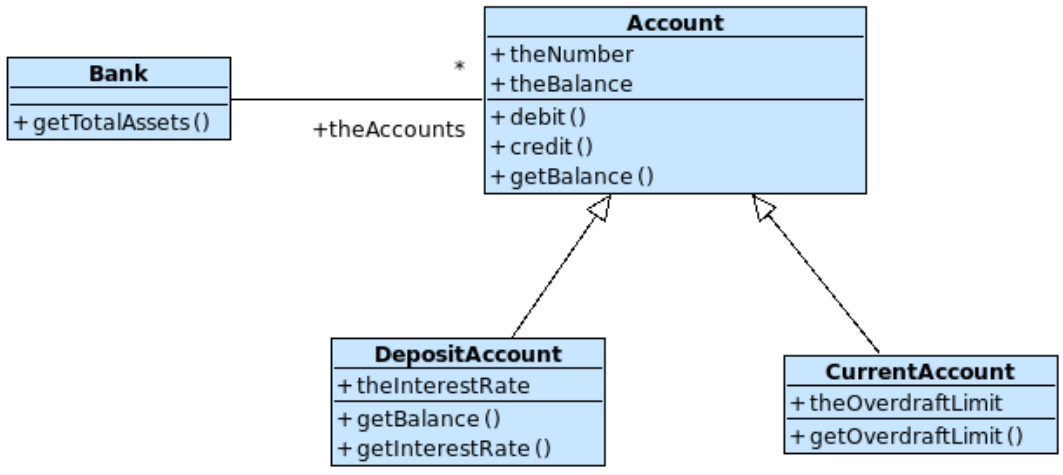


Figure: Diagramme de classe `Bank/Account`

Notre première architecture

Nous utilisons une collection de type `ArrayList` permettant de gérer la relation *une banque est associée à un ensemble de comptes*.

```
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<Account> theAccounts;
}
```

Notre deuxième application - Classe Bank

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

 public Bank(){
   this("");
 }

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

Notre deuxième application - Classe Bank

```
public void creditAccount(String aNumber, int anAmount){
 for(Account acc : theAccounts)
  if(aNumber.equals(acc.getNumber()))
   acc.credit(amount);
}

public void debitAccount(String aNumber, int anAmount){
 for(Account acc : theAccounts)
  if(number.equals(acc.getNumber()))
   acc.debit(amount);
}


public int getAccountBalance(String aNumber){
 for(Account acc : theAccounts)
  if(number.equals(acc.getNumber()))
   return acc.getBalance();
}

public int getTotalAssets(){
 int totalAssets = 0;
 for(Account acc : theAccounts)
  totalAssets += acc.getBalance();
}

//ATTRIBUTES
String theName;
//RELATIONS
private java.util.ArrayList<Account> theAccounts;
}
```

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());
 }
}
```

Spécialisation

Une entreprise embauche plusieurs employés qui sont aussi des personnes.

Figure: Relation de spécialisation

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.

Les attributs et les opérations hérités de la classe parent ne sont pas répétés dans la définition de la classe (**sauf bien sur en cas de redéfinition**) : on se contente d'indiquer les attributs et opérations rajoutés par la classe spécialisée.

Spécialisation

```
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;
}
```

La classe *Object*

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`.

Spécialisation

Typiquement nous pourrons utiliser ces classes de la manière suivante :

```
GregorianCalendar d1 = new GregorianCalendar(1973,0,15);
GregorianCalendar d2 = new GregorianCalendar(1983,11,22);

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 40
System.out.println(employee.getAge()); // affiche 29

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

L'instruction suivante est impossible :

```
person.getSalary()
```

Spécialisation

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. Notons que l'appel au constructeur de la classe parent ne crée pas d'objet personne.

Spécialisation

Si un attribut de la classe parent est déclaré `private`, cet attribut n'est pas accessible par les sous-classes. Il faut donc 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;
}
```

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.

Redéfinition

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);
}
```

Le mot clé `this` désigne l'objet courant : l'objet receveur du message

Redéfinition

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

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

il s'affichera sur la console :

```
Nom: Vincent
Age: 40

Nom:Jean
Age: 29
Salaire:1500
Numéro de référence: 1234
```

