Programmation par objets

1 Introduction

La plupart des logiciels sont programmés à l'aide de langages dits «à objets», notamment Java et C++. Les raisons de ce succès sont principalement économiques : les logiciels ainsi conçus sont plus fiables, plus faciles à maintenir et à faire évoluer.

Ce qui caractérise les langages à objets sont, par ordre d'importance :

Favoriser la structuration des logiciels selon les types de données manipulées, plutôt que sur les traitements. Le programmeur se pose en premier la question : quelles sont les types de choses qui interviennent dans l'application ? Il décrit alors chacun de ces types d'objet au moyen d'une ***classe*** qui amalgame des données pour représenter l'état de ce type d'objets et des procédures qui réalisent les opérations possibles sur ces objets. Par exemple, pour gérer un réseau ferroviaire, le programmeur définira les types d'objets train, voiture, ligne, gare... Pour chaque type d'objet, il indiquera les propriétés de ces objets et les actions que l'on peut leur faire subir : un train est formé d'une collection de voitures, on peut y accrocher ou décrocher une voiture, chaque voiture a une capacité... Les logiciels ainsi conçus sont plus faciles à maintenir et à faire évoluer car les types de données manipulées sont beaucoup plus stables que les traitements invoqués dans une application. Les traitements peuvent changer selon l'évolution des besoins, alors que les types de données demeurent les mêmes et sont donc réutilisables dans les versions successives ou les extension du logiciel.

Garantir une totale séparation entre les spécifications d'un type de données et sa mise-en-œuvre. C'est la notion d'***encapsulation***. L'utilisateur d'une classe n'a accès qu'aux procédures publiques de la classe, ce qui garantie le respect des spécifications en empêchant toute manipulation incongrue des données concrètes qui servent à réaliser ces spécifications.

Manipuler de façon homogène des données de types similaires mais cependant disparates : l'***héritage*** permet, à partir d'une classe de base, de définir des types plus précis. Par exemple, une application de bibliothèque gère des «documents», représentés par une classe **`Document`**, classe de base, qui peuvent être des livres, des revues, des dictionnaires... chacune des variétés sera représentée par une classe dérivée de la classe de base : **`Livre`**, **`Revue`**, **`Dictionnaire`**... En plus d'établir une hiérarchie de précision entre classes, l'héritage s'accompagne de la notion dynamique de ***procédure virtuelle***. Une telle procédure, déclarée dans la classe de base, peut avoir une version différente dans chaque classe dérivée. C'est en cours d'exécution que la procédure convenable est appelée selon le type précis de l'objet auquel elle s'applique. Ce mécanisme s'appelle la ***liaison dynamique***.

Afin de donner un aperçu précis de ce style de programmation, nous l'illustrerons par le langage Java. Java présente de nombreux avantages pour une première approche : il offre tous les aspects des langages à objets et reste relativement simple. C++ est également un langage remarquable, autant par les concepts supplémentaires qu'il introduit que par les performances des programmes qu'il permet de réaliser, mais il est d'une complexité bien plus grande et nécessite une grande technicité.

2 Eléments de syntaxe

2.1 Spécification d'une classe

L'exemple suivant est la définition d'une classe qui modélise des objets «décompteurs». L'encadré suivant montre la ***spécification*** de cette classe, c'est-à-dire ce qui est visible pour les utilisateurs de cette classe. Sa fonctionnalité est indiquée par les commentaires de spécification accompagnant chaque procédure.

```
class Decompteur {

  static final int initParDefaut=100;

  Decompteur(int v)
  // prérequis : v>=0
  // constructeur : décompteur initialisé à v

  Decompteur()
  // constructeur : décompteur initialisé à initParDéfaut

  void decremente()
  // effet : décrémente le décompteur s'il n'est pas nul

  boolean estNul()
  // résultat : indique si le décompteur est nul

  static Decompteur max (Decompteur d1, Decompteur d2)
  // résultat : le plus grand décompteur parmi d1 et d2
 }
```

Un objet de ce type possède un état changeant qui est un entier positif ou nul. L'initialisation lors de la création d'un objet est réalisée par un ***constructeur***. Ici il y a deux constructeurs : le premier constructeur **`Decompteur(int v)`** reçoit en paramètre l'état initial, le second **`Decompteur()`**initialise le décompteur à la valeur **`initParDefaut`**.

Les choses ***statiques*** sont les choses indépendantes de tout objet. Leur définition est accompagnée du vocable **`static`**.

- Une ***donnée statique*** existe dès le début de l'exécution, en un seul exemplaire, même si aucun objet n'a été créé. C'est le cas ici de **`initParDefaut`**, constante qui vaut 100 (le vocable **`final`** indique que cette donnée est constante, ne pourra être modifiée en cours d'exécution).
- Une ***procédure statique*** est une procédure qui reçoit tous ses paramètres par les parenthèses. Elle n'a pas d'objet distingué auquel elle s'applique. C'est le cas ici de la fonction **`max`**. Ses seuls paramètres sont deux décompteurs **`d1`** et **`d2`** indiqués dans les parenthèses.

Les choses ***non statiques*** sont les choses propres à chaque objet.

