La Programmation Par Objet

Amélie Lambert

USSE09 - Java

Développement d'applications

Développement d'applications

L'élaboration d'une application est progressive : du 1er modèle, description des besoins du client au dernier, programme complètement testé.

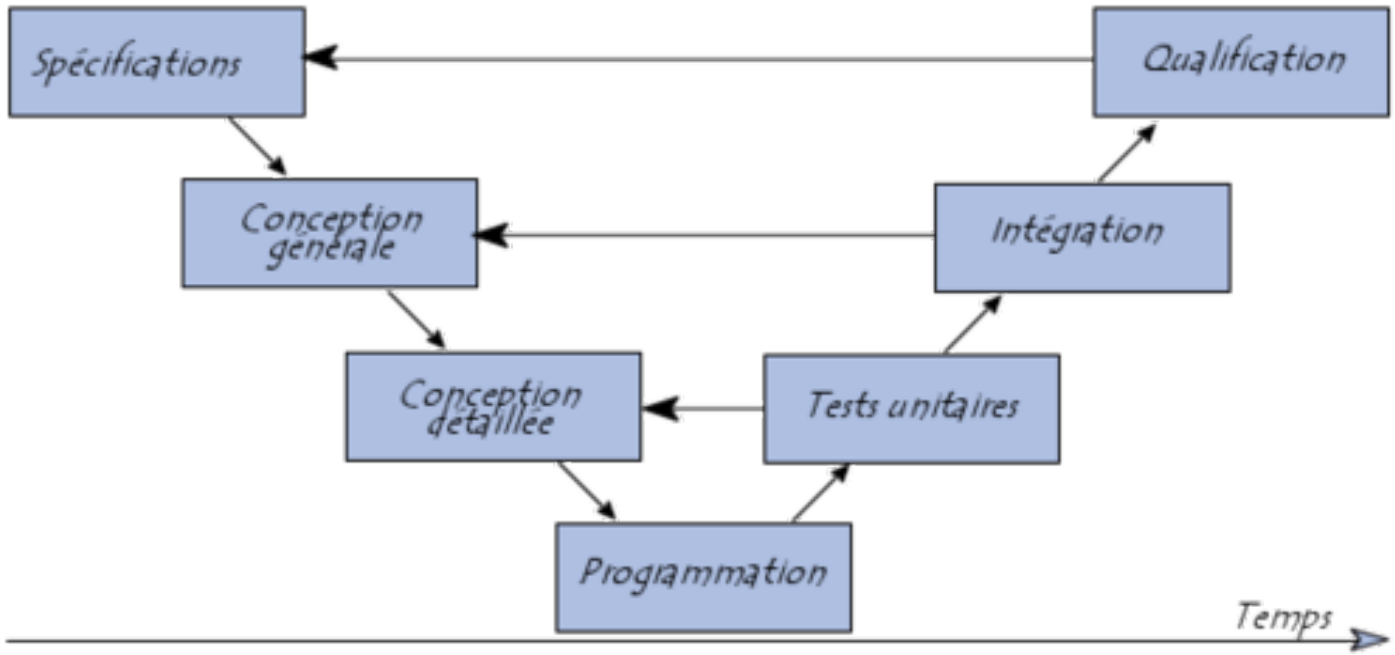
Développement d'applications

L'élaboration d'une application est progressive : du 1er modèle, description des besoins du client au dernier, programme complètement testé.

Les étapes du développement d'applications :

1. Analyse : spécification des fonctionnalités du système
2. Conception : architecture
3. Conception détaillée : élaboration des modules
4. Programmation
5. Tests unitaires
6. Tests d'intégration
7. Qualification

Développement d'applications



Etape 1 du développement d'applications : Spécification

- Dans cette étape, on s'attache à répondre à trois grandes questions :
  - comprendre le problème, **identifier** les **données** et les **résultats attendus**
  - dégager les grandes **fonctionnalités** du système (spécification fonctionnelle)
  - identifier les **ressources** nécessaires (matérielles et humaines)
- Cette phase est **indépendante de tout langage de programmation**

Etapes 2/3 du développement d'applications : Conception

- La conception consiste à **modéliser l'architecture du problème**

- Pour cela, on décompose le problème en sous-problèmes plus simples :
  - Ceci donne naissance à un ensemble d'unités informatiques ou **modules** qui constituent le logiciel d'application.
  - Les **modules** et leurs **relations** sont spécifiés.

- **Attention :** ce processus est itératif et peut conduire à un retour vers la spécification.

Etapes 4 du développement d'applications : Programmation

- Il s'agit, dans cette étape, de **programmer** (coder) la conception, c'est à dire l'ensemble des **modules** de l'application.

- Plus précisément, il s'agit de traduire les traitements en terme de **structures de contrôle** et les données en termes des **structures de données** du langage de programmation choisi.

Etapes 5/6/7 du développement d'applications : Validation/Vérification

- **Tests unitaires (niveau module)** :

  chaque module est testé individuellement

- **Tests d'intégration (niveau général)**

  les modules sont testés ensemble

- **Tests de qualification**

  Est ce que l'on répond aux spécifications

Etapes 5/6/7 du développement d'applications : Validation/Vérification

- **Tests unitaires (niveau module)** :

  chaque module est testé individuellement

- **Tests d'intégration (niveau général)**

  les modules sont testés ensemble

- **Tests de qualification**

  Est ce que l'on répond aux spécifications

L'évolution du système est contrôlée via le processus

conception/programmation/test

Les modules

La programmation modulaire et les modules

Définition : La *programmation modulaire* consiste à décomposer une grosse application en modules qui peuvent être :

- des groupes de fonctions
- des groupes de méthodes et de traitement

Elle permet de pouvoir développer les modules indépendamment et donc :

- La réutilisabilité (dans d'autres applications)
- L'extensibilité (ajout de nouvelles fonctionalités)

La programmation modulaire et les modules

Définition : La *programmation modulaire* consiste à décomposer une grosse application en modules qui peuvent être :