Polymorphisme

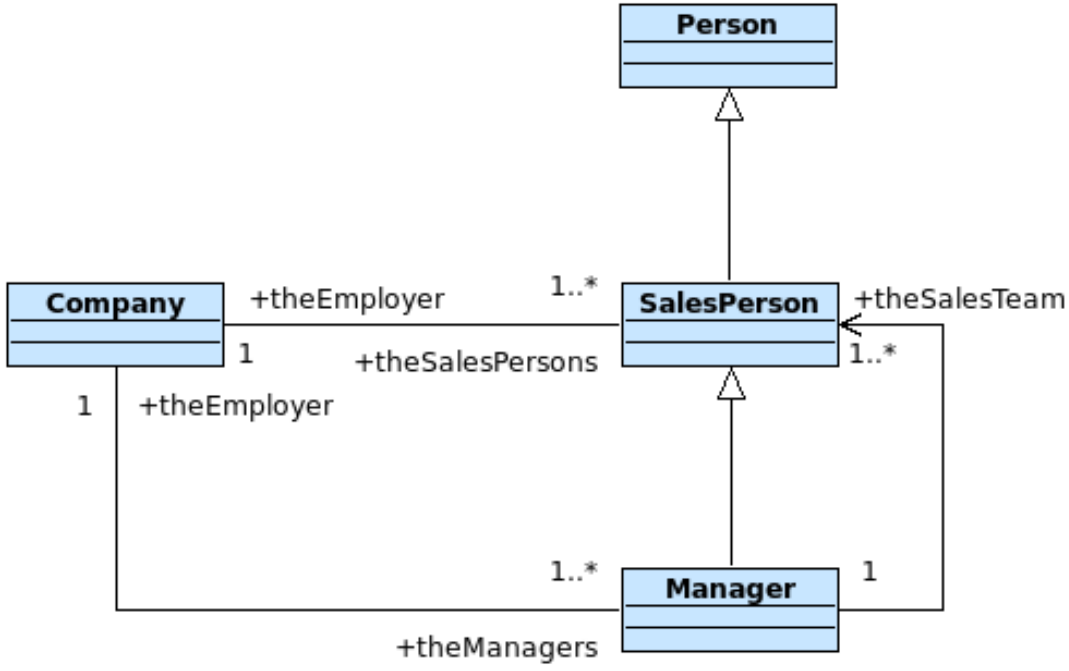


Figure: Diagramme de classe avec spécialisations

Polymorphisme

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` :

```
for(SalesPerson salesPerson : theSalesPersons)
  salesPerson.display();

for(Manager manager : theManagers)
  manager.display();
```

Polymorphisme

L'utilisation du polymorphisme donne la solution illustrée par le diagramme de classe :

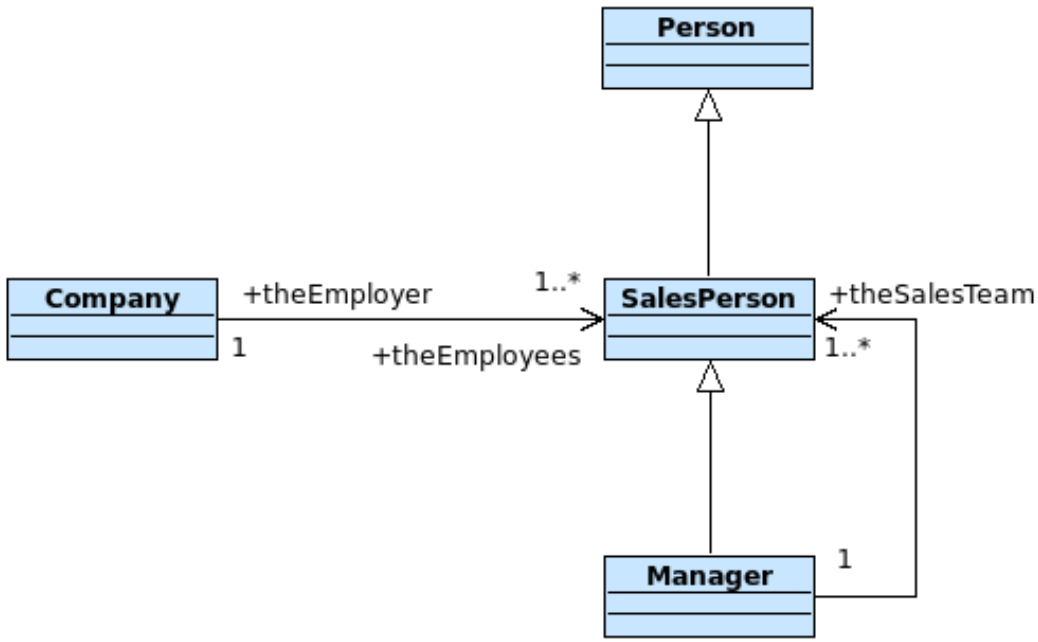


Figure: Diagramme de classe amélioré

Polymorphisme

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

```
for(SalesPerson employee : theEmployees)
  employee.display();
}
```

L'opération `display` dans `Person` est dite polymorphique. L'opération `display` dans `SalesPerson` est dite redéfinie.

Mot clé *final*

Si 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 être étendue : le mécanisme d'héritage est bloqué.
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.

Polymorphisme

Le polymorphisme permet de substituer un objet parent par un objet fils.