- Une ***donnée non statique*** est une donnée qui accompagne chaque objet et qui caractérise son état. Dans cet exemple, il n'y a pas de données non statiques visibles par les utilisateurs de la classe (il y a une donnée non statique, un entier i, mais inaccessible par l'extérieur).
- Une ***procédure non statique***, encore appelée ***méthode d'objet***, est une procédure qui s'applique à un objet de la classe appelé ***instance courante***. L'instance courante est un paramètre qui n'est pas explicitement indiqué dans les parenthèses. Dans la programmation d'une méthode d'objet, on peut citer explicitement l'instance courante par le vocable **`this`**. Dans l'exemple, il y a deux méthodes d'objet : **`decremente()`** qui décrémente **`this`** s'il n'est pas nul et **`estNul()`** qui permet de tester si **`this`** est nul.

Autrement dit, un composant non statique a un **`this`** alors qu'un composant statique n'en a pas.

2.2 Utilisation d'une classe modèle d'objets

2.2.1 Notion de référence - création d'objets

Les objets sont désignables par une ***référence***. La référence à un objet lui confère une ***identité*** qui le distingue des autres. Ainsi deux objets peuvent être dans le même état et cependant ne pas être le même. Concrètement, un objet est quelque part en mémoire et sa référence est son adresse dans la mémoire.

Pour manipuler un objet, on peut être amené à lui donner un nom au moyen d'une ***déclaration*** :

```
Decompteur echeance1;
```

En Java, une telle déclaration ne crée pas d'objet. La chose ainsi déclarée, **`echeance1`**, est une référence qui ne désigne rien, la référence **`null`** (ce n'est pas pareil en C++, une déclaration similaire crée effectivement un objet : C++ est plus sophistiqué que Java, mais aussi plus complexe).

La ***création*** d'un objet, en Java, se fait au fil de l'exécution, au moyen d'une expression de création qui sollicite un constructeur :

```
new Decompteur(12)
```

Cette expression crée un décompteur initialisé à 12 et rend en résultat la référence à l'objet créé. Cette référence peut alors être captée par un nom déclaré afin d'utiliser l'objet :

```
echeance1 = new Decompteur(12);
```

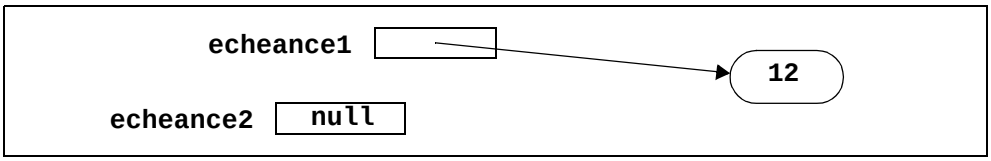
Parfois, la déclaration s'accompagne d'une création :

```
Decompteur echeance1 = new Decompteur(12);
```

Les noms déclarés de type classe sont toujours des références. Ainsi, plusieurs noms peuvent désigner le même objet. Exemple :

```
Decompteur echeance2;
```

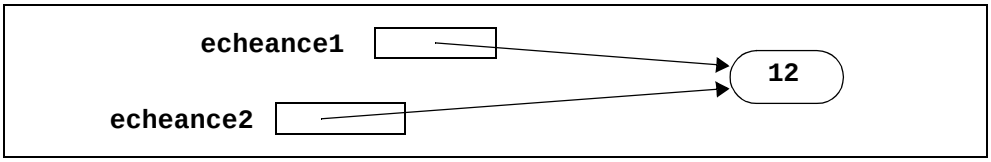
La situation peut être illustrée par le dessin suivant :



Après exécution de l'instruction :

```
echeance2 = echeance1;
```

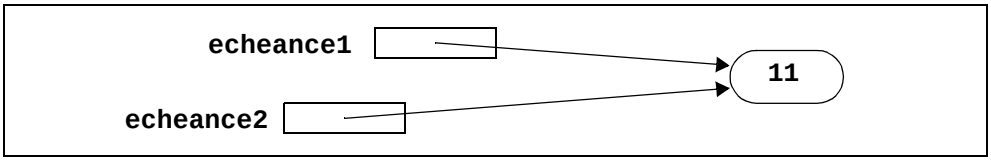
La situation devient :



`echeance1` et **`echeance2`** désignent le ***même*** objet décompteur, et ça se voit : si par exemple on utilise le nom **`echeance1`** décrémenter le décompteur :

```
echeance1.decremente();
```

Le décompteur désigné par **`echeance2`** se trouve décrémenté, puisque c'est le même :



2.2.2 Appel des procédures et accès aux données d'une classe

Depuis le texte d'un programme qui utilise une classe, l'accès aux composants de la classe est différent selon qu'il s'agit d'un composant statique ou non statique.

Pour ***accéder à un composant statique***, la syntaxe est :

classe.composant

Exemples :
`Decompteur.initParDefaut` est la valeur d'initialisation par défaut des décompteurs
```
Decompteur echeance3 = new Decompteur(8);
Decompteur x = Decompteur.max(echeance1, echeance3);
```
`x` désigne alors le décompteur qui a plus grand état parmi **`echeance1`** et **`echeance3`**, c'est-à-dire **`echeance1`**. Si on décrémente **`x`**, on décrémente **`echeance1`**, car c'est la référence au plus grand qui est rendue par la procédure **`max`**.

Pour ***accéder à un composant non statique***, la syntaxe est :

objet.composant

La classe du composant sollicité est déterminée par la classe de l'objet. Exemples :
`echeance2.decremente();` décrémente le décompteur désigné par **`echeance2`**
`echeance1.estNul()` teste si **`echeance1`** est nul
`echeance1` et **`echeance2`** sont ici les instances courantes, paramètres sur lesquels s'appliquent respectivement les procédures **`decremente`** et **`estNul`**.

