www.centraliup.fr.st

Modélisation Objet

(Support Java)

Georges Linares

Modélisation Objet

(Support Java)

Table des matières

Introduction
Principes de la programmation orientée objet 3
Classes et Objets 4
Héritage simple, Héritage multiple 5
Typage & Polymorphisme 7
Introduction à Java 9

Version 1.0 ◆ Publiée le 30/12/2002

Principes de la programmation orientée objet

La programmation procédurale

Dans la programmation traditionnelle – dite **programmation procédurale** – on décompose hiérarchiquement le problème posé en sous-problèmes plus faciles, eux-mêmes décomposés jusqu'à n'obtenir que des problèmes élémentaires.

Exemple : *Diagonalisation d'une matrice en programmation procédurale*

- *Recherche d'un pivot (Comparer des réels)*
- *Combinaison des lignes (Combiner des coefficients)*
- *Combinaison des colonnes (Combiner des coefficients)*

Dans la programmation procédurale, les données et les traitements sont toujours séparés : l'ensemble des **structures de données** et des **algorithmes** forme le programme final. On s'en doute, une telle distinction risque de poser quelques problèmes.

En effet, les solutions développées grâce à la méthode procédurale sont nécessairement liées au problème posé, donc l'intérêt de ces solutions se limite au contexte précis du problème. D'où les inconvénients suivants :

- Une faible réutilisabilité et faible généricité, qui rendent la solution peu rentable, car aucun fragment du programme ne peut servir pour un autre (on parle de 'programme jetable').
- Les différents segments du programme sont dépendants les uns des autres.
- Plus le problème est complexe, plus la conception est difficile, car la démarche de décomposition est peu naturelle.

Finalement, l'approche procédurale est trop bas niveau, et pas assez abstraite pour résoudre le problème dans sa globalité, d'où une démarche peu adaptée.

L'approche 'objets'

L'approche 'objets' offre un autre angle d'analyse et de résolution du problème posé. Plutôt que de résoudre le problème directement, on va d'abord modéliser l'univers du problème : les acteurs, les relations, etc. dans une sorte de simulation.

Les formes intermédiaires abstraites qui découlent de la **programmation orientée objet** sont universelles, peu liées au problème, et donc sont **stables** pour des problèmes différents car elles sont **réutilisables**, **évolutives** (par enrichissement du modèle) et plus **faciles à tester** et à maintenir.

Exemple : *Programme de calcul du meilleur chemin pour un voyageur de commerce (optimisation). Différentes méthodes de programmation…*

Procédural :

- *Données : carte, villes, routes.*
- *Algorithmes de recherche..*

Objet :

- *Une ville est un objet qui a une population, un réseau routier, etc.*
- *Une route est un objet qui a une vitesse limite, un nombre de voies, un débit moyen, etc.*
- *Un graphe peut être créé, enrichi, parcouru.*
- *Un réseau routier est un graphe dont les arcs sont des routes.*

Classes et Objets

Objets

Un **objet** est une entité de l'univers du problème. Cette entité peut être plus ou moins concrète, tant qu'elle possède des limites conceptuelles.

Exemple : *La nationale 7 est un objet. Mais une fonction mathématique peut être un objet.*

On détermine un objet par :

- Son **état** : C'est l'ensemble de propriétés (caractéristiques statiques) et de valeurs associées (caractéristiques dynamiques, *i.e.* qui évoluent au cours de l'exécution du programme).

Exemple : *Objet Ma_voiture*

Propriétés	*Valeurs*
Nombre de places :	*5*
Puissance :	*7*

- Son **comportement** : C'est l'ensemble des méthodes rattachées à l'objet. Ces méthodes peuvent être des interactions avec d'autres objets du modèle, ou bien des opérations sur l'objet (création, destruction, accès, modification de l'état de l'objet par action sur les valeurs).
- Son **identité** : C'est une propriété unique, non valuée, qui identifie l'objet de façon unique.

Relations entre objets

Différentes relations peuvent exister entre objets. Voici les dénominations les plus courantes :

- **Acteurs** : utilisation d'autres objets.
- **Serveurs** : utilisation par un autre objet.
- **Agents** : Utilisations mutuelles.
- **Contenance** : Un objet en contient un autre.

Classes

Une **classe** est un ensemble d'objets qui partagent les mêmes propriétés (mais pas forcément les mêmes valeurs) et les mêmes comportements. Cet ensemble homogène est abstrait, dans le sens ou il n'existe pas vraiment au même titre que les objets. C'est seulement une description d'objets. De cette définition, il découle qu'un objet est un représentant ou une instance d'une classe.

Exemple : *Classe Maison :*

Propriétés : *Nombre de pièces :*
Surface :
Couleur :
Propriétaire :

Méthodes : *Vendre*
Repeindre
Agrandir

Objets et classes sont fondamentalement différents. Un objet est une entité concrète, alors qu'une classe n'est qu'une description. Techniquement, la définition des classes et de leurs relations est généralement préalable à l'exécution d'un programme (statique) ; alors que les objets sont créés et détruits tout au long du programme.

Mais une classe est un **méta-objet**, c'est-à-dire un objet abstrait qui en décrit d'autres.

Héritage simple, Héritage multiple

Relations entre classes

L'héritage est un moyen de structurer l'ensemble des objets de l'univers du problème. Les relations qui lient les différentes classes peuvent être :

- **Composition** : Une classe peut être composée d'autres classes (ex : *une voiture est composée d'un objet moteur, d'un objet châssis, etc.*).
- **Spécialisation** : Une classe décrit un type particulier d'éléments d'une autre classe : elle décrit un sous-ensemble de l'ensemble des objets de la super-classe.
- **Généralisation** : On peut généraliser les classes dérivées à une super-classe les regroupant.