- Il est toujours possible d'utiliser une référence de la classe parent pour désigner un objet de la classe fille. Il y a transtypage implicite de la fille vers la mère.

Example

On peut utiliser une référence de type SalesPerson pour désigner un Manager

```
SalesPerson s;
Manager m = new Manager(...);
s=m;
```

Polymorphisme

- Le choix de la méthode à exécuter (pour une méthode polymorphe) ne se fait pas statiquement à la compilation mais dynamiquement à l'exécution en fonction du type réel de l'objet et non pas en fonction du type déclaré de la référence à l'objet.

Example

```
SalesPerson s;
Manager m = new Manager(...);
s=m;
s.display();
```

Quand j'invoque la méthode s.display() c'est la méthode de SalesPerson ou bien celle de Manager qui est exécutée ?

Polymorphisme

- Le choix de la méthode à exécuter (pour une méthode polymorphe) ne se fait pas statiquement à la compilation mais dynamiquement à l'exécution en fonction du type réel de l'objet et non pas en fonction du type déclaré de la référence à l'objet.

Example

```
SalesPerson s;
Manager m = new Manager(...);
s=m;
s.display();
```

Quand j'invoque la méthode s.display() c'est la méthode de SalesPerson ou bien celle de Manager qui est exécutée ?

Réponse Celle de Manager même si s fait référence à une SalesPerson car s désigne un employé de type Manager.

Polymorphisme

Soit une classe B qui hérite d'une classe A. B spécialise la classe A par l'opération op() (op() n'est donc pas définie par A).

```
A a = new B();
a.op();
```

Polymorphisme

Soit une classe B qui hérite d'une classe A. B spécialise la classe A par l'opération op() (op() n'est donc pas définie par A).

```
A a = new B();
a.op();
```

NON

Polymorphisme

Soit une classe B qui hérite d'une classe A. B spécialise la classe A par l'opération op() (op() n'est donc pas définie par A).

```
A a = new B();
((B)a).op();
```

Transtypage explicite de la classe parent vers la classe spécialisée : c'est à nous de vérifier que l'objet est bien du type dans lequel on transtype la référence.

Classe Person

```
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;
}
```

Classe SalesPerson

```
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("Salaire:" + theSalary + "\t" + "Numero de référence:"
         theReferenceNumber + "\t" + "Chiffre de vente:" + theSalesFigure);
   }

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

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

Classe Manager

```
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 cumulées:" + this.getAccumulatedSales());
   }

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

   //ATTRIBUTES
   private int theAccumulatedSales;

   //RELATIONS
   private java.util.HashSet theSalesTeam;
```

Classe Company

```
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);
      for(SalesPerson salesPerson : theEmployees)
        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;
}
```

Classe Application

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

Classe Application

donne la sortie :

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

Classe abstraite

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.**

Classe abstraite

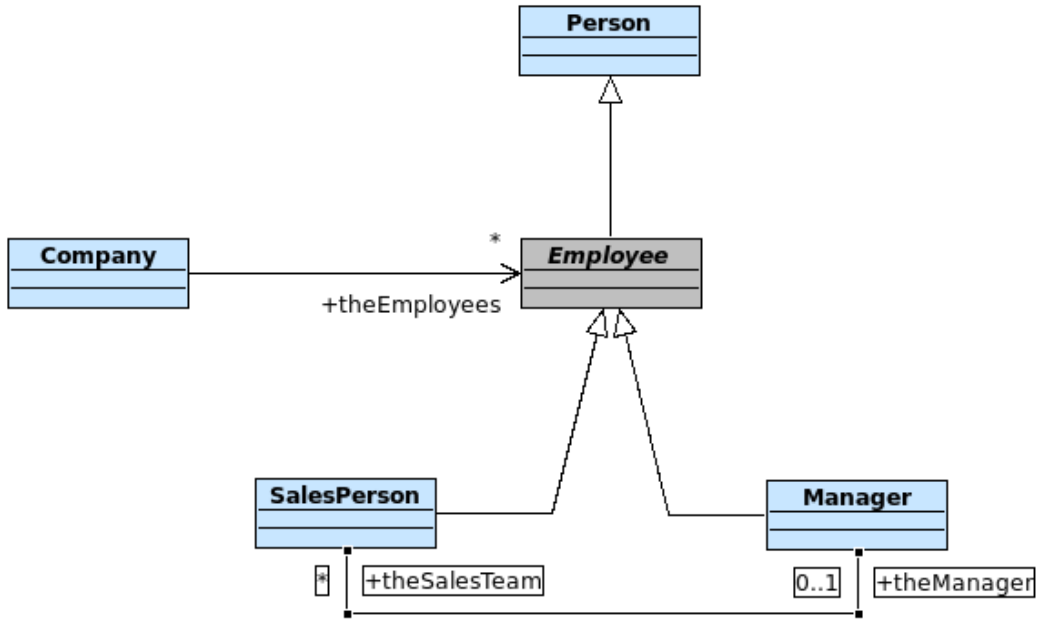


Figure: Diagramme de classe avec classe abstraite

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 classe `Employee` comme abstraite.

Classe 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(...);
```

Classe abstraite

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;
}
```

Opération abstraite

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.

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;
}
```

Nouvelle Application

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());
   System.out.println("Bonus pour " + s2.getName() + ":" + s2.getBonus());
   System.out.println("Bonus pour " + m1.getName() + ":" + m1.getBonus());
}
```

donne la sortie :

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

Surcharge d'opération addEmployee

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);
}
```

Interface

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.