- des groupes de fonctions
- des groupes de méthodes et de traitement

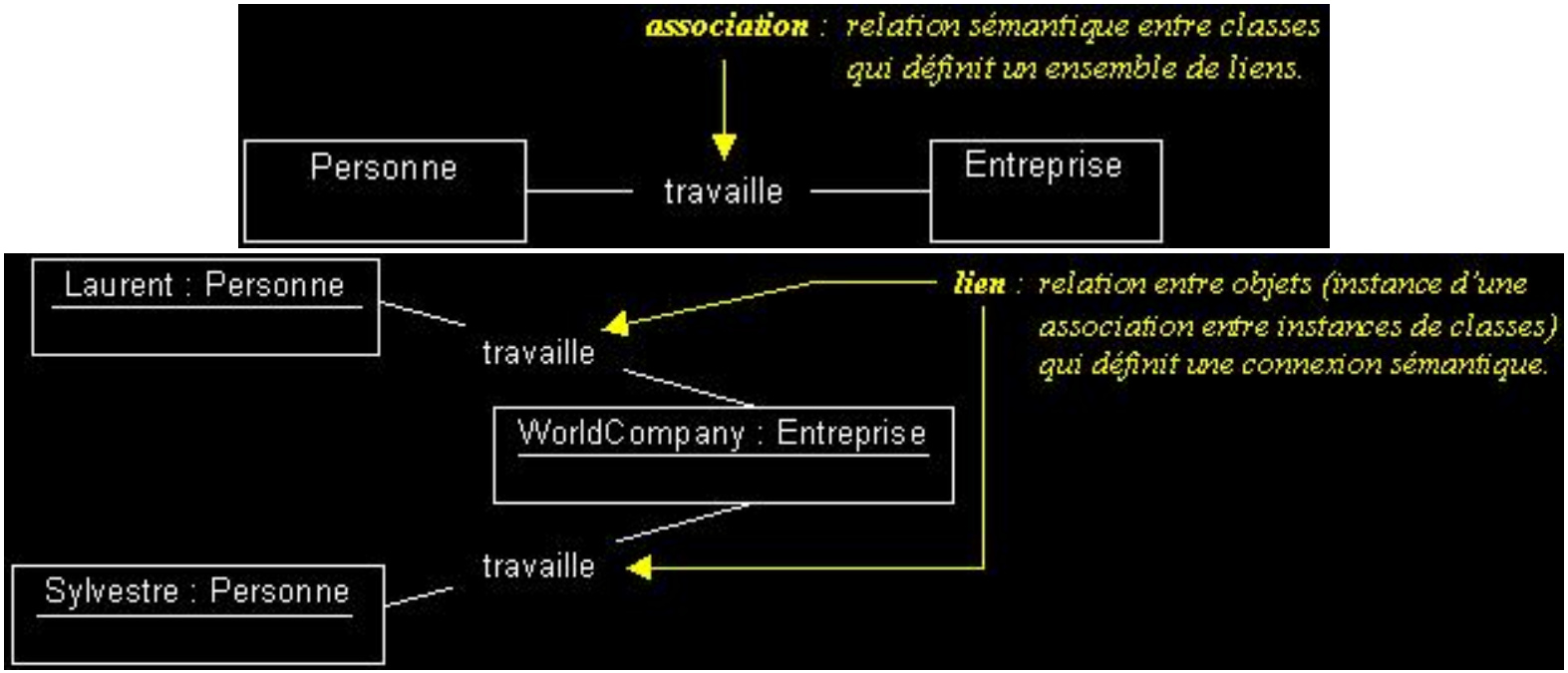
Elle permet de pouvoir développer les modules indépendamment et donc :

- La réutilisabilité (dans d'autres applications)
- L'extensibilité (ajout de nouvelles fonctionalités)

Développement d'un module

- *par spécification :* définition des modules et des leurs relations
- *par spécialisation :* Héritage

Exemple de developpement par spécification : diagrammes de classe UML (Unified Modeling Language)



De la spécification à la programmation d'un module

La **spécification** est un **modèle** du module, la description ce que le module doit faire.

On distingue :

- les spécifications informelles, par exemple un texte en français.
- les spécifications semi-formelles avec une syntaxe plus précise ou comportant des diagrammes plus ou moins standardisés.
- les spécifications formelles qui présentent une syntaxe et une sémantique.

La **programmation** consiste à coder les spécifications formelles.

Exemple 1 : Spécifications informelles

1. On souhaite représenter des rectangles dans un programme informatique,

Exemple 1 : Spécifications informelles

1. On souhaite représenter des rectangles dans un programme informatique,
2. La hauteur et la largeur de ces rectangles peuvent varier,

Exemple 1 : Spécifications informelles

1. On souhaite représenter des rectangles dans un programme informatique,
2. La hauteur et la largeur de ces rectangles peuvent varier,
3. Il faut pouvoir calculer la surface d'un tel rectangle,

Exemple 1 : Spécifications informelles

1. On souhaite représenter des rectangles dans un programme informatique,
2. La hauteur et la largeur de ces rectangles peuvent varier,
3. Il faut pouvoir calculer la surface d'un tel rectangle,
4. Il faut pouvoir déplacer ces rectangles dans l'espace qui est borné entre deux constantes (abscisse max et ordonée max).

Exemple 1 : Spécifications semi-formelles

1. Il faut définir ce qu'est un point de l'espace. Il sera caractérisé par son abscisse et son ordonnée. Il faut pouvoir initialiser :
   - un point par défaut (origine),
   - un point à partir de ses coordonnées,
   - un point à partir d'un autre point.

Exemple 1 : Spécifications semi-formelles

1. Il faut définir ce qu'est un point de l'espace. Il sera caractérisé par son abscisse et son ordonnée. Il faut pouvoir initialiser :
   - un point par défaut (origine),
   - un point à partir de ses coordonnées,
   - un point à partir d'un autre point.

2. Il faut définir ce qu'est un rectangle. Il sera caractérisé par sa hauteur, sa largeur et son origine. Il faut pouvoir initialiser :
   - un rectangle vide,
   - un rectangle à partir d'un point origine,
   - un rectangle à partir des coordonnées de son origine,
   - un rectangle à partir d'un point origine, d'une largeur et d'une hauteur.