Héritage simple

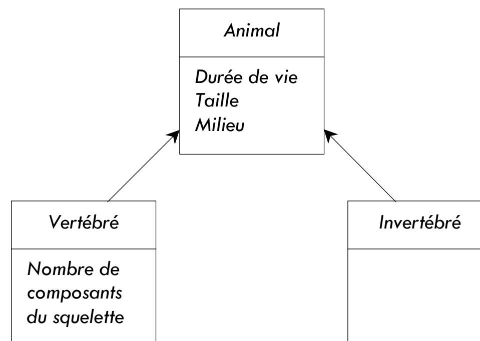
L'**héritage** est une relation de spécialisation. Une classe peut hériter d'une autre classe de deux façons :

- La description est enrichie : la classe dérivée hérite des propriétés et méthodes de la super-classe, ou elle est enrichit cette description de ses méthodes et propriétés propres.
- Par substitution : des méthodes spécifiques masquent celles de la super-classe.

Exemple : *Un article de luxe est décrit dans la classe article, mais le calcul du prix diffère par un taux de TVA différent, ce qui nécessite une nouvelle méthode qui masque celle de la classe article.*

L'héritage permet de structurer les classes en arbres de classes.

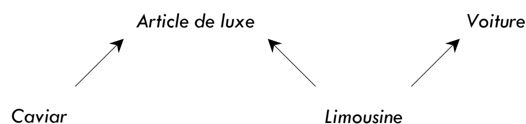
Exemple :



Héritage multiple

On peut très bien envisager qu'un objet appartienne à différentes catégories, étant un représentant de plusieurs classes. On parle alors d'**héritage multiple**. Cette méthode présente beaucoup d'intérêt, et se révèle techniquement très pratique et vraiment cohérente avec un modèle dit 'mental'.

Exemple :



Cependant, avec ce type de structure, l'arbre des classes n'est justement plus un arbre, mais un graphe orienté, sans circuits. En effet, pour un arbre, il faut que chaque nœud ne possède qu'un parent, ce qui n'est pas le cas d'un graphe. Cela rend le modèle moins hiérarchisé, moins clair, et moins lisible. C'est pourquoi cette notion est encore discutée, et n'est pas acceptée

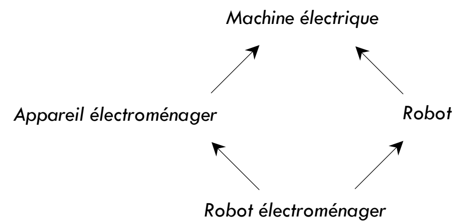
partout.

Plus grave, accorder trop d'importance au point de vue fonctionnel (anticiper sur les contraintes de programmation, vouloir par exemple 'mettre en facteur' le maximum de code) peut aboutir à un mauvais modèle, c'est-à-dire un modèle qui n'a aucun rapport à la réalité logique.

Héritage répété

Un **héritage répété** se produit lorsqu'une classe hérite d'une autre classe par différents chemins. Ceci pose quelques problèmes : les attributs hérités sont-ils dupliqués ? Comment accéder à ces différentes occurrences d'attributs ? Il n'y a pas de réponse univoque à ces questions, mais dans les langages C++ et Java, le compilateur ordonne les ancêtres et n'insère que le premier en cas d'héritage répété.

Exemple :



En bref

Un bon modèle, c'est avant tout de bonnes classes. Pour garantir l'élaboration de classes de bonne qualité, il faut :

- Bien analyser le problème et distinguer les entités de l'univers ;
- Définir précisément les classes ;
- Veiller à la qualité des liens entre les classes, en limitant les dépendances au maximum (pour une meilleure réutilisabilité) ; de plus, le modèle doit être concret, logique, et intelligible, pour un bon graphe d'héritage.

L'utilisation de la programmation orientée objet nécessite un investissement initial plus important que pour la programmation procédurale. Mais cet investissement est récupéré dans une large mesure par la réutilisabilité des objets dans d'autres programmes.

Typage & Polymorphisme

Typage

Le **typage** dans les langages procéduraux est caractérisé par les élément suivants :

- **Catégorisation des données** : sémantique et opérateurs valides,
- **Codage associé** : espace mémoire et représentation de l'information.

Exemple :

```
int i;
```

- *Catégorie : entier naturel (signé), opérations arithmétiques classiques.*
- *Codage : Complément à deux, 4 octets.*

Langages plus ou moins typés

Dans les langages **fortement typés**, comme le C, le Pascal, le C++, ou le Java, les données sont obligatoirement typées. Elles ne peuvent pas changer de type. Cette méthode a pour avantage la sécurité et l'efficacité, mais par contre le système est rigide et très bas niveau.

Dans les langages traditionnels, on peut palier à cette rigidité par des alternatives au typage dynamique :

- En Pascal : recours à un champ variable,
- En C, les unions (voir *Algorithmique & Programmation Support C++, Première Année*).

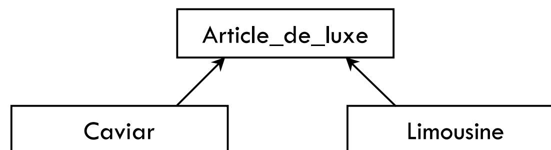
Dans les langages **faiblement typés**, les données ne sont pas toujours typées. Par exemple, dans le *Lisp*, la procédure de exo(x,y) permet à x et y de prendre des valeurs quelconques de type spécifié à l'appel de la procédure. *Smalltalk* est un autre exemple de langage objet faiblement typé.

