Chapitre 3

Programmation orientée objet dans Java

1. Introduction

La programmation orientée objet existe depuis 1967 (date d'apparition du langage Simula '67). Cependant, elle n'est vraiment devenue un des paradigmes de la programmation qu'au milieu des années 1980.

Au contraire de la programmation structurée traditionnelle, la programmation orientée objet met dans une même et unique structure les données et les opérations qui leurs sont associées. En programmation traditionnelle, les données et les opérations sur les données sont séparées, les structures de données sont donc envoyées aux procédures et fonctions qui les utilisent. La programmation orientée objet résout de nombreux problèmes inhérents à cette conception en mettant dans une même entité les attributs et les opérations. Cela est plus proche du monde réel, dans lequel tous les objets disposent d'attributs auxquels sont associés des activités. Java est un *pur* langage orienté objet, ce qui signifie que le niveau le plus externe de la structure des données est l'*objet*. *Il n'y a pas de constante, de variable ni de fonction indépendante en Java*. *On accède à toute chose via les classes et les objets*. C'est un des aspects les plus agréables de Java. D'autres langages orientés objets plus hybrides ont conservé des aspects des langages structurés en plus de leurs extensions objet. Par exemple, C++ et Pascal Objet sont des langages orientés objet, mais permettent toujours d'écrire des programmes structurés, ce qui diminue l'efficacité des extensions orientées objet. Ceci n'est pas permis en Java.

2. Le concept de classe

Une classe est le support de l 'encapsulation : c'est un ensemble de données et de fonctions regroupées dans une même entité. Une classe est une description abstraite d'un objet. Les fonctions qui opèrent sur les données sont appelées des méthodes. Instancier une classe consiste à créer un objet sur son modèle. Entre classe et objet il y a, en quelque sorte, le même rapport qu'entre type et variable.

Dans Java : tout appartient à une classe sauf les variables de types primitifs (int, float…). Pour accéder à une classe il faut en déclarer une instance de cette classe (ou un objet). Une classe

comporte sa déclaration, des variables et la définition de ses méthodes. Une classe se compose de deux parties : un en-tête et un corps. Le corps peut être divisé en 2 sections : la déclaration des données et des constantes et la définition des méthodes. Les méthodes et les données sont pourvues d'attributs de visibilité qui gère leur accessibilité par les composants hors de la classe.

2.1. La syntaxe de déclaration d'une classe

```
modificateurs nom_classe [extends classe_mere]
[implements interface]
{
// insérer ici les champs et les méthodes
}
```

Les modificateurs de classe (ClassModifiers) sont :

Modificateur	Rôle
Abstract	la classe contient une ou des méthodes abstraites, qui n'ont pas de définition explicite. Une classe déclarée abstract ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe et qui implémente les méthodes nécessaires pour ne plus être abstraite.
Final	la classe ne peut pas être modifiée, sa redéfinition grâce à l'héritage est interdite. Les classes déclarées final ne peuvent donc pas avoir de classes filles.
Private	la classe n'est accessible qu'à partir du fichier où elle est définie
Public	La classe est accessible partout

Les modificateurs **abstract** et **final** ainsi que **public** et **private** sont mutuellement exclusifs.

Le mot clé **extends** permet de spécifier une superclasse éventuelle : ce mot clé permet de préciser la classe mère dans une relation d'héritage.

Le mot clé **implements** permet de spécifier une ou des interfaces que la classe implémente. Cela permet de récupérer quelques avantages de l'héritage multiple.

L'ordre des méthodes dans une classe n'a pas d'importance. Si dans une classe, on rencontre d'abord la méthode A puis la méthode B, B peut être appelée sans problème dans A.

3. Les objets

Les objets contiennent des attributs et des méthodes. Les attributs sont des variables ou des objets nécessaires au fonctionnement de l'objet. En java, une application est un objet. La classe est la description d'un objet. Un objet est une instance d'une classe. Pour chaque instance d'une classe, le code est le même, seul les données sont différentes à chaque objet.