Exemple 1 : Spécifications semi-formelles

1. Il faut définir ce qu'est un point de l'espace. Il sera caractérisé par son abscisse et son ordonnée. Il faut pouvoir initialiser :
   - un point par défaut (origine),
   - un point à partir de ses coordonnées,
   - un point à partir d'un autre point.
2. Il faut définir ce qu'est un rectangle. Il sera caractérisé par sa hauteur, sa largeur et son origine. Il faut pouvoir initialiser :
   - un rectangle vide,
   - un rectangle à partir d'un point origine,
   - un rectangle à partir des coordonnées de son origine,
   - un rectangle à partir d'un point origine, d'une largeur et d'une hauteur.
3. Il faut pouvoir calculer la surface d'un rectangle.

Exemple 1 : Spécifications semi-formelles

1. Il faut définir ce qu'est un point de l'espace. Il sera caractérisé par son abscisse et son ordonnée. Il faut pouvoir initialiser :
   - un point par défaut (origine),
   - un point à partir de ses coordonnées,
   - un point à partir d'un autre point.
2. Il faut définir ce qu'est un rectangle. Il sera caractérisé par sa hauteur, sa largeur et son origine. Il faut pouvoir initialiser :
   - un rectangle vide,
   - un rectangle à partir d'un point origine,
   - un rectangle à partir des coordonnées de son origine,
   - un rectangle à partir d'un point origine, d'une largeur et d'une hauteur.
3. Il faut pouvoir calculer la surface d'un rectangle.
4. Il faut pouvoir déplacer un rectangles dans l'espace.

Exemple 1 : Spécifications formelles et programmation

```
public class Point {
    public int x;
    public int y;
    /**
    * Constructeur sans paramètre le point est en (0,0)
    */
    public Point() {
      x=0;
      y=0;}
    /**
    * Constructeur avec paramètres abscisse et ordonnee
    * @param abscisse abscisse du point
    * @param ordonnee ordonnee du point
    */
    public Point(int abscisse, int ordonnee) {
      x = abscisse;
      y=ordonnee;}
    /**
    * Constructeur avec paramètre p, le point est créé en (p.x,p.y)
    * @param p duquel on extrait les coordonnées
    */
    public Point(Point p) {
      x=p.x;
      y=p.y;}
}
```

Exemple 1 : Spécifications formelles et programmation

```
public class Rectangle {
    public int largeur = 0;
    public int hauteur = 0;
    public Point origine;
    public final static int XMAX = 100;
    public final static int YMAX = 100;

    /**
    * Constructeur sans paramètre, on crée le rectangle où
    * l'origine est en (0,0), la largeur vaut 0 et la hauteur vaut 0.
    */
    public Rectangle() {
      origine = new Point();
    }
    /**
    * Constructeur avec paramètre p, on crée le rectangle où l'origine
    * est en (p.x,p.y), la largeur vaut 0 et la hauteur vaut 0.
    * @param p Point origine
    */
    public Rectangle(Point p) {
      origine = p;
    }
```

Exemple 1 : Spécifications formelles et programmation

```
 /**
 * Constructeur avec paramètres l et h, on crée le rectangle où
 * l'origine est en (0,0), la largeur vaut l et la hauteur vaut h.
 * @param l largeur du rectangle
 * @param h hauteur du rectangle
 */
 public Rectangle(int l, int h){
  origine=new Point(0,0);
  largeur = l;
  hauteur = h;
 }
/**
 * Constructeur avec paramètres p, l et h, on crée le rectangle où
 * l'origine est en (p.x,p.y), la largeur vaut l et la hauteur vaut
 * @param p Point origine
 * @param l largeur du rectangle
 * @param h hauteur du rectangle
 */
 public Rectangle(Point p,int l,int h){
  origine = p;
  largeur = l;
  hauteur = h;
}
```

Exemple 1 : Spécifications formelles et programmation

```
  /**
   * Deplace l'origine d'un rectangle en (x,y)
   * @param x abscisse de la position finale
   * @param y ordonnée de la position finale
   *
   */
    public void deplacer(int x, int y){
      if (0<=x  x <=XMAX  0<=y  y<=YMAX)
        origine.x=x;
        origine.y=y;
    }

    /**
    * Calcule la surface d'un rectangle
    * @return la surface du rectangle
    */
    public int surface() {
      return largeur * hauteur;
    }
}
```

Programmation par objet

La notion d'objet

Notion d'objet

- Un **objet** est une **unité de structuration** du logiciel.
- Un objet :
  - possède un **état** (sa mémoire propre) constitué de valeurs (ses données).
  - possède des **actions** qui peuvent agir sur cet état pour éventuellement le modifier
  - **contrôle ce qu'un utilisateur peut voir et utiliser**.
- L'intérêt du concept d'objet est double :
  - structurer le logiciel
  - assurer la protection contre d'éventuelles erreurs

Caractéristiques d'un objet (1/2)

- Un objet **encapsule** les **données** (attributs) et les **méthodes** (fonctions qui déterminent son comportement).

  Les données et les méthodes sont par conséquent intimement liées au sein de l'objet.

- Un objet présente l'ensemble des services qu'il peut rendre sous la forme d'une **interface**. On parle **d'abstraction de données** car il sépare les deux aspects :
  - **L'aspect client**, l'interface contenant une vue abstraite des services qu'il peut rendre est rendue visible
  - **L'aspect serveur**, la manière dont les services sont rendus est tenue secrète.

Caractéristiques d'un objet (2/2)

Tout objet peut communiquer avec d'autres objets au travers d'interfaces bien définies.

Cependant, ils ne sont pour autant pas autorisés à savoir comment ceux-ci sont implémentés.

Un objet possède la propriété de **masquer l'information**.

Les détails de la programmation sont cachés au sein des objets.