Remarque : pour une procédure manipulant un objet de la classe, le choix d'offrir une procédure statique ou non statique est souvent une simple question de «style» qui n'a aucune conséquence sémantique. Ce n'est qu'à travers l'héritage et les procédures virtuelles que le choix a une importance sémantique, en distinguant le paramètre particulier **`this`** pour servir de support à la liaison dynamique.
Ici nous avons choisi d'offrir la fonction **`max`** sous forme d'une procédure statique. Nous l'avons fait par raison de symétrie : pourquoi choisir **`d1`** plutôt que **`d2`** comme instance courante ? Nous aurions pu programmer cette fonction en tant que méthode d'objet, la différence étant simplement syntaxique dans la définition et l'utilisation de cette fonction :

```
class Decompteur {
 ...
  Decompteur max (Decompteur d2)
  // résultat : le plus grand décompteur parmi this et d2,
}

...

Decompteur x = échéance1.max(echeance3);
```

2.3 Rédaction d'une classe

2.3.1 Encapsulation : private / public

Une classe représente une abstraction. Les objets modélisés doivent avoir le comportement spécifié, «ni plus, ni moins». L'état d'un objet va cependant devoir être représenté par des données «concrètes». Dans l'exemple du décompteur, cette donnée est une variable entière (**`int i;`**). Mais seules les choses spécifiées doivent être accessibles par les programmes qui utilisent cette classe. Cela est indiqué par les vocables **`public`** pour les choses offertes et **`private`** pour les choses innacessibles. C'est ce qu'on appelle l'***encapsulation***. un des principaux aspects de la programmation «par objets», et curieusement c'est un aspect qui n'a rien à voir avec les «objets». C'est une notion qui

concerne les règles de portés des noms : les choses privées sont visibles depuis l'intérieur de la classe et invisibles depuis les textes extérieurs à la classe.

Généralement, les données qui servent à représenter l'état des objets sont privées. Elles ne sont manipulées que depuis les procédures qui réalisent la fonctionalité de la classe.

```
class Decompteur {

  public static final int initParDefaut=100;

  private int i; // état du décompteur

  public Decompteur(int v) {
  // prérequis : v>=0
  // constructeur : décompteur initialisé à v
    i=v;
  }

  public Decompteur() {
  // constructeur : décompteur initialisé à initParDéfaut
    i=initParDefaut;
  }

  public void decremente() {
  // effet : décrémente this s'il n'est pas nul
    if (i!=0) {i--;}
  }

  public boolean estNul() {
  // résultat : indique si this est nul
    return(i==0);
  }

  public static Decompteur max (Decompteur d1, Decompteur d2){
  // résultat : le plus grand des décompteurs parmi d1 et d2,
    if (d1.i>=d2.i) {retur d1;} else {return d2;}
  }
}
```

Ici par exemple, l'innaccssibilité directe de la variable **`i`** assure que l'etat d'un décompteur ne sera jamais négatif, une propriété voulue et qui a sans doute son importance...

3 Différents rôles des classes

Les classes peuvent servir à de nombreux usages. D'ailleurs dans un langage comme Java, toute chose doit être programmée dans une classe. On peut essayer de discerner les divers usages suivants :

3.1 Simple regroupement de fonctionnalités

On regroupe dans une classe les procédures et données concernant un même thème. Par exemple, dans la bibliothèque Java, la classe **`Math`** regroupe les fonctions de calcul usuelles :

`Math.sin`, **`Math.cos`**... ainsi que les constantes classiques : **`Math.PI`**, **`Math.E`**...

Il n'est ici aucunement question «d'objet». On ne crée pas d'objet **`Math`**, «**`new Math()`**» serait une absurdité.

3.2 Modélisation d'objet

Il s'agit dans ce cas de «vrais» objets. Une telle classe est destinée à être instanciée. Souvent ces objets ont un état changeant. Un exemple simple est la classe **`Personne`** qui représente des personnes avec les informations concernant leur état civil :

```
class Personne {

  private String nom; private String preNom;
  private String adresse;

  public Personne(String n, String pn, String adr) {
    nom=n; preNom=pn; adresse=adr;
  }

  public String getNom() {return nom;}

  public String getPreNom() {return preNom;}

  public String getAdresse() {return adresse;}

  public void setAdresse(String adr){adresse=adr;}
}
```

Dans cet exemple, le nom et le prénom d'une personne ne peut être modifié; alors que l'adresse peut l'être. Ceci est réalisé par ce qu'on appelle des ***accesseurs*** : accesseurs en consultation (**`getNom`**, **`getPrenom`**, **`getAdresse`**) et accesseur en modification (**`setAdresse`**).

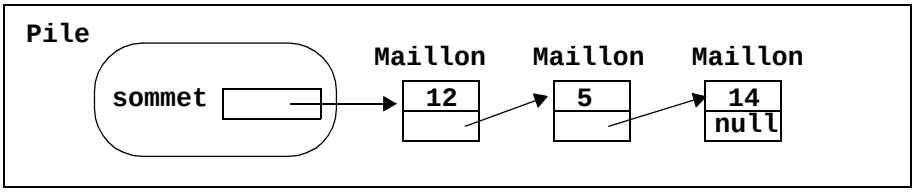
Dans cette catégorie, on trouve notamment les ***structures de données classiques*** qui réalisent diverses collections changeantes d'objets de types variés. Les bibliothèques offrent ainsi des ***piles*** (**`Stack`**), des ***files d'attente*** (**`File`**), des ***listes*** (**`List`**), des ***ensembles*** (**`Set`**), des ***tables de correspondance*** (**`Table`** ou **`Map`**), des ***vecteurs*** de taille changeante en cours d'exécution (**`Vector`**)... L'exemple suivant est une classe qui représente une pile d'éléments de type quelconque **`T`** :

Spécification :

```
class Pile<T>{

  Pile() // pile vide

  void empiler(T e)
  // effet : this = empilé(this,e)

  void depiler()
  // prérequis : this n'est pas vide
  // effet : this=dépilé(this)

  T sommet()
  // prérequis : this n'est pas vide
  // résultat : sommet(this)
}
```

L'état d'un objet de type **`Pile`** est une suite finie de valeur de type **`T`**. La méthode **`empiler`** ajoute un élément, la méthode **`depiler`** retire le plus récent élément non encore retiré et **`sommet`** rend en résultat le plus récent élément empilé non encore retiré.