3.1. La création d'un objet : instancier une classe

Il est nécessaire de définir la déclaration d'une variable ayant le type de l'objet désiré. La déclaration est de la forme **classe nom_variable.**

Exemple :

```
MaClasse m;
```

L'opérateur **new** se charge de créer une instance de la classe et de l'associer à la variable.

Exemple :

```
m = new MaClasse();
```

Il est possible de tout réunir en une seule déclaration.

Exemple :

```
MaClasse m = new MaClasse();
```

Chaque instance d'une classe nécessite sa propre variable. Plusieurs variables peuvent désigner un même objet.

En Java, tous les objets sont instanciés par allocation dynamique. Dans l'exemple, la variable m contient une référence sur l'objet instancié (contient l'adresse de l'objet qu'elle désigne : attention toutefois, il n'est pas possible de manipuler ou d'effectuer des opérations directement sur cette adresse comme en C).

Si m2 désigne un objet de type MaClasse, l'instruction **m2 = m** ne définit pas un nouvel objet mais m et m2 désignent tous les deux le même objet.

L'opérateur **new** est un opérateur de haute priorité qui permet d'instancier des objets et d'appeler une méthode particulière de cet objet : **le constructeur**. Il fait appel à la machine virtuelle pour obtenir l'espace mémoire nécessaire à la représentation de l'objet puis appelle le constructeur pour initialiser l'objet dans l'emplacement obtenu. Il renvoie une valeur qui référence l'objet instancié.

Si l'opérateur new n'obtient pas l'allocation mémoire nécessaire, il lève l'exception **OutOfMemoryError**.

Remarque : sur les objets de type String : Un objet String est automatiquement créé lors de l'utilisation d'une constante chaîne de caractères sauf si celle-ci est déjà utilisée dans la classe. Ceci permet une simplification dans l'écriture des programmes.

Exemple :

String chaine = "bonjour" et String chaine = new String("bonjour") sont équivalents.

3.2. La durée de vie d'un objet

Les objets ne sont pas des éléments statiques et leur durée de vie ne correspond pas forcément à la durée d'exécution du programme.

La durée de vie d'un objet passe par trois étapes :

- la déclaration de l'objet et l'instanciation grâce à l'opérateur new

Exemple :

```
nom_classe nom_objet = new nom_classe( ... );
```

- l'utilisation de l'objet en appelant ces méthodes
- la suppression de l'objet : elle est automatique en java grâce à la machine virtuelle. La restitution de la mémoire inutilisée est prise en charge par le récupérateur de mémoire (**garbage collector**). Il n'existe pas d'instruction delete comme en C++.

3.3. La création d'objets identiques

Exemple :

```
MaClasse m1 = new MaClasse();
MaClasse m2 = m1;
```

m1 et m2 contiennent la même référence et pointent donc tous les deux sur le même objet : les modifications faites à partir d'une des variables modifient l'objet.

Pour créer une copie d'un objet, il faut utiliser la méthode **clone()** : cette méthode permet de créer un deuxième objet indépendant mais identique à l'original. Cette méthode est héritée de la classe **Object** qui est la classe mère de toutes les classes en Java.

Exemple :

```
MaClasse m1 = new MaClasse();
MaClasse m2 = m1.clone();
```

m1 et m2 ne contiennent plus la même référence et pointent donc sur des objets différents.

3.4. Les références et la comparaison d'objets

Les variables déclarés de type objet ne contiennent pas un objet mais une référence vers cet objet. Lorsque l'on écrive c1 = c2 (c1 et c2 sont des objets), on copie la référence de l'objet c2 dans c1 : c1 et c2 réfèrent au même objet (ils pointent sur le même objet). L'opérateur == compare ces références. Deux objets avec des propriétés identiques sont deux objets distincts :

Exemple :

```
Rectangle r1 = new Rectangle(100,50);
Rectangle r2 = new Rectangle(100,50);
if (r1 == r1) { ... } // vrai
```

```
if (r1 == r2) { ... } // faux
```

Pour comparer l'égalité des variables de deux instances, il faut munir la classe d'un méthode à cette effet : la méthode **equals** héritée de **Object**.