Un exemple d'objet

Un magnétoscope possède les caractéristiques d'un objet :

- Un magnétoscope possède une structure propre. Il résulte d'un assemblage de modules spécifiques (**ses données**),
- L'utilisateur peut interagir avec lui au moyen de boutons qui constituent son interface et remplissent des fonctions précises (**ses méthodes**),
- Un magnétoscope **encapsule** au sein de sa structure ses données et ses méthodes,
- L'utilisateur n'a pas besoin de connaître le fonctionnement interne, ni la technologie sous-jacente (**masquage de l'information**),
- Il peut être connecté à un autre objet tel qu'un récepteur TV ou un camescope pour former un système électronique.

La conception par objets

Les types de programmation

- La **programmation procédurale** envisage un programme comme une suite de commandes passées à une machine (Pascal, Fortran, basic,...)
- La **programmation fonctionnelle** place la notion de fonction au centre de la programmation. Un programme est vu comme une fonction qui met en correspondance un ensemble de données et un ensemble de résultats (Lisp, Scheme, ML,...)
- Dans la **programmation par objets**, un programme est conçu comme une sorte de modèle de la réalité (du domaine de l'application) (Ada)
- La **programmation orientée objets (POO)** propose la même vision avec des concepts plus avancés, en particulier les concepts d'héritage, d'interface et de polymorphisme. C++ et Java supportent l'ensemble de ces concepts.

Notion de modèle

Le développement d'un système part d'une spécification imprécise, floue, avec éventuellement des ambiguïtés et/ou des inconsistances. Elle est exprimée en langue naturelle dans un cahier des charges.

Un **modèle est une abstraction** :

- il présente **le point de vue d'un observateur sur un aspect du monde réel** (isolation de certaines propriétés d'un objet)

- il est décrit dans un formalisme dont la **syntaxe** et la **sémantique** sont clairement définies

Méthodologies

Analyse : spécification des fonctionnalités du système

Agenda
tableau:Page[]
noterRDV(String,Date) modifierRDV(String,Date) supprimerRDV(Date)

Conception : architecture et élaboration des modules :

Programmation :

```
public class Agenda{
 Page[] tableau;
 public void noterRDV(String rdv,Date date){
     ....}
 public void modifierRDV(String rdv,Date date){
     ...}
 public void supprimerRDV(Date date){
     ...}
}
```

Les classes

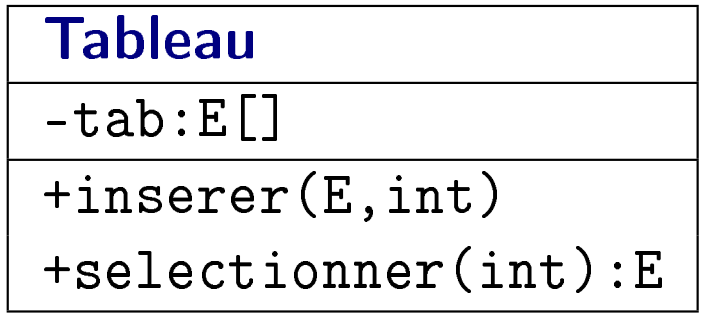
Notion de classe

- Les classes sont des plans de construction pour un ensemble d'objets. Une classe est une sorte de modèle pour décrire une collection d'objets. Différentes classes décrivent différents ensembles d'objets.
- Un objet est une **instance** (un exemplaire) d'une classe
- Toute classe définit :
  - Les données caractéristiques de ses objets. Ce sont ses **variables d'instance (ou attributs)**. Chaque objet d'une classe possède une copie de l'ensemble des variables d'instance avec des valeurs spécifiques qui forme l'état de l'objet.
  - L'ensemble des actions que l'on peut effectuer sur les objets de la classe. Elles sont appelées **méthodes**. Elles agissent sur l'état de l'objet concerné.

Représentation UML d'une classe

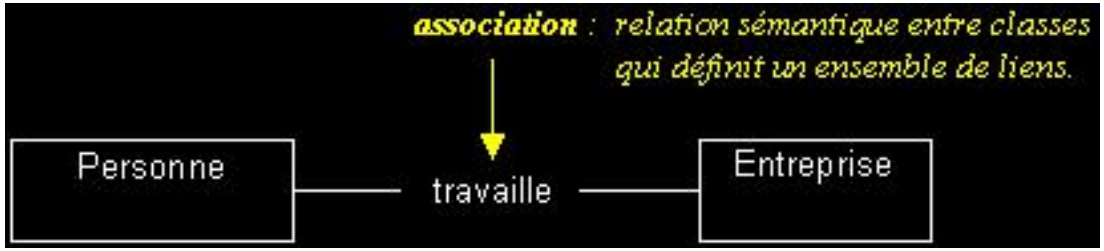
- La spécification d'un objet est décrite par une interface. L'interface est implémentée par une classe.
- **Remarque :** en Java, l'interface peut être décrite explicitement ou bien contenue dans la définition de la classe
- **Exemple de classe :** la classe `Tableau`

Les signes - et + indiquent les droits d'accès aux membres de la classe



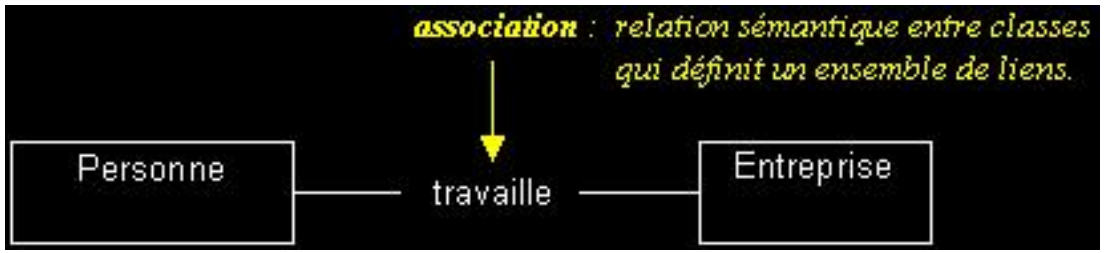
Différence entre classe et objet

On définit les classes `Personne` et `Entreprise` et leur relation (ici une association) `travaille`