Langages objet

Les langages objet présentent des caractéristiques liées au typage qui les distinguent des autres langages :

- les classes sont des types ;
- les types classes sont liés entre eux par une organisation hiérarchique.
- Il peut y avoir plusieurs types pour un même objet, d'où un affaiblissement du typage.

Considérons l'exemple suivant :



Dans cet exemple de conceptualisation extrêmement poussée, une limousine est un article de luxe, donc *ma_rolls* peut être typé de deux façons différentes : c'est un objet de type *Limousine*, mais c'est également un objet de type *Article_de_Luxe*.

Néanmoins, des problèmes se posent si on utilise l'écriture suivante :

```
Limousine ma_rolls;
Article_De_Luxe art;
Art=ma_rolls;
```

Cette portion de code est cohérente mais malheureusement elle se révèle syntaxiquement incorrecte en liaison statique. D'autre part, la méthode *démarrer()* a-t-elle un sens pour *Article_De_Luxe* ? (Pour que cela fonctionne, il faudrait *caster* explicitement *art* en objet de type *Limousine*) Et la méthode *prixTTC()* a-t-elle le même sens pour un article de luxe et une limousine ?

Polymorphisme

Les liaisons statiques mettent en jeu un typage à la compilation, tandis que les liaisons dynamiques sont le fait de typage qui ont lieu pendant l'exécution. Les méthodes alors utilisées sont celles de l'objet concret (final), elles sont donc sélectionnées dynamiquement.

C'est précisément ce qu'on appelle le **polymorphisme**.

Le polymorphisme n'est pas une surcharge ! En effet, la surcharge est une méthode qui en masque une autre. Elle découle de la possibilité de certains langages évolués de nommer de la même façon plusieurs méthodes de signature différente (la signature est composée des types des arguments de la fonction, et du type renvoyé).

Par définition, le polymorphisme est la sélection dynamique des méthodes en fonction du type de l'objet sélectionné.

Il permet de gagner grandement en cohérence du modèle, et améliore dans une large mesure la généricité des algorithmes. En effet, le polymorphisme permet la manipulation des données au niveau d'abstraction maximal. Et plus les données sont abstraites, plus l'algorithme est générique.

Exemple d'un algorithme de tri :

Dans la version traditionnelle, on élaborera un code pour chaque type de donnée, …mais on aura au final qu'un seul algorithme.

Si on considère qu'un objet « triable » est un objet pour lequel on dispose d'une relation d'ordre (il suffit de pouvoir comparer deux objets pour les trier : logique, mais encore fallait-il le remarquer), on peut mettre en œuvre un code commun utilisant uniquement des fonctions de comparaisons spécifiques au type de données traitées. On peut même à l'extrême trier différentes données de type différent dans un même ensemble.

Mécanisme

Le polymorphisme met néanmoins en action un mécanisme sous-jacent relativement complexe, qui doit stocker le type concret de l'objet pointé, en plus de l'adresse, et proposer pour chaque fonction des tables de fonctions virtuelles, en fonction de chacun des types référencés. Ce mécanisme est entièrement masqué à l'utilisateur, et c'est précisément ce qui fait tout son intérêt dans le cadre de la programmation objet.

En Java, aucun problème, le mécanisme est implicite :

```
class document
{
     Public void afficher() {System.out.println("document");}
};
class dessin extends document {public void afficher(){...}};
class texte extends document {public void afficher(){...}};

document doc[]=new document[2];
doc[0]=new html():
doc[1]=new texte();
doc[0].afficher();  // Méthode d'affichage d'un html
doc[1].afficher();  // Méthode d'affichage d'un texte
```

Par contre, en C++, il n'y a pas de polymorphisme implicite :

```
void document::afficher(){cout<<"Salut";}
void html::afficher(){...}
void texte::afficher(){...}
document * doc[2];
doc[0]=new html();
doc[1]=new texte();
doc[0]->afficher();     // Affiche "Salut"
doc[1]->afficher();     // Affiche aussi "Salut"
```

Pour utiliser le polymorphisme, il faut définir des méthodes virtuelles, qui activeront le *dynamic binding* (sélection dynamique des méthodes). Pour cela, on ajoute le mot clé *virtual*.

```
virtual void document::afficher(){cout<<"Salut";}
```

Mais prudence ! Le polymorphisme consomme des ressources mémoire et diminue l'efficacité. On voit donc que l'intérêt fonctionnel se paie. Cher. Très cher. C'est pourquoi le C++ permet de le désactiver, et ne l'implémente d'ailleurs pas par défaut, contrairement au Java.

On peut avoir recours aux **méthodes virtuelles pures** pour interdire l'instanciation de la super-classe, qui devient du même coup une class abstraite. Pour cela, il faut utiliser la syntaxe suivante :

```
virtual void document::afficher()=0;
```

Introduction à Java

Historique

Le premier langage objet est apparu très tôt, en 1967 : *Simula 67*. Le nouveau concept n'était alors pas encore adapté aux problèmes de l'époque et il a été mis de côté. *Smalltalk 80* a été développé par *Apple* en 1980 et était un langage purement objet, dont s'est inspiré *Objective* C, une extension du C. Puis est apparu le langage objet le plus utilisé, le C++, cependant assez discuté car il n'impose pas au programmeur les contraintes d'un langage purement objet.