Example

tous les employés de l'entreprise doivent implémenter les opérations `setReferenceNumber` et `display`.

- La classe de laquelle est originaire un employé doit avoir au moins ces opérations dans son interface publique.
- 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.

Interface

Nous pouvons modéliser cette situation avec une *interface* Java *Employable*

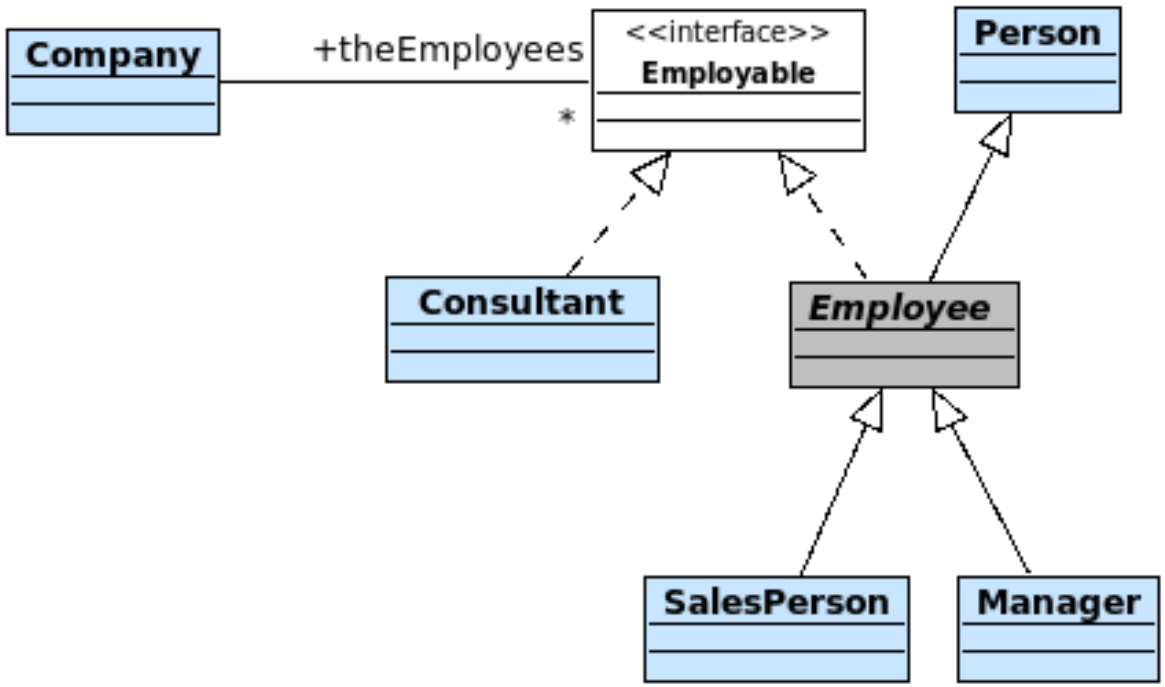


Figure: Diagramme de classe avec une interface

Interface

Maintenant une Company est associée à des objets Employable qui peuvent être issus de n'importe quelle hiérarchie de classes.

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

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(){...}
  //...
}
```

Interface

Et pour la classe Company nous avons :

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

     for(Employable Employee : theEmployees){
       employee.setReferenceNumber(referenceNumber);
       referenceNumber++;
     }
   }

   public final void displayEmployees(){
      System.out.println("Nom de l'entreprise:" + theName);
      for(Employable Employee : theEmployees)
        employee.display();
   }
   //...
}
```

Programme

Partie III : Quelques éléments de la bibliothèque Java

java.lang - Chaîne de caractères

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);
```

java.lang - Chaîne de caractères

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".

java.lang - Chaîne de caractères

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

Méthode de classe- mot clé *static*

On utilise le mot clé *static* pour représenter un attribut de la classe et non pas de l'objet créé à partir de cette classe.