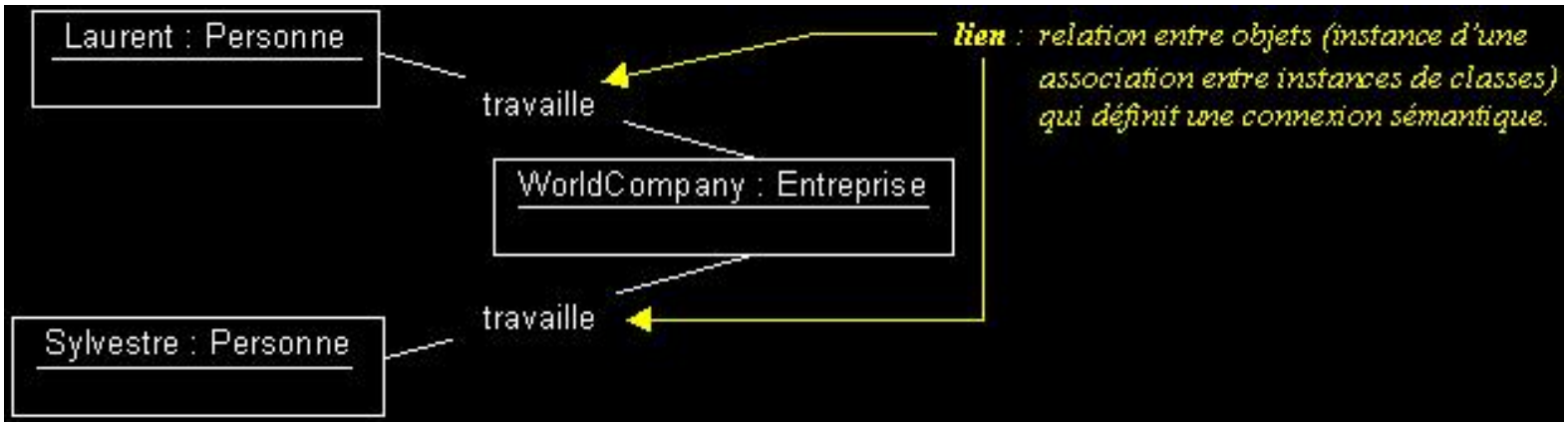


Différence entre classe et objet

On définit les classes `Personne` et `Entreprise` et leur relation (ici une association) `travaille`

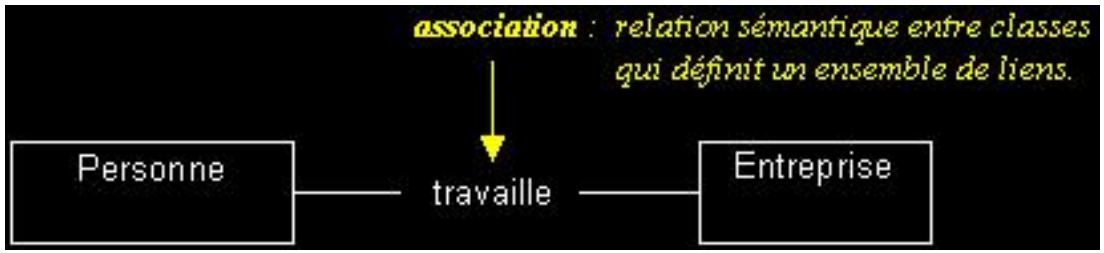


On instancie deux objets `Personne` (Laurent et Sylvestre) et un objet `Entreprise` (WorldCompany) ainsi que le lien entre ces objets :

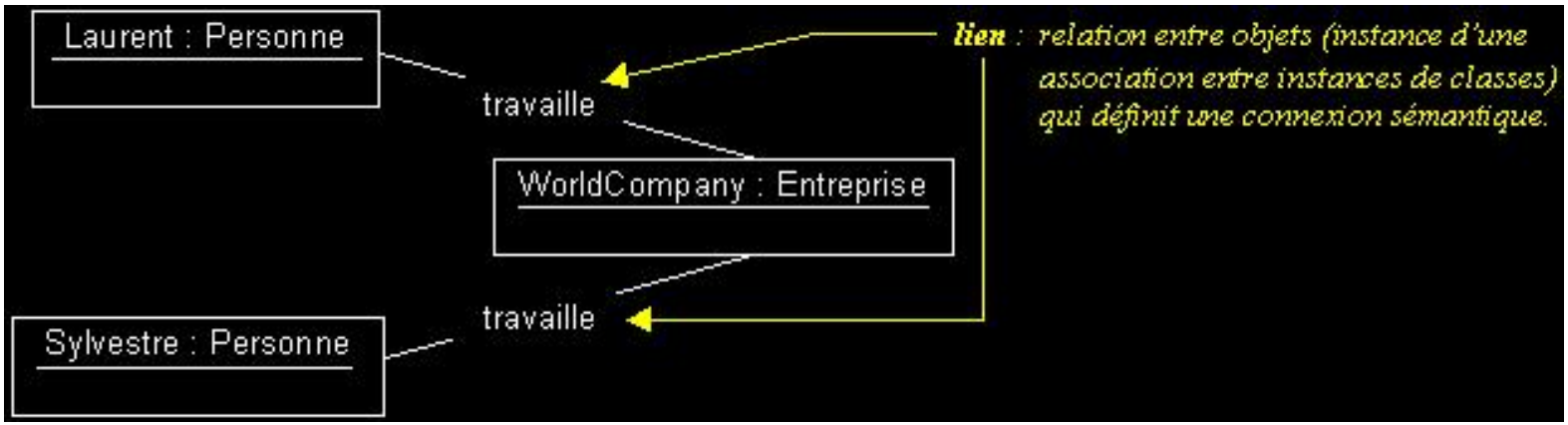


Différence entre classe et objet

On définit les classes `Personne` et `Entreprise` et leur relation (ici une association) `travaille`



On instancie deux objets `Personne` (Laurent et Sylvestre) et un objet `Entreprise` (WorldCompany) ainsi que le lien entre ces objets :



Si on instancie un nouvel objet `Entreprise` (Alstom), les objets `Personne` Laurent et Sylvestre, n'auront pas de lien avec l'objet Alstom.