Cette spécification peut être réalisée de nombreuses façons. Ici nous illustrons une réalisation à base de maillons chaînes. Cette méthode utilise une classe auxiliaire, **`Maillon`**. Un maillon est un couple constitué d'une élément de type **`T`** et de la référence à un autre maillon. Chaque maillon est associée à un élément de la pile. Sont champ **`element`** contient l'élément et le champ **`suivant`** est la référence au maillon associé à l'élément suivant de la pile. Le schéma suivant illustre une pile de nombres entiers qui contient la suite de valeur <12,5,14>. La donnée **`sommet`** de la classe **`Pile`** référence le maillon associé au sommet de la pile. Lorsque la pile est vide, **`sommet`** vaut **`null`** et il n'y a aucun maillon.



Voici cette programmation de la classe **`Pile`** :

```
class Pile<T>{

  private static class Maillon<Te>{
    Te element; Maillon<Te> suivant;
    Maillon(Te e, Maillon<Te> s){element=e; suivant=s;}
  }

  private Maillon<T> sommet;

  public Pile() {
     sommet=null;
  }

  public void empiler(T e){
     sommet=new Maillon<T>(e,sommet);
  }

  void depiler(){
     sommet=sommet.suivant;
  }

  T sommet(){
     return sommet.element;
  }
}
```

3.3 Réalisation de domaines de valeurs

Les langages de programmation offrent quelques types de valeurs classiques et plus ou moins directement pris en charge par les processeurs matériels : nombres entiers de taille limitée (**`int`**, **`long`**); approximation de nombres réels (**`float`**, **`double`**), caractères (**`char`**), valeurs logiques (**`boolean`**). Mais les domaines de valeurs plus sophistiqués doivent être programmés au moyen de classes. Un exemple simple est la représentation des nombres complexes.

```
class Complexe{

  private double x; private double y;

  private Complexe(double x, double y){
    this.x=x; this.y=y;
  }

  public static Complexe cartesien(double x, double y){
  // résultat : x + y.i, complexe de partie réelle x
  // et de partie imaginaire y
    return new Complexe(x,y);
  }

  public static Complexe polaire(double r, double a){
  // résultat : r.e^^ia, complexe de rayon r et d'angle a
    return new Complexe(r*Math.cos(a),r*Math.sin(a));
  }

  public Complexe add(Complexe z){
  // résultat : somme de this et de z
    return new Complexe(x+z.x, y+z.y);

  public String mul(Complexe z){
  // résultat : produit de this et de z
    return new Complexe(x*z.x-y*z.y, x*z.y+y*z.x);
  }
  public String toString(){ // résultat : this en clair
    return ""+x+"+"+y+"i";
  }

  public double getX(){ // résultat : partie réelle de this
    return x;
  }
  ...
}
```

Dans cet exemple nous avons choisi de programmer un ***constructeur privé***. Il y au moins deux raisons à cela :

- Cette classe représente un domaine de valeurs, l'ensemble des nombres complexes. Nous répugnons à «créer» des complexes. L'expression «**`new Complexe(...)`**» frise l'absurdité, un peut comme si pour parler de l'entier 12 on écrivait «**`new 12`**». Les éléments d'un domaine de valeurs ne sont pas des objets au sens normal du mot, on ne les crée pas, on les dénote. Ici, pour dénoter les complexes nous avons des ***fabricateurs***, les fonctions statiques **`cartesien`** et **`polaire`**. Les «**`new`**» sont tous cachés à l'intérieur des fonctions de la classe, fabricateurs et opérations à résultat **`Complexe`**.
- Rendre le constructeur public ne permettrait pas les deux formes de dénotation des nombres complexes, la forme cartésienne et la forme polaire. On peut certes offrir plusieurs constructeurs, mais ils doivent être distingués par le nombre ou le type des paramètres. Or ici les paramètres sont dans les deux cas deux nombres réels, la distinction serait donc impossible.

Lorsqu'une classe qui réalise un domaine de valeurs, les notions d'identité et de référence n'ont pas de sens pour ces «faux objets». Il n'y a pas lieu de distinguer entre deux nombres complexes égaux

à «3+2×*i*», de même qu'il n'y a pas lieu de distinguer entre deux «12», contrairement à deux «vrais objets» tels que des personnes qui peut avoir même nom et même prénom et cependant ne pas être la même. Dans cette catégorie, on peut citer les classes suivantes :

`String` : représentation des chaînes de caractères.

`Rationnel` : représentation des nombres rationnels. Ils permettent le calcul exact tant qu'on ne pratique que les quatre opération d'addition, de soustraction, de multiplication et de division.

`GrandEntier` (alias **`BigInteger`**) : nombres entiers non bornés. Ils permettent de manipuler des nombres entiers qui peuvent s'exprimer avec des milliers de chiffres. Ils sont utiles notamment en cryptographie, pour coder et décoder les messages confidentiels.