L'ancêtre du Java est *OAK* développé par *Sun* en 1991. L'idée était de créer un langage complètement indépendant de la machine. Le premier Java date de 1994, toujours par *Sun*. Dès lors, la vie de Java était indissociablement liée à celle des navigateurs Internet. Plus récemment, en 2000, la nouvelle mouture Java 2 (version 1.3) s'est imposée sur le marché.

Motivations

Le langage Java vise essentiellement la programmation Internet. Sa spécificité tient du fait que son code est mobile : il est possible de l'exécuter sur des machines distantes du serveur, et surtout des machines hétérogènes.

Cette approche a posé de nombreux problèmes, notamment au niveau de la portabilité et de la transportabilité totales (niveau hardware : sur des processeurs différents, niveau software : des systèmes d'exploitation différents). Il fallait aussi assurer la sécurité du contenu par rapport aux droits de la machine distante, et garantir un code suffisamment sûr pour le système d'exploitation distant. Enfin il fallait gérer des problèmes de gestion de mémoire distante, où les notions d'adresse ou de pointeurs n'avaient, dès lors, plus aucun sens.

Principe

Le Java est tout d'abord un langage objet. Il reprend la syntaxe du C++, tout en restant plus simple que celui-ci. Le code Java est indépendant de la plate-forme utilisée, et la gestion dynamique de la mémoire qui s'avérait un véritable casse-tête en C++ devient avec le Java totalement automatisée, et donc plus sûre.

Le problème de la portabilité vient surtout de la compilation. En effet, le programme compilé devient des segments de code machine, spécifiques au système sur lequel a eu lieu la compilation, alors que, bien sûr, le source est compilable sur toute plate-forme… L'idée est d'insérer une couche logicielle et de créer une Machine Virtuelle Java (JVM) logicielle, présentant la même interface sur toutes les machines physiques, mais masquant celles-ci. Le source Java est dès lors compilé en J-Code, lui-même interprété par une JVM.

Le J-Code est donc indépendant de l'environnement matériel et logiciel, alors que la JVM dépend du système. C'est donc le J-Code qui est portable et mobile !

La tendance actuelle est à la compilation partielle ou a la volée du J-Code, pour éviter l'interprétation systématique du J-Code par la JVM.

La compilation et l'exécution d'un programme Java se fait en plusieurs étapes :

- Ecriture du programme. Exemple : *ex1.java*
- Compilation en utilisant la commande *javac*. Exemple : *javac ex1*, qui produit un *ex1.class*, en J-Code.
- Exécution par l'invocation de java (la JVM). Exemple : *java ex1*

Structure

En Java, tout est classe, sauf les types numériques, booléens et caractères. Ecrire un programme Java, c'est donc définir et implémenter des classes. Il n'y a ni fonctions globales, ni variables globales. Par contre une classe peut contenir la fonction principale (de lancement) main. Une telle fonction peut ensuite être invoquée à partir d'une console.

Mon premier programme en Java :

```
import java.io.* // Importation du package io
class salut
{
    public static void main(string args[])
    {
        afficher();
        system.exit(0); // Sortie du programme
    }
    public void afficher() // Fonction Hello World !
    {
        system.out.println("Salut tout le monde !");
```

```
    }
}
```

Variables

En Java, toutes les variables sont typées. Elles sont initialisées dès leur création automatiquement à 0 et null pour les références. Une variable ne peut être définie que dans une classe : il n'y a pas de variables globales. Définition et initialisation : ***type nom*** = ***valeur***;

Il existe deux types de données fondamentaux :

- Types primitifs : ils sont manipulés par valeur ;
- Types références : ils sont manipulés par référence.

Leur codage est indépendant de la plate-forme, ce qui est bien sûr indispensable pour la portabilité car par exemple certains processeurs codent d'abord les octets de poids faibles, puis ceux de poids forts (*Intel*), alors que d'autres font l'inverse…

Les types primitifs du langage Java sont :

byte	8 bits
short	16 bits
int	32 bits
long	64 bits
float	32 bits IEEE 754
double	64 bits IEEE 754
char	16 bits Unicode
boolean	1 bit (true/false)

Les types références sont aussi appelés types complexes ou composés. Ce sont des 'tableaux' : des vecteurs ou des matrices. D'autres types références sont les classes ou les interfaces.

<u>Exemple :</u>

```
int tab[];            // Vecteur d'entiers
int[] tabaussi;       // Idem !
float matrice[][];    // Matrice de flottants
```

Les tailles ne sont pas spécifiées à la déclaration, elles le sont à l'aide de la méthode *new* :

```
int tab[]=new int[5];
float matrice[][]=new float[4][6];
char tabchar[][]=new char[10][];
tabchar[0]=new char[4];
```

L'utilisateur ne s'occupe pas de la libération de la mémoire.

Les variables finales sont des variables particulières dont la valeur ne peut être fixée qu'une fois. On utilise la syntaxe suivante :

```
final int T;
final float f=0.0;
//...
T=3;
```

Opérateurs

Les opérateurs de Java sont similaires à peu de choses près à ceux du C++ :

+	-	*	/	%	
+=	-=	*=	/=	%=	
>	<	<=	>=	==	!=
!	&	|	^	&&	||
!=	&=	|=	?=		
<<	>>	>>>	<<<	<<=	>>=

Structures de contrôle

La syntaxe des structure de test est la même qu'en C++ :

if(***condition***) {***instructions***} [else {***instructions***}]

<u>Exemple :</u>

```
if(I==0) {system.out.println("nul");}
else {system.out.println("non nul");}
```

De même pour la distribution de tests :

switch(***variable***)
{
    case ***valeur1*** :

```
        instructions
        case valeur2 :
        instructions
        …
        default :
        instructions
}
```

Quelques boucles sont bien entendu disponibles :

- `do` {*instructions*} `while`(*condition*)
- `while`(*condition*) `do` {*instructions*}
- `for`(*initialisation ; condition ; instruction*) {*instructions*}

Ainsi que les instructions préférées des professeurs :

- *break* : sortie forcée d'une structure (prière de ne pas l'utiliser hors des *switch*) ;
- *continue* : passe directement à l'itération de boucle suivante.