Généralités sur les opérations

On distingue 3 sortes d'opérateurs :

- les **constructeurs** qui construisent physiquement un objet du type
- les **accesseurs** qui permettent la restitution d'une partie d'une valeur de la classe

```
( selectionner(t,i) -> e)
```

- les **transformateurs** qui transforment une valeur de la classe (qui modifient l'état de l'objet)

```
(inserer(t,e,i))
```

On distingue aussi les opérations :

- **publiques**, accessibles à un client du type (signe +)
- **privées**, cachées à tout utilisateur du type (signe -)

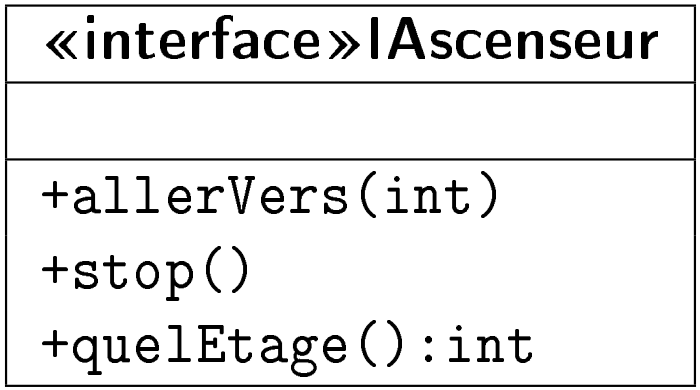
Exemple : modélisation d'un ascenseur

- Dans le monde réel, un ascenseur peut-être décrit par
  - Ses caractéristiques : poids max, étage inférieur et supérieur, matériaux et technologie utilisés, ...
  - Ses fonctionnalités : aller d'un étage à l'autre, s'arrêter, indiquer l'étage courant, ...

- La modélisation consiste à s'abstraire de certaines de ses caractéristiques et fonctionnalités de manière à concevoir un objet idéal qui réponde au cahier des charges de l'application.

  La modélisation revient à choisir les attributs et les opérations pertinentes (méthodes) par rapport aux besoins de l'application

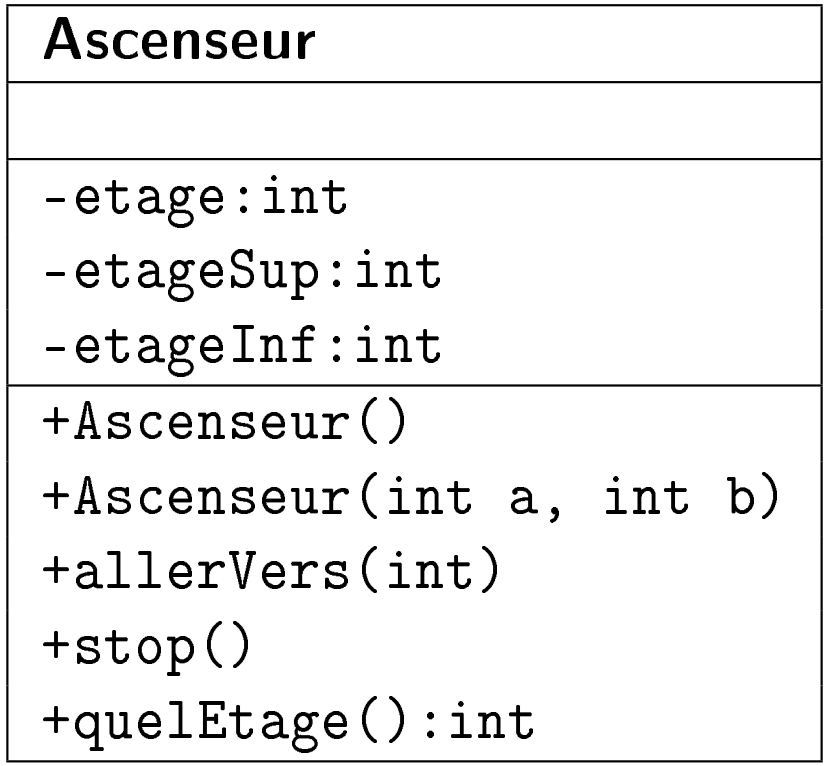
Représentation UML de l'interface `IAscenseur`



L'interface IAscenseur en Java

```
interface IAscenseur{
/**
 * sémantique
 *   déplace l'ascenseur vers l'étage i
 * @param i l'étage d'arrivée
 * préconditions
 * allerVers(a,i) : etageInf(a) <= i <= etageSup(a)
 */
public void allerVers(int i);
/**
 * sémantique
 *   arrête l'ascenseur
 */
public void stop();
/**
 * sémantique
 *   retourne le num de l'étage courant
 * @return le num de l'étage
 */
public int quelEtage();
}
```

Représentation UML de la classe `Ascenseur`



Programmation de la classe Ascenseur en Java

```
public class Ascenseur implements IAscenseur{
  private int etage = 0;
  private int etageInf = 0;
  private int etageSup = 8;

  public Ascenseur(){}

  public Ascenseur(int a,int b){
    if (a<b) {
      etageInf = a;
      etageSup = b;}
    else //erreur }

  public void allerVers(int v){
    if (etageInf <= v  v <= etageSup)
      etage = v;
    else //erreur}

  public void stop(){}

  public int quelEtage(){
    return etage; }
}
```

Exemples de Classes

Classe Rectangle

```
public class Rectangle{
  private double largeur = 10.0;
  private double hauteur = 5.0;

  public Rectangle(){}

  public Rectangle(int l, int h){
    largeur = l;
    hauteur = h;
  }

  public double surface(){
    return largeur*hauteur;
  }

  public double perimetre(){
   return 2*(largeur+hauteur);
  }
  ...
}
```

Classe Compte

```
public class Compte{
  private int solde = 0;

  public Compte(){}

  public Compte(int solde){
    this.solde = solde;
  }

  public void crediter(int n){
    this.solde = this.solde + n;
  }

  public void debiter(int n){
    this.solde = this.solde - n;
  }

  public int combien(){
    return solde;
  }
}
```

Remarque : `this` est l'objet sur lequel l'opération est effectuée.