4 Héritage, méthodes virtuelles, classes abstraites

4.1 Héritage

L'héritage permet de définir des types «plus précis» à partir d'un type plus «vague». La classe vague est appelée ***classe de base***, la classe plus précise est dite ***classe dérivée***, ou ***héritière***. Un exemple simple peut être illustré par les types «personnes» (classe de base) et «salariés» (classe dérivée). Un salarié est également une personne. Il a toutes les propriétés d'une personne, un nom et un certain nombre d'enfants, et des propriétés supplémentaires, un salaire et une prime annuelle :

La notation **`class Salarie extends Personne`** indique que la classe **`Salarie`** hérite de la classe **`Personne`**.

```
class Personne {
  public String nom;
  public int nombreEnfants;
  public Personne(String n) {nom=n; nombreEnfants=0;}
}

class Salarie extends Personne {
  public int salaire;
  public Salarie (String n, int s) {super(n); salaire=s;}
  public int prime() {return salaire*nombreEnfants/10;}
}
```

Lorsque un objet **`Salarie`** et créé, il faut d'abord l'initialiser en tant qu'objet **`Personne`**. Pour cela il faut appeler un constructeur de **`Personne`** en lui passant les paramètres. En Java ceci est fait au début du constructeur de la classe dérivée grâce à l'instruction **`super(`***paramètres***`)`**.

4.2 Règle de compatibilité de types induite par l'héritage

Un objet de type **`Salarie`** est également un objet de type **`Personne`**, on peut donc l'utiliser partout où on peut utiliser un objet **`Personne`**. De façon générale, partout où on peut utiliser un objet d'une classe **`C`**, on peut également utiliser un objet de classe dérivée de **`C`** : en partie gauche d'affectation d'une référence déclarée de type **`C`**, en paramètre de procédure de type **`C`**...

En revanche on ne peut utiliser un objet de type **`Personne`** là ou on attend un objet de type **`Salarie`**, car les propriétés supplémentaires de **`Salarie`** sont absentes dans la classe de base.

4.3 Méthodes virtuelles

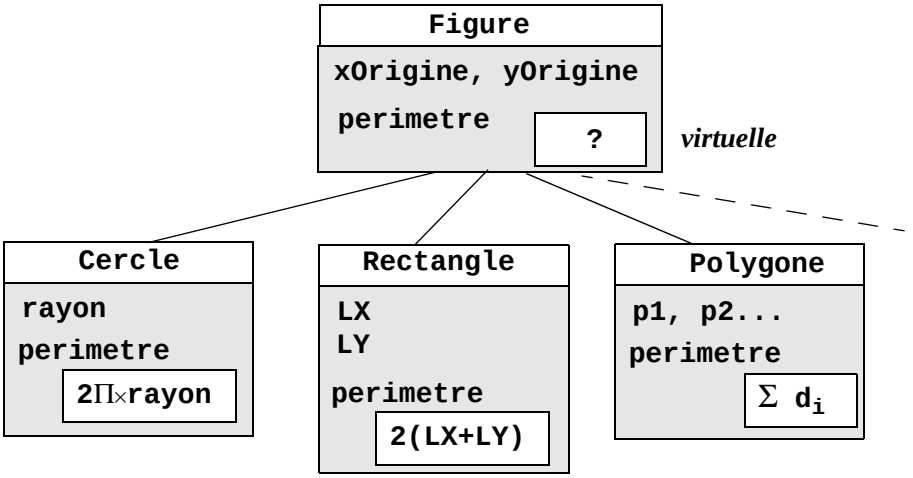
Les méthodes virtuelles permettent de prévoir des opérations similaires sur des objets d'espèces différentes. Une telle méthodes est destinée à être définie ou redéfinie dans chaque classe dérivée.

Une classe peut annoncer une méthode sans donner sa programmation. Une telle méthodes est dite ***abstraite*** (**`abstract`**). Une classe qui possède des méthodes abstraite est également qualifiée d'abstraite.

```
abstract class A {
  abstract void P();
  void Q() {...}
}

class B extends A {
  void P() {...}
  void Q() {...}
}
```

Dans l'exemple ci-dessus, la classe **A** annonce mais ne programme pas la méthode **P**. C'est une classe abstraite. La classe dérivée **B** définit **P** et redéfinit **Q**. Une classe abstraite étant incomplète, il est interdit de créer des objets de ce type.

Comme exemple simple, on peut considérer diverses catégories de figures fermées : des cercles, des rectangles, des polygones. Toutes ces variétés d'objets possèdent la notion de périmètre. Cependant la réalisation de l'opération est différente selon la catégorie. Pour cela, on indique, dans la classe **`Figure`**, que la fonction **`perimetre`** est abstraite.