```
public class Person{
  private static int nbPerson;
  private String theName;
  public Person(String aName){
    theName = aName;
    nbPerson++;
  }
  public static int getNbPerson(){
   return nbPerson;
  }
}

public static void main(String arg[]){
  Person p1 = new Person("Vincent");
  Person p2 = new Person("Joseph");
  System.out.println(Person.getNbPerson());
}
```

java.lang - 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.

java.util

Aller voir la librairie java.util pour tout ce qui à rapport aux collections

Gestion des exceptions

De nombreuses fonctions Java sont susceptibles de générer des exceptions, c'est à dire des erreurs.

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.

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`.

java.io - 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.

java.io - 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));
```

java.io - Lecture de données tapées au clavier

La construction d'un flux peut échouer : une erreur fatale, appelée exception en Java, est alors générée. 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);
}
```

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.

java.io - Écriture fichier

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).

java.io - Écriture fichier

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)
```

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.

java.io - Écriture fichier

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

java.io - Écriture fichier

```
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
```

java.io - Lecture fichier

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.

java.io - Lecture fichier

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

java.io - Lecture fichier

```
import java.util.*;
import java.io.*;
public class lire{
  public static void main(String[] arg){
    // 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;
    int age=0;
    // gestion des éventuelles erreurs
    try{
      while((ligne=IN.readLine())!=null){
        champs=ligne.split(",");
        prenom=champs[0];
        age=Integer.parseInt(champs[1]);
        System.out.println(""+new personne(prenom,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
```

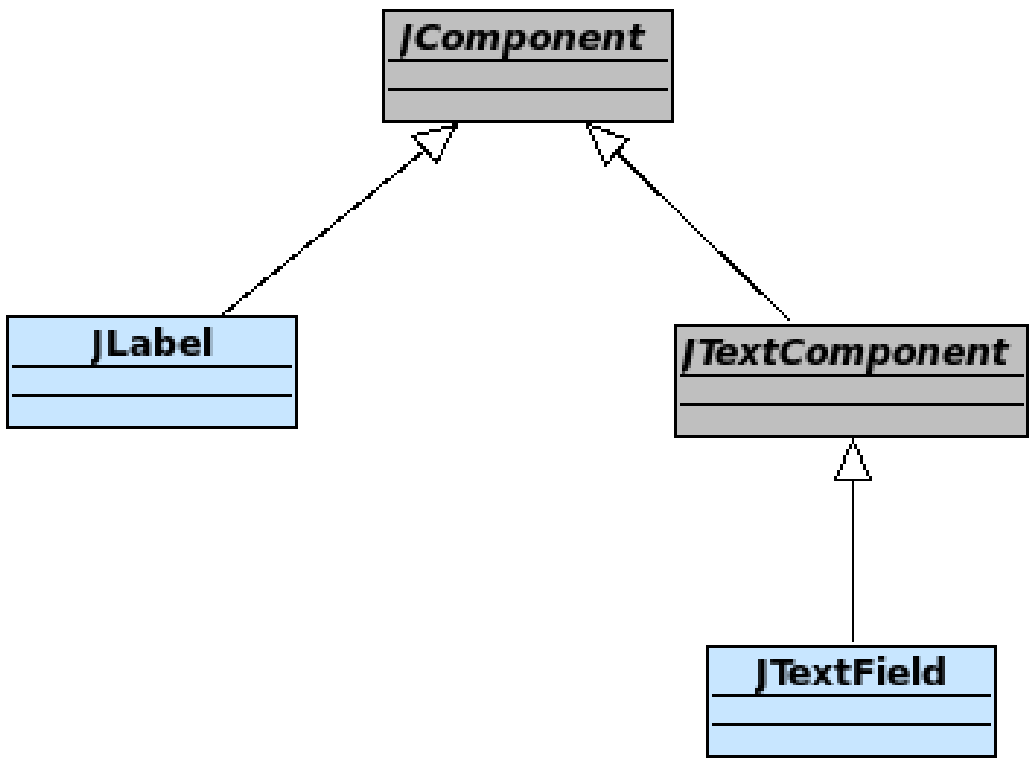
Programme

Partie IV : Interfaces graphiques

javax.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.

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.



Quelques JComponent

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

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 chaîne 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`.

```
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);
  }
}
```