Pour s'assurer que deux objets sont de la même classe, il faut utiliser la méthode **getClass()** de la classe **Object** dont toutes les classes héritent.

Exemple :

```
(obj1.getClass().equals(obj2.getClass())
```

3.5. L'objet nul

L'objet null est utilisable partout. Il n'appartient pas à une classe mais il peut être utilisé à la place d'un objet de n'importe quelle classe ou comme paramètre. null ne peut pas être utilisé comme un objet normal : il n'y a pas d'appel de méthodes et aucune classe ne puisse en hériter.

Le fait d'initialiser une variable référent un objet à null permet au ramasseur de miettes de libérer la mémoire allouée à l'objet.

3.6. Les variables de classes

Elles ne sont définies qu'une seule fois quelque soit le nombre d'objets instanciés de la classe. Leur déclaration est accompagnée du mot clé static

Exemple :

```
public class MaClasse() {
static int compteur = 0;
}
```

L'appartenance des variables de classe à une classe entière et non à un objet spécifique permet de remplacer le nom de la variable par le nom de la classe.

Exemple :

```
MaClasse m = new MaClasse();
int c1 = m.compteur;
int c2 = MaClasse.compteur;
```

c1 et c2 possèdent la même valeur.

Ce type de variable est utile pour par exemple compter le nombre d'instanciation de la classe qui est faite.

3.7. La variable this

Cette variable sert à référencer dans une méthode l'instance de l'objet en cours d'utilisation. this est un objet qui est égale à l'instance de l'objet dans lequel il est utilisé.

Exemple :

```
private int nombre;
public maclasse(int nombre) {
nombre = nombre; // variable de classe = variable en paramètre
du
// constructeur
}
```

Il est préférable d'écrire :

this.nombre = nombre;

Cette référence est habituellement implicite :

Exemple :

```
class MaClasse() {
String chaine = "test";
Public String getChaine() { return chaine) ;
// est équivalent à public String getChaine (this.chaine);
}
```

This est aussi utilisé quand l'objet doit appeler une méthode en se passant lui même en paramètre de l'appel.

3.8. L'opérateur instanceof

L'opérateur **instanceof** permet de déterminer la classe de l'objet qui lui est passé en paramètre. La syntaxe est **objet instanceof classe**.

Exemple :

```
void testClasse(Object o) {
if (o instanceof MaClasse )
System.out.println("o est une instance de la classe MaClasse");
else System.out.println("o n'est pas un objet de la classe
MaClasse");
}
```

Il n'est toutefois pas possible d'appeler une méthode de l'objet car il est passé en paramètre avec un type **Object**.

Exemple :

```
void afficheChaine(Object o) {
if (o instanceof MaClasse)
System.out.println(o.getChaine());
// erreur à la compil car la méthode getChaine()
```

```
//n'est pas définie dans la classe Object
}
```

Pour résoudre le problème, il faut utiliser la technique du casting (conversion).

Exemple :

```
void afficheChaine(Object o) {
if (o instanceof MaClasse)
{
MaClasse m = (MaClasse) o;
System.out.println(m.getChaine());
// OU System.out.println( ((MaClasse) o).getChaine() );
}
}
```

4. Les modificateurs d'accès

Ils se placent avant ou après le type de l'objet mais la convention veut qu'ils soient placés avant.

Ils s'appliquent aux classes et/ou aux méthodes et/ou aux attributs.

Ils ne peuvent pas être utilisés pour qualifier des variables locales : seules les variables d'instances et de classes peuvent en profiter.

Ils assurent le contrôle des conditions d'héritage, d'accès aux éléments et de modification de données par les autres objets.

4.1. Les mots clés qui gèrent la visibilité des entités

De nombreux langages orientés objet introduisent des attributs de visibilité pour réglementer l'accès aux classes et aux objets, aux méthodes et aux données.