Classe Complexe

```
public class Complexe{
  private float r = 0.0F;
  private float i = 0.0F;

  public Complexe(){}

  public Complexe(float r, float i){
    this.r = r;
    this.i = i;
  }

  public void add(Complexe c){
     this.i = this.i+c.i;
     this.r = this.r+c.r;
  }

  public void sub(Complexe c){
     this.i = this.i-c.i;
     this.r = this.r-c.r;
  }

 ...
}
```

La création d'objets

Création d'objets

- Un objet (rectangle, compte bancaire ou nombre complexe) est crée (**instancié**) à partir d'une classe.
- Chaque classe détient donc un "plan de construction" pour des objets. Une classe décrit une famille d'objets qui possèdent **la même structure et le même comportement**.
- Toute classe décrit l'ensemble des **variables d'instance** (identificateur, type et éventuellement valeur initiale) et des **méthodes** (signature et programmation).

Mécanisme de création d'un objet

La création d'un objet se fait en 2 étapes :

- **Construction** : c'est à dire allocation de mémoire pour l'objet par l'opérateur new appliqué au constructeur de la classe.

```
new Rectangle();
new Compte();
new Complexe();
```

- **Initialisation** : par appel d'un constructeur. Les paramètres du constructeur sont les initialisateurs de variables d'instance de l'objet.Dans le cas où aucun paramètre n'est spécifié une valeur par défaut est fournie par le langage (Java, C++).

```
new Rectangle(0.0,0.0);
new Compte(700);
new Complexe(3.0F,1.25F);
```

Mécanisme de désignation d'un objet

L'objet crée, pour être manipulé dans un programme, doit porté un nom (**identificateur**).

- **Déclaration** d'une variable :

```
Rectangle unRectangle;
Compte leMien;
Complexe c;
```

- **Association** de la variable à l'objet :

```
unRectangle = new Rectangle(3.0,2.0);
leMien = new Compte(700);
c = new Complexe(5.0F,1.25F);
```

Copie d'un objet

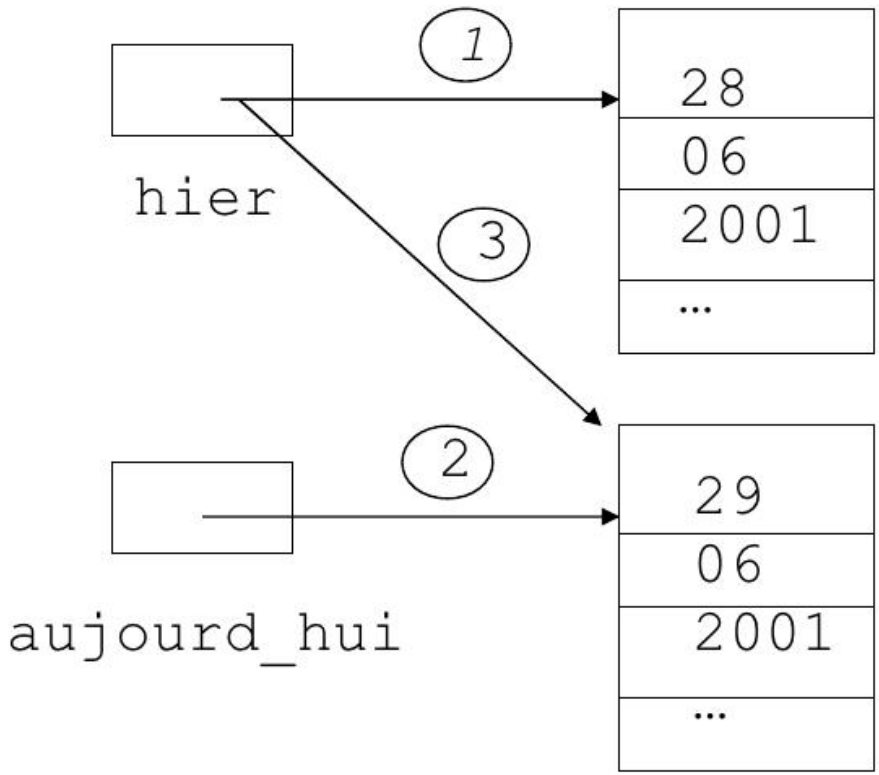
Les variables Java contiennent des **références** à des objets et **non les objets eux-mêmes**.

Lorsque l'on réalise une copie (par affectation), on copie en fait les références d'objets.

```
Jour hier = new Jour(28,06,2001); // 1

Jour aujourd_hui = new Jour(29,06,2001); // 2

hier = aujourd_hui; // 3
```

Représentation graphique

```
Jour hier = new Jour(28,06,2001); // 1

Jour aujourd_hui = new Jour(29,06,2001); // 2

hier = aujourd_hui; // 3
```



Clonage d'un objet

- La copie effective d'objets est réalisée par **clonage**.

  On utilise la méthode `clone()` de la classe `Object`.

```
Jour demain = new Jour(30,06,2001);

hier = (Jour)demain.clone();
```

- La méthode `clone()` retourne un objet de la classe `Object`, une conversion explicite de type est, par conséquent, requise.

Appel de méthode

- Les objets **communiquent** entre eux par **envoi de messages**.

  Les **méthodes** d'un objet correspondent aux messages qu'on peut lui envoyer.

- L'objet récepteur peut voir son état modifié à l'issue de l'exécution d'une méthode.

Structure d'un message

```
recepteur.méthode(arguments)
```

Exemples :

```
Compte leMien = new Compte(700);
leMien.debiter(500);
// l'état de l'objet leMien a changé : le solde vaut 200

Complexe c = new Complexe(5.0F,1.25F);
c.add(2.F,2.F);
// l'état de l'objet c a changé : r vaut 5.F et i vaut 3.25F

Rectangle unRectangle = new Rectangle(3.0,2.0);
unRectangle.surface();
// le message retourne la valeur 6.0
```

Appel de méthodes

- Les objets **communiquent** entre eux par **envoi de messages**.