Première application graphique

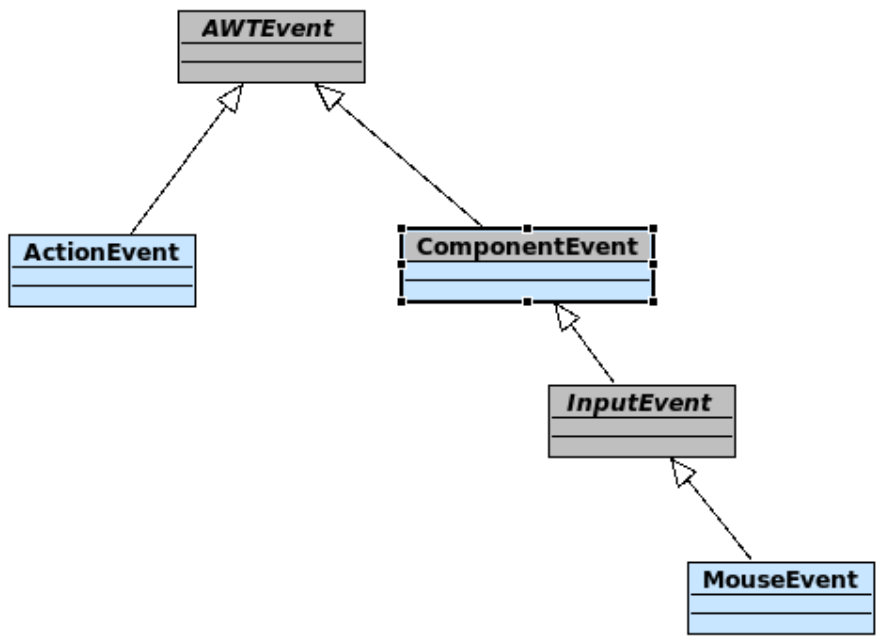
Dans cette deuxième version, nous avons implémenté l'application graphique dans une classe dédiée CompanyFrame.

```
// 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);
  }
}
```

java.awt.event - Les Événements

Les composants graphiques sont associés à des gestionnaires d'événements. Quand l'utilisateur bouge la souris, sélectionne un menu ou appui sur une touche du clavier, un événement survient, stimule l'application, et change éventuellement l'état du système. Les événements sont représentés par des objets de différentes classes événement.

- les mouvements de souris sont représentés par des objets issus de la classe `MouseEvent`.
- La sélection d'un item de menu est représentée par un objet issu de la classe `ActionEvent`.



Les Événements

Un objet événement dispose de différentes propriétés décrivant les aspects de l'événement. Par exemple, un objet issu de `MouseEvent` possède les coordonnées de la position de la souris au moment où est survenu l'événement.

```
MouseEvent event;
event.getX();
event.getY();
```

Un objet issu de ActionEvent possède le nom de l'action associé à cette événement :

```
ActionEvent event;
event.getActionCommand();
```

Les Gestionnaires d'événements

Un gestionnaire d'événements (listener) est un objet qui "écoute" et détecte les événements qui se produisent sur un composant graphique. Lorsqu'un événement est généré, le composant source de cet événement informe tous ses objets listener par un appel de message en lui passant en paramètre l'objet événement. Un listener implémente donc une méthode particulière permettant de recevoir ce message, tel que le définit l'interface qu'il implémente.

Dans le cas d'une action sur un item de menu ou un bouton il s'agit de l'interface `ActionListener` qui défini la méthode `actionPerformed`.

```
void actionPerformed(ActionEvent e)
```

Les Gestionnaires d'événements d'un JMenuItem

La classe AbstractAction est une classe qui implémente l'interface *ActionListener* pour un ensemble d'ActionEvent (événements sur bouton, item de menu...).

```
AbstractAction(String name)
AbstractAction(String name, Icon icon)
```

Soit un item de menu Exit permettant de fermer une application :

```
class ExitAction extends AbstractAction {
  public ExitAction(String label) {
    super(label);
  }

  public void actionPerformed(ActionEvent event) {
    System.out.println(event.getActionCommand());
    System.exit(0);
  }
}
```

Que l'on peut instancier avec :

```
private ExitAction exitAction = new ExitAction("Exit");
```