Classes et objets

La définition d'une classe est analogue à ce qui se fait en C++ :

`class` ***nom*** {***définitions***};

Une classe contient :

- Des définitions de variables d'instances,
- Des définitions de méthodes,
- Des variables de classe,
- Des méthodes de classe,
- D'autres définitions de classe (classes encapsulées ou *inner-class*).

Les variables d'instance sont créées lors de l'instanciation de la classe. La variable ainsi créée appartient à l'instance ou objet. Par exemple, si x et y sont des variables d'instance de la classe Point, chaque objet possède un couple de variables x,y qui lui est propre.

Les méthodes d'instance sont déclarées de façon classique, et invoquée à partir de leur instance : ***objet.méthode***. Leur signature (ce qui les caractérise de façon unique) est composée de leur type de retour, leur liste de paramètres, leur nom et leur classe.

Les variables de classe sont différentes des variables d'instance car leur durée de vie est indépendante des différentes instances de la classe. Une variable de classe est en effet partagée par toutes les instances, elle leur est commune. On les définit par `static` ***type nom***; et on y accède par ***nom_classe.nom_variable***.

Exemple :

```
class Pile
{
    point tab[];
    static public int taille_maxi;
    //...
}
pile.taille_maxi=3;
```

Les méthodes de classe peuvent être invoquées sans instance (comme la fonction *main*), ce qui permet d'encapsuler des méthodes sans créer d'instances de la classe. Elles sont définies par `static` ***méthode*** et utilisée en tapant ***nom_classe.nom_methode***(***liste_de_paramètres***);.

Exemple :

```
class Pile
{
    Point tab[];
    static void redimensionner(int t) {/*...*/}
    //...
}
Pile.redimensionner(3);
```

Une classe possède au moins un constructeur. Attention, comme ne l'indique pas son nom, le constructeur ne construit pas l'objet, mais l'initialise. Les constructeurs surchargés diffèrent par leur liste de paramètres et leur implémentation.

En général, le langage Java rend l'utilisation des destructeurs inutile. En effet, chaque objet possède un compteur de références, et un *Garbage Collector* (littéralement un ramasse-miettes) se charge de détruire périodiquement les objets non référencés. La destruction des objets inutilisés est donc automatique, mais bien sûr pas optimale. Dans ce cas, si l'on veut absolument spécifier

une destruction, on peut invoquer la méthode *finalize()*.

Les modalités de contrôle de l'accès aux membres d'une classe permettent de contrôler l'accès et l'utilisation des variables et des fonctions membres. Cela permet également de masquer une partie de l'implémentation, ce qui confère plus de liberté au développeur de la classe, et plus de sécurité aux utilisateurs. Ces différentes permissions ressemblent à celles des classes C++ :

- Variables et méthodes privées (*private*) : utilisation interne à la classe, mais cette protection n'est effective qu'au niveau de la classe, c'est-à-dire que n'importe quel objet de la classe peut accéder à tous les attributs des objets de même type ;
- Variables et méthodes publiques (*public*) : pour tous, choisi par défaut ;
- Variables et méthodes protégées (*protected*) : utilisation par les classes dérivées et par les classes du *package*. Un package est un ensemble de classes qui partagent leur accès.

<u>Exemple</u> :

```
class Pile
{
    protected Point tab[];
    public void empiler(Point p) {/*...*/};
    public Point depiler() {/*...*/};
    private void agrandir();
}
```

Le langage Java offre la possibilité d'encapsuler des classes à l'intérieur d'autres classes. Une classe encapsulée est une classe définie au sein même d'une autre classe. L'intérêt est de limiter la portée d'une définition de classe, et de permettre la définition de classe 'assistantes', c'est-à-dire des classes utilisées par d'autres classes, mais qui n'auraient aucun sens en dehors de leur contexte bien précis. Cette technique permet également d'éviter l'utilisation inadéquate de classes outils.

Voici une application adéquate de l'encapsulation de classes :

```
class Liste
{
    class Maillon
    {
        int info;
        Maillon suiv;
    }
    private Maillon debut;
    //...
}
```

On peut, de la même manière que pour les membres et les méthodes, autoriser ou interdire l'utilisation de la classe encapsulée par les mots clés *private*, *public*, *protected*, et *package*.

Le langage Java ne supporte pas l'héritage multiple direct. C'est-à-dire qu'une classe ne dériver que d'un seul parent, contrairement au C++.

Dans l'héritage simple, au niveau sémantique, la sous-classe spécialise sa super-classe, donc la super-classe généralise ses sous-classes. Une classe dérivée hérite des méthodes et des attributs de la super-classe. L'accès à ces attributs est réglementé par le statut des variables héritées :

- *public* : les membre hérités sont public et accessibles ;
- *private* : les attributs ne sont pas accessibles ;
- *protected* : les attributs ne sont accessibles qu'aux héritiers.