  Un objet émetteur demande à un objet récepteur de réaliser **un traitement** (une opération).

- L'objet récepteur peut voir son état modifié à l'issue de l'exécution du message.

Exemple :

```
public class Moteur {
  private boolean estDemarre;

  public void demarre() {
    estDemarre=true;
  }
}
```

```
public class Voiture {
  private Moteur moteur;

  public void demarre() {
    moteur.demarre();
  }
}
```

Appel de méthode (1/2)

- **Evaluation des paramètres effectifs** de la méthode invoquée
- **Appel par valeur** : la valeur de chaque paramètre est copiée dans le paramètre formel correspondant :
  - Si les valeurs des arguments sont des **références** (cas des tableaux et des objets), ce sont celles-ci qui sont **copiées et non les objets référencés**.
  - Si les valeurs sont d'un **type primitif** (`int`, `double`, `char`, `boolean`, `...`), **la valeur de l'argument est copiée**.
- Le **récepteur** du message est un **paramètre implicite**. C'est un objet dont la référence est affectée à la pseudo-variable `this`.

Appel de méthode (2/2)

- Les instructions qui forment le corps de la méthode sont exécutées. La structure de contrôle de base est maintenant l'envoi de message.
- L'exécution de la méthode se termine par l'instruction `return` ou bien par la dernière instruction (cas d'une méthode qui retourne une valeur du type `void`).
- La valeur retournée remplace le message. Dans le cas d'une méthode `void`, l'instruction qui suit l'envoi du message est exécutée.

Les membres de classes

- Visibilité des membres de classes
- La pseudo variable `this`
- Constantes

Visibilité des membres de classes

- Les modules logiciels déclarés `public` sont utilisables par n'importe quelle classe.
- Les membres `private` ne sont accessibles que dans la classe qui les définit.

Visibilité : Exemple

```
public class Principale {

  public static void main(String[] args){
    int a;
    Pile p= new Pile();
    a =5;
    p.haut = 1;
    p.empiler(a);
    p.pile[2] = a;
  }
}
```

```
public class Pile {
  private int[] pile;
  private int haut;

  public Pile(){
    pile = new int[10];
    haut = 0;
  }
  public empiler(int a){
    pile[haut]=a;
    haut++;
  }
}
```

Questions :

- Distinguez les objets et les méthodes publiques et privées.
- Dans le module contenant la classe `Principale`, indiquez quelles instructions sont autorisées.

Protection des variables d'instance

- Il est de bon usage de déclarer les variables d'instances `private` et non `public`.
- La bonne approche est de fournir des **méthodes d'accès** à ces variables d'instance et de les garder **cachées**.
- Les méthodes d'accès sont alors les seules portes d'accès à ces variables.
- La correction, le déboggage et la maintenance du programme s'en trouvent améliorés.

Exercice : Résultat du programme ?

```
public class Complexe{
 private float r = 0.0F;
 private float i = 0.0F;

 public Complexe(float r, float i){
   this.r = r;
   this.i = i;}

 public void add(Complexe c){
    this.i = this.i+c.i;
    this.r = this.r+c.r;}
}

public class Test{
 public static void main(String[] args)
 {
 Complexe x = new Complexe(3,4);
 Complexe y = new Complexe(2,1);
 x.add(y);
 System.out.print("partie réelle de x = "+ x.r);
 System.out.print("partie réelle de y = "+ x.i);
 x.i=90.1f;
 }
}
```

Exercice : Résultat du programme ?

```
public class Test{
  public static void main(String[] args){
  Complexe x = new Complexe(3,4);
  Complexe y = new Complexe(2,1);
  x.add(y);
  System.out.print("partie réelle de x = "+ x.r);
  System.out.print("partie réelle de y = "+ x.i);
  x.i=90.1f; }
}
```

Ce code produit une erreur car les variables d'instances sont privées.

Exercice : Résultat du programme ?

On ajoute les méthodes suivantes à la classe `Complexe`

```
public float getR(){ // accesseur (getter)
  return r;  }

public float getI(){ // accesseur (getter)
  return i; }

public void setR(float x){ //transformateur (setter)
  this.r=x; }

public void setI(float x){ //transformateur (setter)
  this.i=x; }
```

On ré-écrit la fonction principale :

```
public class Test{
  public static void main(String[] args){
  Complexe x = new Complexe(3,4);
  Complexe y = new Complexe(2,1);
  x.add(y);
  System.out.print("partie réelle de x = "+ x.getR());
  System.out.print("partie réelle de y = "+ x.getI());
  x.i=90.1f; }
}
```

Remarque sur les constructeurs

Il est possible de déclarer **plusieurs constructeurs** pour une même classe.

Exemple :

```
public class Jour{
  int leJour;
  int leMois;
  int l_an;

  public Jour(){}

  public Jour(int j,int m,int a){
    leJour = j;
    leMois = m;
    l_an = a; }

  public Jour(int m,int a){
    leMois = m;
    l_an = a; }
}
```

La référence d'objet this

this désigne l'objet recepteur de la méthode.

Exemple :

```
public class Jour{
  int j;
  int m;
  int a;

  public Jour(){}

  public Jour(int j,int m,int a){
    this.j = j;
    this.m = m;
    this.a = a; }

  public Jour(int m,int a){
    this.m = m;
    this.a = a; }
}
```

Autre utilisation de this

```
public class Jour{
  int j;
  int m;
  int a;

 public Jour(int j){
    this(j,12,2022);
  }

 public Jour(int m, int a){
    this.m = m;
    this.a = a;
    this.j = 23;
  }

 public Jour(int j,int m, int a){
    this.j = j;
    this.m = m;
    this.a = a;
  }
  ...
}
```

Déclaration de constante

- Lorsqu'une variable d'instance représente une valeur constante pour la classe, on la déclarera `final` qui signifie que sa valeur ne peut pas être modifiée

- **Exemples :**

```
final double PI = 3.14159265358;
final double TABLE[] = {1.28,3.32,8.97,5.65};
final String MSG = "OK";
```

- Traditionnellement, en Java, une constante est notée en majuscules