Il existe 3 modificateurs qui peuvent être utilisés pour définir les attributs de visibilité des entités (classes, méthodes ou attributs) : public, private et protected. Leur utilisation permet de définir des niveaux de protection différents (présenté dans un ordre croissant de niveau de protection offert):

Modificateur	Rôle
Public	Une variable, méthode ou classe déclarée public est visible par tous les autres objets. Dans la version 1.0, une seule classe public est permise par fichier et son nom doit correspondre à celui du fichier. Dans la philosophie orientée objet aucune donnée d'une classe ne devraient être déclarée publique : il est préférable d'écrire des méthodes pour la consulter et la modifier.
par défaut : package friendly	Il n'existe pas de mot clé pour définir ce niveau, qui est le niveau par défaut lorsqu'aucun modificateur n'est précisé. Cette déclaration permet à une entité (classe, méthode ou variable) d'être visible par toutes les classes se trouvant dans le même package.
Protected	Si une classe, une méthode ou une variable est déclarée protected, seules les méthodes présentes dans le même package que cette classe ou ses sous classes pourront y acceder. On ne peut pas qualifier une classe avec protected.
Private	C'est le niveau de protection le plus fort. Les composants ne sont visibles qu'à l'intérieur de la classe : ils ne peuvent être modifiés que par des méthodes définies dans la classe prévues à cet effet.

Ces modificateurs d'accès sont mutuellement exclusifs.

4.2. Le mot clé static

Le mot clé static s'applique aux variables et aux méthodes.

Les variables d'instance sont des variables propres à un objet. Il est possible de définir une variable de classe qui est partagée entre toutes les instances d'une même classe : elle n'existe donc qu'une seule fois. Une telle variable permet de stocker une constante ou une valeur modifiée tour à tour par les instances de la classe. Elle se définit avec le mot clé static.

Exemple :

```
public class Cercle {
static float pi = 3.1416f;
float rayon;
public Cercle(float rayon) { this.rayon = rayon; }
public float surface() { return rayon * rayon * pi;}
}
```

Il est aussi possible par exemple de mémoriser les valeurs min et max d'un ensemble d'objets de même classe.

Une méthode static est une méthode qui n'agit pas sur des variables d'instance mais uniquement sur des variables de classe. Les méthodes ainsi définies peuvent être appelée avec la notation

classe.méthode au lieu de objet.méthode. Ces méthodes peuvent être utilisées sans instancier un objet de la classe.

Il n'est pas possible d'appeler une méthode d'instance ou d'accéder à une variable d'instance à partir d'une méthode de classe statique.

4.3. Le mot clé final

Le mot clé final s'applique aux variables, aux méthodes et aux classes.

Une variable qualifiée de **final** signifie que la variable est **constante**. Le compilateur ne contrôle ni empêche la modification. On ne peut déclarer de variables **final** locales à une méthode. Les constantes sont qualifiées de **final et static**.

Exemple :

```
public static final float PI = 3.1416f;
```

Une méthode final ne peut pas être redéfinie dans une sous classe. Une méthode possédant le modificateur final pourra être optimisée par le compilateur car il est garanti qu'elle ne sera pas sous classée.

Lorsque le modificateur final est ajouté à une classe, il est interdit de créer une classe qui en hérite.

4.4. Le mot clé abstract

Le mot clé abstract s'applique aux méthodes et aux classes.

Abstract indique que la classe ne pourra être instanciée telle quelle. De plus, toutes les méthodes de cette classe abstract ne sont pas implémentées et devront être redéfinies par des méthodes complètes dans ses sous classes.

Abstract permet de créer une classe qui sera une sorte de moule. Toutes les classes dérivées pourront profiter des méthodes héritées et n'auront à implémenter que les méthodes déclarées abstract.