Syntaxe : `class` ***nom_classe_dérivée*** `extends` ***nom_super_classe***

Allocation de mémoire

L'opérateur *new* alloue de la mémoire sur demande de l'utilisateur. Son premier rôle est de créer la variable et de réserver l'espace mémoire nécessaire à son stockage. Si la mémoire est insuffisante, une exception de type *OutOfMemoryError* est générée. Il est à noter que l'allocation est implicite dans les constructeurs. Le second rôle de l'opérateur *new* est d'initialiser une variable.

<u>Exemple</u> :

```
int a=new int(2);
employe=new Personne("Durand");
```

L'opérateur *new* renvoie une référence. Même si l'allocation est dynamique, il n'est pas nécessaire de s'occuper de la libération de la mémoire, car celle-ci est automatique en Java.

Un objet est toujours construit et détruit. Le constructeur est invoqué par l'opérateur *new*.

Lorsque aucun constructeur n'est spécifié, le compilateur implémente un constructeur par défaut, sans argument, et initialisant automatiquement toutes les valeurs à 0. Dès qu'un constructeur, même avec arguments, est défini, le constructeur par défaut est supprimé. Donc soit on ne donne aucun constructeur, soit on spécifie tous les constructeurs !

<u>Exemple</u> :

```
class Point
{
    private int x,y;
    Point(int a,int b) {x=a; y=b;}
};
Point p=new Point(2,4); // Fonctionne !
Point q=new Point(); // Provoque une erreur...
```

Un objet d'une classe dérivée est une instance de ses ancêtres. Chacun des constructeurs des super-classes est invoqué implicitement ou explicitement dans les constructeurs de la classe dérivée. Donc ces constructeurs doivent nécessairement exister !

<u>Deux exemples d'invocation implicite</u> :

```
// 1)
class Compte{float solde;};
class Compte_rem extends Compte
{
    float taux;
    Compte_rem(){taux=5.5;} // Tout va bien
};

// 2)
class Compte
{
    float solde;
    Compte(float s){solde=s;}
};

class Compte_rem extends Compte
{
    float taux;
    Compte_rem(){taux=5.5;} // Erreur car le constructeur sans argument
};                          // n'est pas trouve !
```

Si on veut invoquer explicitement un constructeur, il faut utiliser les noms réservés *super* et *this*, qui appellent respectivement le constructeur de la super-classe et le constructeur de l'objet sur lequel on invoque la méthode :

```
class Compte_rem extends Compte
{
    float taux;
    Compte_rem(float t)
    {
        super(0.0);
        taux=t;
    }
    Compte_rem(){this(5.5);}
};
```

L'affectation

Pour les types primitifs (types de base), lorsque le contenu d'une variable est affecté à une autre, Java procède à une recopie par valeur. Par contre, les types référence (tableaux, types complexes, classes, etc.) sont recopiés par adresse, et ce, automatiquement.

<u>Exemple</u> :

```
// Recopie par valeur :
int a, b=4;
a=b;

// Recopie de l'adresse :
int t[], s[];
t=new int[3];
s=t;
int p2, p=new Point(2,3);
p2=p
```

Les exceptions

Pour illustrer l'intérêt de gérer les exceptions dans un programme Java, étudions le cas d'une fonction fort simple, qui calcule la valeur réelle d'une fraction, tout ce qu'il y a de plus banal…

```
class fraction
{
    int num, int dem;
    float frac2reel()
    {
```

```
        return num/(float)dem;
        // Et si dem==0 ?
    }
};
```

Cette fonction pose problème lorsque le dénominateur est égal à 0. En effet, on aimerait éviter l'erreur d'exécution, fatale au programme, et avertir l'utilisateur de sa méprise.

On pourrait envisager de renvoyer une valeur particulière, codant pour une erreur. Même si pour d'autres cas cela peut être mis en œuvre, ici, justement, on ne peut renvoyer un *float* qui signifierait qu'il y a eu une erreur, car toutes les valeurs sont significatives, c'est-à-dire que la fonction *frac2reel* peut potentiellement renvoyer n'importe quelle valeur réelle.

Une solution serait d'avoir un paramètre dédié, par exemple passé par référence à la fonction :

```
float frac2reel(integer code_erreur)
{
    if(dem==0.)
    {
        code_erreur=1;
        return INF;
    }
    else code_erreur=0;
    return num/(float)dem;
}
```

Le problème de cette technique est qu'il alourdit la programmation de façon assez conséquente (à chaque appel de la fonction, il faudra créer et initialiser un entier et l'envoyer à la fonction, puis le tester en sortie…). De plus, de manière plus générale, son coût est important dans la mesure où l'erreur en question reste exceptionnelle.

Une autre solution serait d'arrêter purement et simplement l'application, ce qui n'est pas satisfaisant dans la mesure où une fonction relativement bas niveau ne peut au point de vue conceptuel décider à elle seule de l'arrêt du programme.

On peut également ne rien faire du tout, ignorer l'erreur, et continuer l'exécution. Là encore, il y a de grands risques d'erreur par la suite, et de complications dans le déboguage.

La solution que propose le langage Java (ainsi que d'autres langages de programmation comme le C++, voir le cours de Programmation Algorithmique Avancée) est le mécanisme des exceptions. Ce mécanisme a l'avantage d'être standard, alors que la plupart des fonctions ont des mécanismes de gestion d'erreur aussi diversifiés que leur origine. Ce mécanisme est de plus automatisé : en effet, tout ce qui est systématique dans le traitement des erreurs est masqué au développeur. Enfin, il donne une modélisation objet satisfaisante des erreurs.