La classe de base **`Figure`** est ***abstraite***, et on ne peut pas créer d'objets de cette classe. Sa seule raison d'être est de regrouper les diverses variétés de figures. On peut considérer l'analogie suivante : les «mammifères» sont un concept qui regroupe des espèces précises : vaches, chèvres... dans la nature on ne voit jamais un «mammifère», mais seulement des vaches, des chèvres... Les mammifères sont une classe abstraite avec un fonctionnalité commune, les femelles allaitent leurs petits (procédure virtuelle), qui a une réalisation particulière selon la classe précise.

```
abstract class Figure {
  double xOrigine; double yOrigine;
  Figure(double x, double y) {xOrigine=x; yOrigine=y;}
  abstract double perimetre();
}

class Cercle extends Figure {
  double rayon;
  Cercle(double xCentre, double yCentre, double r) {
  super(xCentre, yCentre); rayon=r;
  }
  double perimetre() {return 2*Math.PI*rayon;}
}

class Rectangle extends Figure {
  double largeur; double hauteur;
  Rectangle(double xCoin, double xCoin, double lx, double ly) {
    super(xCoin, yCoin); largeur=lx; hauteur=ly;
  }
  double perimetre() {return 2*(largeur+hauteur);}
}

class Polygone extends Figure {
  ...
}
```

4.4 Liaison dynamique des méthodes virtuelles

Lorsque la méthode **`perimetre()`** est appelée sur un objet référencé en tant que **`Figure`**, c'est ***la méthode du type précis de l'objet***, **`Cercle`**, **`Rectangle`** ou **`Polygone`**, qui est exécutée, bien que la désignation de l'objet soit de type vague. Le compilateur n'a pas pu décider quelle méthode appeler, c'est au moment de l'exécution que le choix de la bonne procédure est réalisé. Ce mécanisme s'appelle la ***liaison dynamique***. Exemple :

```
Cercle c = new Cercle(4,10,12);
Rectangle r = new Rectangle(4,10,12);
...
Figure f;
...
f = c; System.out.print(f.perimetre()); // périmètre de cercle
f = r; System.out.print(f.perimetre()); // périmètre de rectangle
```

La méthode adaptée au type précis de l'objet est appelée grâce à une information mémorisée dans l'objet. Cette information n'est pas très coûteuse : c'est généralement un pointeur sur une description de la classe de l'objet.

Les méthodes virtuelles sont un puissant outil de généralisation et d'extensibilité. Un algorithme **A** manipule des objets, disons de classe **C**, en exigeant que ces objets possèdent une certaine fonctionnalité concrétisée par une procédure virtuelle **p**. Pour utiliser cet algorithme **A** pour des objets de classe **D** ***nouvellement inventée***, un utilisateur n'aura qu'à déclarer que **D** hérite de **A** et programmer convenablement la procédure **p**. Un exemple classique de ceci est le tri de collections (listes,

tableaux...). Un algorithme de tri n'a pas besoin de connaître le type précis des objets à trier, il a seulement besoin de comparer les éléments qu'il tri. Pour cela il travaille sur des objets qu'on peut appeler «comparable», dotés d'une méthode virtuelle de comparaison censée réaliser une relation d'ordre (ça, le compilateur ne peut pas le vérifier...) :

```
abstract class Comparable {
  public abstract boolean superieur(Comparable unAutre);
  // résultat : indique si this est supérieur à unAutre
}

class Algo { // quelques algorithmes utiles
  static void trier(Comparable[] aTrier){
  // effet : trie le tableau aTrier par ordre croissant
    ...
    if(aTrier[i].superieur(aTrier[j])...
    ...
  }
}
```

Pour trier un tableau d'objets de type **`Personne`** selon un certain critère, par exemple selon leur année de naissance, il suffit que la classe **`Personne`** hérite de **`Comparable`** et programme ce critère de comparaison dans la méthode **`superieur`** :

```
class Personne extends Comparable {
  String nom; int anneeDeNaissance; int poids;
  Personne(String n, int a){nom=n; annéeDeNaissance=a;}
  public boolean superieur{Personne p){
    return anneeDeNaissance>p.anneeDeNaissance;
  }
}
```

4.5 Cas limite de classe abstraite : interfaces

Un cas limite important d'usage de procédures virtuelle est celui de classes «totalement abstraites» : elles ne possèdent aucune données et seulement des méthodes abstraites. En Java, on les appelle des ***interfaces*** (**`interface`** au lieu de **`class`**), et l'héritage est rebaptisé «mise en œuvre» (**`implements`** au lieu de **`extends`**). En effet, quant on «hérite» d'une interface, on hérite en quelque sorte de dettes, les méthodes qu'il faut programmer.

Une interface joue le rôle de cahier des charges que l'utilisateur doit respecter pour que ses objets puissent participer à certains traitements. C'est aussi un moyen élégant de passer des fonctions en paramètre de fonctions, ce qu'on appelle généralement «***programmation d'ordre supérieur***». Pour exemple nous pouvons compléter l'exemple précédent. Si nous voulons trier des personnes selon plusieurs critères, la technique précédente, qui consiste à programmer la fonction de comparaison en tant que méthode des objets à trier ne suffit pas, puis qu'on ne peut pas programmer plusieurs fois la méthode **`superieur`**. Dans ce cas, il faut ***passer le comparateur en paramètre supplémentaire*** :

```
interface Comparateur<T> {
  public boolean superieur(T lUn, T lAutre);
  // résultat : indique si lUn est supérieur à lAutre
}

class Algo { // quelques algorithmes utiles
  static void trier(Vector<T> aTrier, Comparateur comp){
  // effet : trie le vecteur aTrier selon l'ordre comp
    ...
    if(comp.sperieur(aTrier.at(i),aTrier.at(j))...
    ...
  }
}
```

Pour utiliser la procédure **`Algo.trier`**, il faut maintenant fabriquer le comparateur souhaité en programmant une mise-en-œuvre de l'interface **`Comparateur<Personne>`**, par exemple pour comparer les personnes selon leur poids, et passer une instance de ce comparateur en paramètre du tri :

```
class CompareSelonPoids extends Comparateur<Personne> {
  public boolean superieur{Personne p1, Personne p2){
    return p1.poids>p2.poids;
  }
}

...

Vector<Personne> population;
...
Algo.trier(population, new CompareSelonPoids());
...
```