Les Gestionnaires d'événements de la JFrame

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 fenêtre
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
```

Les Gestionnaires d'événements de la JFrame

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

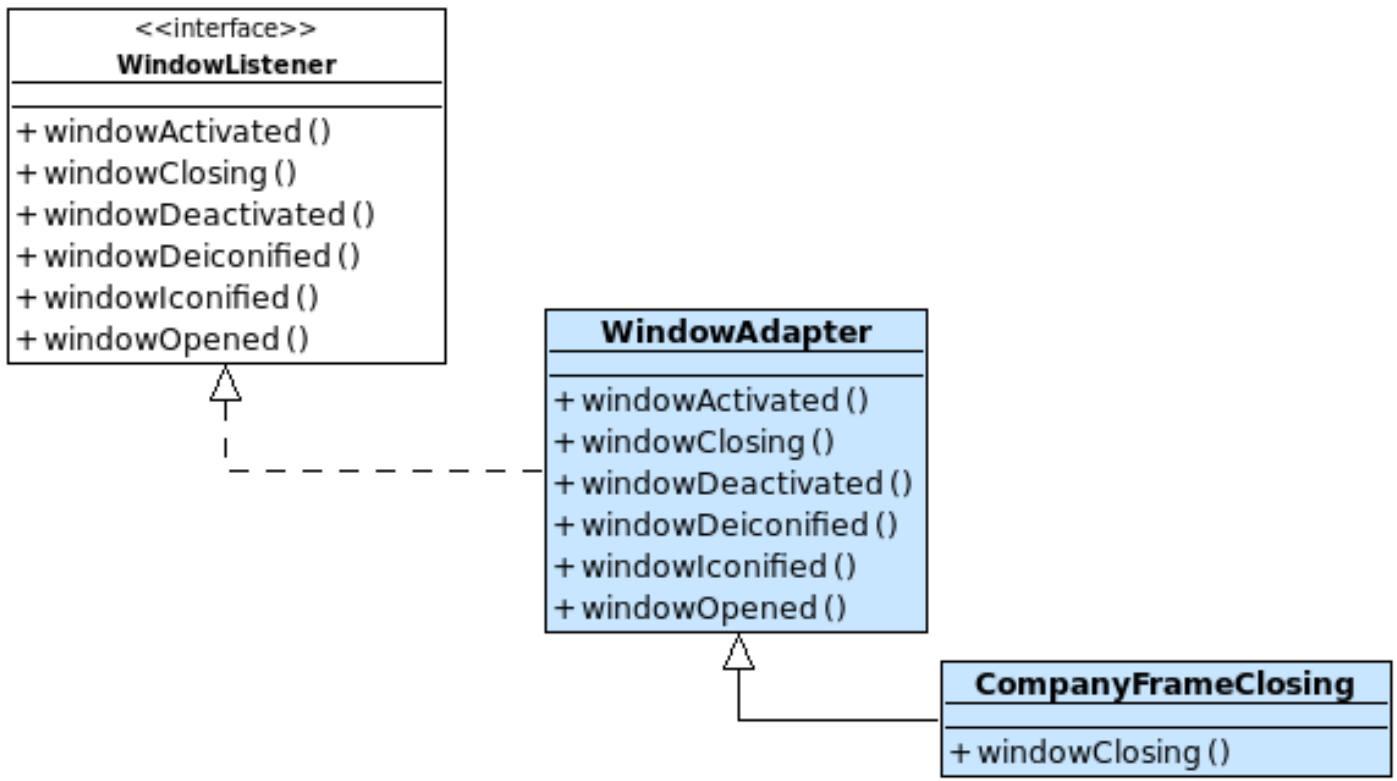


Figure: Notre gestionnaire d'événement

Associer un listener à un JComponent

On dira qu'un `JComponent` *enregistre* un listener. La méthode qui permet d'associer un listener à un composant graphique est appelée *méthode d'enregistrement*.

Par exemple *JButton* et *JMenuItem* héritent de la méthode *addActionListener(ActionListener l)* de la classe *AbstractButton* :

```
JMenuItem exitMenuItem = new MenuItem("Fermer");
exitMenuItem.addActionListener(new ExitAction("Exit"));
```

Une *JFrame* héritent de la méthode *addWindowListener(WindowListener l)* de la classe *Window* :

```
public class CompanyFrame extends JFrame{
  public CompanyFrame(String caption){
    ...

    this.addWindowListener(new CompanyFrameClosing());
  }
```

Associer un listener à un JComponent

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'élément de menu, l'u-tilisateur 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, entrée/sortie du domaine d'un composant, bouton pressé, relâche)
KeyListener		public void addKeylis-tener(KeyListener)	événement clavier (touche tapée, pressée, relachée)

Gestion de la fermeture de la JFrame

```
//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);
     }
  }
}
```

Gestion de la fermeture de la JFrame

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

Barre 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);
```

Menus et items - Version détaillée

```
void add(JMenuItem menuItem)
```

Les menus et les items sont préparés comme ceci :

```
JMenu fileMenu = new JMenu("File");
fileMenu.setMnemonic('F');
JMenuItem exitMenuItem = new MenuItem("Fermer");
exitMenuItem.addActionListener(new ExitAction("Exit"));
exitMenuItem.setMnemonic('x');
fileMenu.add(exitMenuItem);
theMenuBar.add(fileMenu);
this.setJMenuBar(theMenuBar);
```

Menus et items - Version simplifiée

```
JMenuItem add(Action a)
```

Les menus et les items sont préparés comme ceci :

```
JMenu fileMenu = new JMenu("File");
fileMenu.setMnemonic('F');
JMenuItem exitMenuItem = fileMenu.add(exitAction);
exitMenuItem.setMnemonic('x');
theMenuBar.add(fileMenu);
this.setJMenuBar(theMenuBar);
```

Notre fenêtre de base avec barre de menu

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

public class CompanyFrame extends JFrame{
  public CompanyFrame(String caption){
    ...
    this.assembleMenuBar();
  }

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

  //ATTRIBUTES
  private JMenuBar theMenuBar = new JMenuBar();
  private ExitAction exitAction = new ExitAction("Exit");

  // CLASSES INTERNES
  public class ExitAction extends AbstractAction{
     public ExitAction(String label){
        super(label);
     }
     public void actionPerformed(ActionEvent event){
        System.exit(0);
     }
  }
```

Classe interne

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

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

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

Pattern Singleton

Le patron de modélisation singleton permet de restreindre l'instanciation des classes a un seul objet par classe. Pour cela chaque classe possède un attribut statique qui va permettre de stocker l'unique instance de la classe qui est créée au démarrage de l'application. Leur constructeur est protégé et une méthode statique *get_nom_de_la_classe* permet de retourner cette unique instance.