Exemple :

```
abstract class ClasseAbstraite {
ClasseBastraite() { ... //code du constructeur }
void méthode() { ... // code partagé par tous les descendants}
abstract void méthodeAbstraite();
}
class ClasseComplete extends ClasseAbstraite {
ClasseComplete() { super(); … }
void méthodeAbstraite() { ... // code de la méthode }
```

```
// void méthode est héritée
}
```

Une méthode abstraite est une méthode déclarée avec le modificateur abstract et sans corps.
Elle correspond à une méthode dont on veut forcer l'implémentation dans une sous classe.
L'abstraction permet une validation du codage : une sous classe sans le modificateur abstract et sans définition explicite d'une ou des méthodes abstraites génère une erreur de compilation. Une classe est automatiquement abstraite dès qu'une de ses méthodes est déclarée abstraite. Il est possible de définir une classe abstraite sans méthodes abstraites.

4.5. Le mot clé synchronized

Permet de gérer l'accès concurrent aux variables et méthodes lors de traitement de thread (exécution « simultanée » de plusieurs petites parties de code du programme)

4.6. Le mot clé volatile

Le mot clé **volatile** s'applique aux variables.
Précise que la variable peut être changée par un périphérique ou de manière asynchrone. Cela indique au compilateur de ne pas stocker cette variable dans des registres. A chaque utilisation, on lit la valeur et on réécrit immédiatement le résultat s'il a changé.

4.7. Le mot clé native

Une méthode **native** est une méthode qui est implémentée dans un autre langage. L'utilisation de ce type de méthode limite la portabilité du code mais permet une vitesse d'exécution plus rapide.

5. Les propriétés ou attributs

Les données d'une classe sont contenues dans les propriétés ou attributs. Ce sont des variables qui peuvent être des variables d'instances, des variables de classes ou des constantes.

5.1. Les variables d'instances

Une variable d'instance nécessite simplement une déclaration de la variable dans le corps de la classe.

Exemple :

```
public class MaClasse {
public int valeur1 ;
int valeur2 ;
protected int valeur3 ;
private int valeur4 ;
```

```
}
```

Chaque instance de la classe a accès à sa propre occurrence de la variable.

5.2. Les variables de classes

Les variables de classes sont définies avec le mot clé static

Exemple :

```
public class MaClasse {
static int compteur ;
}
```

Chaque instance de la classe partage la même variable.

5.3. Les constantes

Les constantes sont définies avec le mot clé final : leur valeur ne peut pas être modifiée.

Exemple :

```
public class MaClasse {
final double pi=3.14 ;
}
```

6. Les méthodes

Les méthodes sont des fonctions qui implémentent les traitements de la classe.

6.1. La syntaxe de la déclaration

La syntaxe de la déclaration d'une méthode est :

```
modificateurs type_retourné nom_méthode ( arg1, ... ) {... }
// définition des variables locales et du bloc d'instructions
}
```

Le type retourné peut être élémentaire ou correspondre à un objet. Si la méthode ne retourne rien, alors on utilise **void.**

Le type et le nombre d'arguments déclarés doivent correspondrent au type et au nombre d'arguments transmis. Il n'est pas possible d'indiquer des valeurs par défaut dans les paramètres. Les arguments sont passés par valeur : la méthode fait une copie de la variable qui lui est locale. Lorsqu'un objet est transmis comme argument à une méthode, cette dernière reçoit une référence qui désigne son emplacement mémoire d'origine et qui est une copie de la variable. Il est possible de modifier l'objet grâce à ces méthodes mais il n'est pas possible de remplacer la référence contenue dans la variable passée en paramètre : ce changement n'aura lieu que localement à la méthode.

Les modificateurs de méthodes sont :

Modificateur	Rôle
Public	la méthode est accessible aux méthodes des autres classes
Private	l'usage de la méthode est réservé aux autres méthodes de la même classe
Protected	la méthode ne peut être invoquée que par des méthodes de la classe ou de ses sous classes
Final	la méthode ne peut être modifiée (redéfinition lors de l'héritage interdite)
Static	la méthode appartient simultanément à tous les objets de la classe (comme une constante déclarée à l'intérieur de la classe). Il est inutile d'instancier la classe pour appeler la méthode mais la méthode ne peut pas manipuler de variable d'instance. Elle ne peut utiliser que des variables de classes.
synchronized	la méthode fait partie d'un thread. Lorsqu'elle est appelée, elle barre l'accès à son instance. L'instance est à nouveau libérée à la fin de son exécution.
Native	le code source de la méthode est écrit dans un autre langage

Sans modificateur, la méthode peut être appelée par toutes autres méthodes des classes du package auquel appartient la classe.