L'usage de classes satisfaisant une interface pour pratiquer une programmation d'ordre supérieur est un moyen plus général que le simple passage de fonction en paramètre.L'objet que l'on passe en paramètre peut avoir des attributs de données qui constituent un ***contexte***. Ce contexte, initialisé grâce aux paramètres de construction de l'objet, permet ainsi de choisir au moment de l'exécution la fonction parmi une «famille de fonctions». Par exemple, soit à trier une population selon la proximité de son poids à un poids connu en cours d'exécution, il suffit de passer en paramètre une instance de l'***objet fonctionnel*** suivant :

```
class PoidsProcheDe extends Comparateur<Personne>{
  private int poidsCible;
  public PoidsProcheDe(int cePoids){poidsCible=cePoids;}
  public boolean superieur{Personne p1, Personne p2){
    return Math.abs(p1.année>p2.poids;
  }
}
...
Algo.trier(population, new CompareSelonPoids(lePoidsVoulu));
...
```

Encore plus général : si le contexte de l'objet fonctionnel est une variable qui change à chaque appel, on arrive à passer ainsi une machine à état en paramètre, par exemple pour générer des séquences de valeurs.

D'un point de vue plus «grand public», cette technique est largement utilisée pour programmer les interfaces homme-machine des applications fenétrées. Dans un tel système, les «widgets» que sont les «boutons ok», les menus déroulants... génèrent des événements qui doivent déclencher des traitements. Pour cela, le concepteur de l'application doit programmer des «***écouteurs d'évènements***» (**`Listener`**) qui sont des objets fonctionnels qu'il passe en paramètre au bouton qui génère l'événement à traiter. C'est ainsi qu'on parvient à connecter des traitements spécifiques et nouvellement inventés au système de fenétrage préexistant.

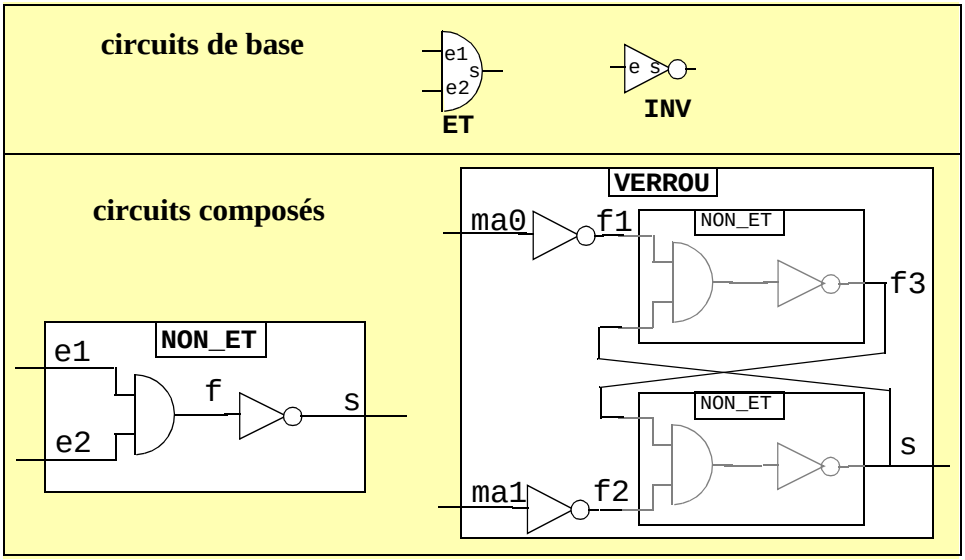
5 Exemple récapitulatif : simulateur de circuits logiques

L'exemple suivant utilise, à faible dose, toutes les notions vues précédemment, de manière presque «caricaturale». Il s'agit de la programmation d'un simulateur de circuits logiques (simple). Voici une description rapide de ce qu'on cherche à simuler :

Les circuits logiques sont soit des ***circuits de base*** (par exemple inverseur **`INV`**, porte et **`ET`**) soit des ***circuits composés***, résultant d'un assemblage d'autres circuits.

Les composants d'un circuit composé sont reliés par des ***fils*** : le circuit composé **`NON_ET`** utilise le fil **`f`** pour connecter la sortie d'une instance de **`ET`** à l'entrée d'une instance de **`INV`**.

Les ***broches***, fils d'entrée et de sortie d'un modèle de circuit, permettent de connecter une instance de circuit à des fils effectifs qui le relient à d'autres circuits.



Un fil est porteur d'une valeur logique 0, 1 ou X, X représentant la valeur indéterminée. Ces valeurs seront représentées par les caractères **`'0'`**, **`'1'`** et **`'X'`**. Initialement, les fils portent la valeur indéterminée **`'X'`**.

Une analyse du problème a conduit aux remarques suivantes :

- Les circuits et les fils ont des caractéristiques qui en font tout naturellement des objets.
- Les broches d'un circuit désignent les fils qui relient ce circuit à d'autres circuits. Cette désignation peut avantageusement être installée lors de la création du circuit, au moyen de références aux fils d'interconnexion passés en paramètre du constructeur.
- Le simulateur doit avoir accès à la liste de tous les circuits de base et à la liste de tous les fils. Il est judicieux de faire de ces listes des membres statiques dans lesquelles chaque objet, circuit de base ou fil, se rajoute lors de sa création.