Toutes les exceptions sont des classes dérivant d'une super-classe *exception*. On définit des classes d'erreurs qui sont dérivées de *exception*. Lorsqu'une erreur se produit, une exception est levée, et les exceptions sont récupérées par l'appelant.

Définition d'une classe d'exception :

```
class err_num extends exception
{
    String msg;
}
```

Levée d'une exception :

```
class frac
{
    int num;
    int den;
    float frac2reel() throws err_num
    {
        if(den==0) throw new err_num();
        return num/(float)den;
    }
};
```

Le type de l'exception doit être spécifié dans la définition de la méthode, par le mot clé *throws*. Lever une exception signifie créer un objet d'une classe dérivée de *exception*. Enfin, il ne suffit de lever l'exception, il faut aussi l'envoyer aux fonctions appelantes, par l'instruction *throw*. Attention, l'envoie d'une exception stoppe l'exécution de la méthode.

Récupération des exceptions :

```
try
{
    frac f=new frac(2,0);
    float t=f.frac2reel();
    System.out.println("salut");
}
```

```
catch(err_num e)
{
    System.out.println(e.msg);
}
```

Principe de la récupération des exceptions : Si une exception est levée par une instruction du bloc *try*, alors un gestionnaire convenable est recherché dans les clauses *catch*(). L'exécution reprend après la séquence *try/catch*, ce qui a une incidence non négligeable sur le déroulement du programme.

La recherche du gestionnaire adéquat se fait dans l'ordre d'apparition, sur les critères suivants :

- Il faut que le type de l'exception définie dans le *catch* soit le type de l'exception levée ;
- Mais aussi si le type n'est autre qu'une super classe de l'exception levée !

Il est important de noter que le premier gestionnaire acceptable stoppe la recherche et est sélectionné. Si aucun gestionnaire ne convient, la recherche se poursuit ans la pile d'exécution (dans les méthodes appelantes), jusqu'à trouver un gestionnaire adapté. Si aucun n'est trouvé, l'application est stoppée et l'exception est renvoyée au système d'exploitation (qui généralement se contente de l'afficher…).

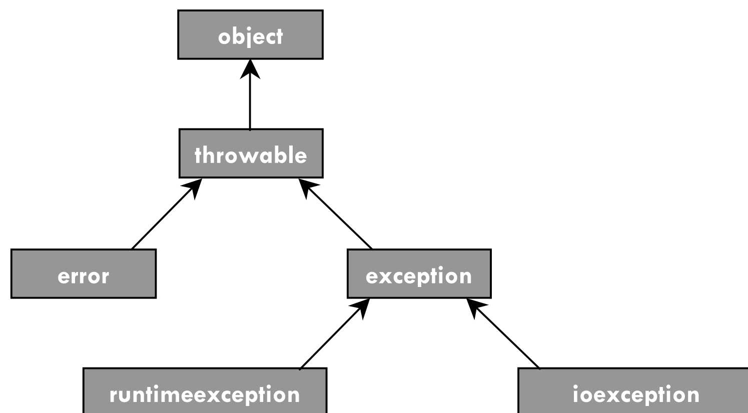
Pour éviter cela, on peut avoir recours à la clause *finally*, qui récupère toutes les exceptions levées.

Exemple :

```
class err extends exception {...};
class err_num extends err {...};
try
{
    frac f=new frac(2,0);
    float t=f.frac2reel();
    System.out.println("Salut !");
}
catch(err e)         {System.out.println("Exception");}
catch(err_num e)     {System.out.println("Erreur numérique");}
catch(ioexception e) {System.out.println("Erreur d'entrée/sortie");}
finally              {System.out.println("Erreur inconnue";}
```

Dans cet exemple, le premier catch récupère toutes les exceptions de type err, ainsi que de tout type dérivant de err, dont err_num. Le second catch n'a donc aucune chance de s'exécuter. Si on veut traiter spécifiquement le cas des erreurs numériques, il faut placer le second catch en premier !

Hiérarchie des classes d'exceptions standards :



Classes abstraites

Certaines classes regroupent des propriétés ou des méthodes mais ne sont pas instanciables. Pour déclarer ce type de classe, appelé classe abstraite, il suffit de faire précéder la déclaration de la classe par le mot clé *abstract*.

L'intérêt des classes abstraites réside dans l'apport dans la cohésion du modèle dans le sens où on considère la classe comme une entité conceptuelle, ne correspondant pas à un objet 'réel'.

Une méthode abstraite n'a pas d'implémentation, par contre les clases dérivées doivent forcément les implémenter, ce qui permet d'imposer des contraintes aux classes dérivées. Les classes abstraites se révèlent ainsi un véritable outil pour la conception.

Une classe abstraite peut ne pas contenir de méthodes abstraites, par contre une classe qui contient au moins une méthode abstraite se doit d'être abstraite.

Exemple :

```
abstract class Valeur_num {abstract void afficher();}
class fraction extends Valeur_num {...}
```

Le mot clé *final* bloque l'héritage polymorphe, et indique que la classe est une feuille de l'arbre d'héritage.

Interfaces

Les interfaces sont nées de la constatation suivante : il n'y a pas de séparation interface/implémentation en Java, contrairement au C++ par exemple (interface dans le fichier *.h*, et implémentation dans le fichier *.cpp* ou dans un fichier déjà compilé).

Une interface est en quelque sorte un contrat entre une classe et ses utilisateurs : un engagement que la classe doit remplir un certain nombre de fonctionnalités. L'interface est un patron définissant un ensemble de méthodes abstraites.