```
public class CompanyFrame extends JFrame{

  protected CompanyFrame(String caption){ ... }

  public static CompanyFrame getCompanyFrame() { return companyFrame; }

  //ATTRIBUTS
  private static CompanyFrame companyFrame =
                                    new CompanyFrame("Gestionnaire de Company");
}
```

Pattern Model-View-Controller

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

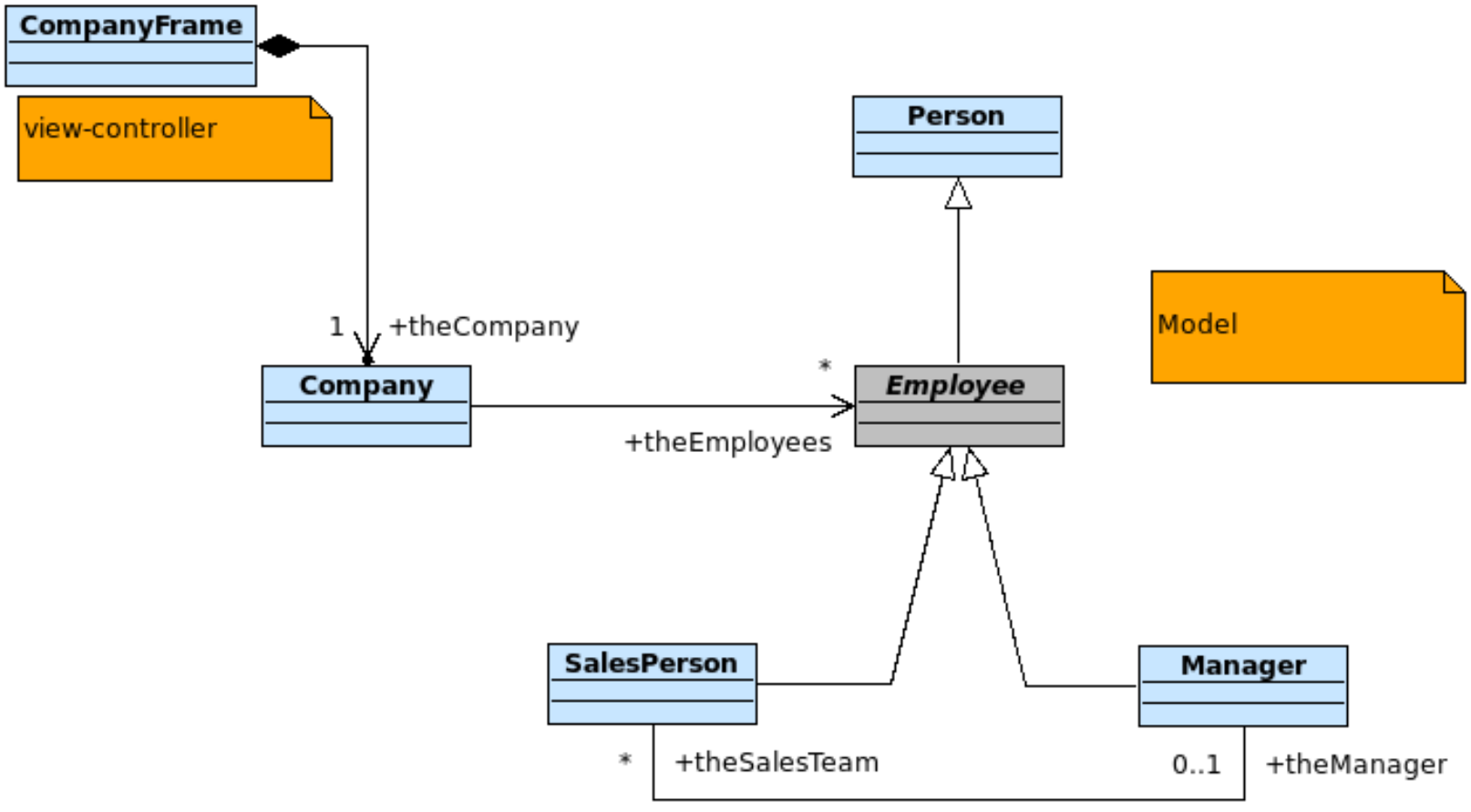


Figure: Patron de conception MVC