5.1 Principe de fonctionnement du simulateur

Le but du simulateur est de faire fonctionner les circuits par instants successifs, unité de temps simulé. Chaque unité de temps est décomposé en deux phases :

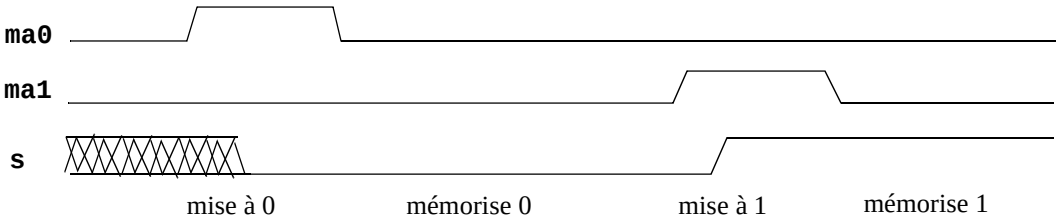
Première phase

Chaque circuit de base possède une méthode d'évaluation, **`evalue()`**, qui définit son comportement. Le simulateur appelle les méthodes **`evalue()`** de tous les circuits de base, ce qui a pour effet d'attribuer une ***valeur future*** aux fils de sortie en fonction des ***valeurs courantes*** portées par les fils d'entrée (ceci ne modifie pas la valeur courante du fil).

Deuxième phase

Chaque fil possède une méthode appelée **`actualise()`** qui met à jour sa valeur courante au moyen de sa valeur future. Le simulateur appelle, pour chaque fil, sa méthode **`actualise()`**.

Le programme proposé simule un verrou set-reset («latch» en Anglais). C'est une mémoire élémentaire de 1 bit avec deux entrées : **`ma0`** (mise à zéro) et **`ma1`** (mise à 1). Il est réalisé au moyen de deux portes **`NON_ET`** connectés en boucle de rétroaction. Son comportement est illustré par le chronogramme suivant :



5.2 Programme

Cette programmation comporte la classe **`Fil`**, la classe abstraite **`CircuitDeBase`** et ses classes héritées **`INV`** et **`ET`**, les classe qui représentent les circuits composés **`NON_ET`** et **`VERROU`**, et pour terminer la boucle du simulateur qui donne vie au total.

```
class Fil {
  private char EtatCourant; private char EtatFutur;
  public static Vector<Fil> tous = new Vector<Fil>();
  public Fil() {EtatCourant='X'; tous.add(this);}
  public char lire() {return EtatCourant;}
  public void ecrire(char v) {EtatFutur=v;}
  public void actualise() {EtatCourant=EtatFutur;}
}
```

```
abstract class CircuitDeBase {
// liste de tous les circuits de base :
  public static Vector<CircuitDeBase> tous = new Vector<CircuitDeBase>();
  public CircuitDeBase() {tous.add(this);}
  abstract public void evalue(); // procédure virtuelle d'évaluation
}

class ET extends CircuitDeBase {
  private Fil e1; Fil e2; Fil s;
  public ET (Fil ee1, Fil ee2, Fil st) {e1=ee1; e2=ee2; s=st;}
  public void evalue() {
    if (e1.lire()=='0'||e2.lire()=='0') {s.ecrire('0');}
    else if(e1.lire()=='1' && e2.lire()=='1') {s.ecrire('1');}
    else {s.ecrire('X');}
  }
}

class INV extends CircuitDeBase {
  private Fil e; Fil s;
  public INV (Fil ee, Fil st) {e=ee; s=st;};
  public void evalue() {
    if (e.lire()=='0') {s.ecrire('1');}
    else if(e.lire()=='1') {s.ecrire('0');}
    else {s.ecrire('X');}
  }
}
```

Les classe de circuits composés sont de «purs constructeurs». Leur rôle est simplement de créer les composants, les fils d'interconnexion et de «câbler» tout ça...

```
class NON_ET {
  public NON_ET(Fil e1, Fil e2, Fil s) {
    ET et = new ET(e1,e2,f); Fil f=new Fil();
    INV inv = new INV(f,s);
  }
}


class VERROU { // verrou SetReset
  public VERROU(Fil ma1, Fil ma0, Fil s) {
    Fil f1=new Fil(); Fil f2=new Fil(); Fil f3=new Fil();
    INV inv1=new INV(ma1,f1); NON_ET ne1=new NON_ET(f1,s,f3);
    INV inv2=new INV(ma0,f2); NON_ET ne2=new NON_ET(f3,f2,s);
  }
}
```

La boucle du simulateur simule l'exemple illustré sue le chronogramme précédent et affiche le résultat de simulation sous la forme d'une succession de triplets de valeurs :

```
ma0  ma1  s
```

```
public class Simul {
  public static void main(String argv[]) {
    Fil ma0=new Fil(); Fil ma1=new Fil(); Fil s=new Fil();
    VERROU v = new VERROU(ma0,ma1,s);

    String MA0 = "00000000000011111000000000000000000000000000";
    String MA1 = "00000000000000000000000000111111000000000000";

    System.out.print("\nRsultats de simulation\n   ma0 ma1  s\n");

    for(int k=0; k<MA0.length(); k++) {

        ma0.ecrire(MA0.charAt(k)); ma1.ecrire(MA1.charAt(k));

     for(int i=0; i<CircuitDeBase.tous.size(); i++) {
       CircuitDeBase.tous.elementAt(i).evalue();
     }

    for(int i=0; i<Fil.tous.size(); i++) {
        Fil.tous.elementAt(i).actualise();
     }
    System.out.print(
      "    "+ma0.lire()+"   "+ma1.lire()+"   "+s.lire()+"\n");
     }
  }
}
```