Une interface n'est pas instanciable, et peut être vue comme un type particulier de classe abstraite.

Syntaxe de la définition :

```
interface nom_interface
{
      Définition des fonctions nécessairement abstraites
      Définition des constantes de classe (static final)
};
```

Syntaxe de l'utilisation :

```
class nom_classe implements nom_interface
{
      Implémentation des méthodes d'interface
};
```

Les méthodes d'une interface sont systématiquement virtuelles pures, c'est-à-dire sans aucune implémentation par défaut, et publiques. Il n'est pas nécessaire d'utiliser le mot-clé *abstract*. De fait, les classes implémentant l'interface doivent obligatoirement implémenter ses méthodes.

On peut implémenter des interfaces multiples, ce qui se ramène en quelque sorte à un héritage multiple, notons que ce type d'héritage est impossible à programmer autrement en Java.

Syntaxe :

```
class nom_classe implements nom_interface1, nom_interface2
{
      Implémentation de la classe
};
```

Par contre, le problème de l'héritage répété ne se présente pas : en effet, deux implémentations ne peuvent être concurrentes lors de l'héritage.

Pareillement, une interface peut hériter d'une autre interface (héritage simple) ou de plusieurs interfaces (héritage multiple). On utilise pour cela le mot clé *extends* :

```
interface nom_interface extends nom_interface1, nom_interface2
{
      Implémentation de la classe
};
```

Une implémentation peut être manipulée par un identificateur, de façon à généraliser le polymorphisme aux interfaces elles-mêmes :

```
class maclasse implements interf {…};
maclasse obj = new maclasse;
interf it = obj;
```

Classes primitives

Nous avons vu que les types primitifs sont toujours manipulés par valeur. Cependant, pour chaque type primitif, il existe des classes correspondantes encapsulant ces types et implémentant des fonctionnalités spécifiques, et surtout permettant de manipuler les objets par référence. Citons par exemple les classes :

- *Integer*,
- *Float*,
- etc.

Les Packages

Les packages sont des outils de génie logiciel. Concrètement, un package est un ensemble de classes qui permet de :

- Structurer les applications de grande taille,
- Regrouper les classes en groupes homogènes,
- Stocker et récupérer des ensemble de classes.

On regroupe souvent des classes d'après les critères suivants, ceux-ci étant non exclusifs : par fonctionnalités, par niveau de traitement, par domaine, et par variantes d'implémentation.

Pour définir un package, on utilise la syntaxe suivante :

`package` ***identificateur***

Cette mention doit être placée au début du fichier. Les classes contenues dans ce fichier appartiennent alors à ce pakage. Certaines implémentations exigent que les fichiers soient regroupés dans un répertoire portant précisément le nom du package.

L'utilisateur a également la possibilité d'organiser hiérarchiquement les packages.

```
pack1.pack2
```

Cette ligne définit *pack2* en tant que sous-package de *pack1*. Si on organise les fichiers en répertoires, *pack2* sera un sous-répertoire de *pack1*. Cette structure est d'ailleurs chaudement recommandée.

Si la classe est définie comme publique dans un package, elle sera utilisable par les classes des autres packages. Si par contre elle est privée, elle sera masquée aux autres packages, mais restera utilisable à l'intérieur du package concerné.

Au sein d'une même classe, si les attributs sont privés, ils ne seront visibles que par la classe. S'ils sont publics, leur visibilité sera totale, et s'ils sont protégés, ils seront visibles par toutes les classes du package et par les classes dérivées de la classe. C'est l'accès par défaut.

A ce sujet, l'encapsulation des données, c'est rendre l'accès privé à des variables et réglementer leur accès par des méthodes accesseurs. L'encapsulation garantie ainsi la stabilité de l'interface de la classe.

Pour intégrer un package, il faut avoir recours à l'instruction *import*, ce qui suppose les classes préalablement compilées.

Les packages standards sont contenus dans le fichier classes.zip. Le chemin de recherche des packages est stocké dans la variable d'environnement *CLASSPATH*.

Exemple :

```
CLASSPATH=.;C:\java\JDK\Lib\classes.zip;
```

Composants réutilisables

La programmation des composants réutilisables permet une économie de codage, une sureté de fonctionnement et une efficacité prouvée. En fait, c'est l'objectif premier de l'approche objet. De plus, si un composant est massivement utilisé, à terme, il sera plus fiable.

Pour faire des composants plus fiables, les langages de programmation fournissent le plus souvent une infrastructure, c'est-à-dire un modèle de composants, et des composants : les structures de données, et des algorithmes fréquemment utilisés.

Les *Séquences* sont des ensembles ordonnés d'éléments (listes chaînées) à accès séquentiel permettant de créer, de détruire, d'insérer et de rechercher des objets, et éventuellement d'effectuer des comparaisons, des concaténations, des extractions ou des fusions ;

Les *Ensembles* sont des collections d'objets non ordonnés permettant de créer, d'ajouter, de supprimer des éléments, de tester si l'ensemble est vide ou non, mais également d'opérer des réunions, des intersections, des différences symétriques, des inclusions, etc. ;

Les *Fonctions* offrent la possibilité d'associer un unique élément d'un ensemble F à chaque élément d'un autre ensemble E.

Enfin les *Collections* intègrent un environnement complet permettant la gestion d'un ensemble d'éléments, spécifiés par un ensemble d'interfaces. Leur parcours est basé sur des itérateurs, objets indépendants de la structure sous-jacente et parcourant uniformément les collections.