La valeur de retour de la méthode doit être transmise par l'instruction return. Elle indique la valeur que prend la méthode et termine celle ci : toutes les instructions qui suivent return sont donc ignorées.

Exemple :

```
int add(int a, int b) {
return a + b;
}
```

Il est possible d'inclure une instruction return dans une méthode de type void : cela permet de quitter la méthode.

La méthode main() de la classe principale d'une application doit être déclarée de la façon suivante :

Déclaration d'une méthode main() :

```
public static void main (String args[]) { ... }
```

Si la méthode retourne un tableau alors les [] peuvent être préciser après le type de retour ou après la liste des paramètres :

Exemple :

```
int[] valeurs() { … }
```

int valeurs()[] { … }

6.2. La transmission de paramètres

Lorsqu'un objet est passé en paramètre, ce n'est pas l'objet lui même qui est passé mais une référence sur l'objet. La référence est bien transmise par valeur et ne peut pas être modifiée mais l'objet peut être modifié via un message (appel d'une méthode).

Pour transmettre des arguments par référence à une méthode, il faut les encapsuler dans un objet qui prévoit les méthodes nécessaires pour les mises à jour.

Si un objet o transmet sa variable d'instance v en paramètre à une méthode m, deux situations sont possibles :

- si v est une variable primitive alors elle est passée par valeur : il est impossible de la modifier dans m pour que v en retour contienne cette nouvelle valeur.
- si v est un objet alors m pourra modifier l'objet en utilisant une méthode de l'objet passé en paramètre.

6.3. L'émission de messages

Un message est émis lorsqu'on demande à un objet d'exécuter l'une de ses méthodes.

La syntaxe d'appel d'une méthode est : nom_objet.nom_méthode(parametre, …) ;

Si la méthode appelée ne contient aucun paramètre, il faut laisser les parenthèses vides.

6.3. L'enchaînement de références à des variables et à des méthodes

Exemple :

System.out.Println("bonjour");

Deux classes sont impliqués dans l'instruction : **System** et **PrintStream**. La classe **System** possède une variable nommée **out** qui est un objet de type **PrintStream**.**Println()** est une méthode de la classe **PrintStream**. L'instruction signifie : « *utilise la méthode Println() de la variable out de la classe System* ».

6.4. La surcharge de méthodes

La surcharge d'une méthode permet de définir plusieurs fois une même méthode avec des arguments différents. Le compilateur choisit la méthode qui doit être appelée en fonction du nombre et du type des arguments. Ceci permet de simplifier l'interface des classes vis-à-vis des autres classes.

Une méthode est surchargée lorsqu'elle exécute des actions différentes selon le type et le nombre de paramètres transmis.

Exemple :

```
class affiche{
public void afficheValeur(int i) {
System.out.pintln("nombre entier =" + i);
}
public void afficheValeur(float f) {
System.out.println("nombre flottant =" + f);
}
}
```

Il n'est pas possible d'avoir deux méthodes de même nom dont tous les paramètres sont identiques et dont seul le type retourné diffère.

Exemple :

```
class Affiche{
public float convert(int i){
return((float) i);
}
public double convert(int i){
return((double) i);
}
}
```

Résultat à la compilation :

```
C:\>javac Affiche.java
Affiche.java:5: Methods can't be redefined with a different return
type: double
convert(int) was float convert(int)
public double convert(int i){
^
1 error
```

6.5. La signature des méthodes et le polymorphisme

Il est possible de donner le même nom à deux méthodes différentes à condition que les signatures de ces deux méthodes soient différentes. La signature d'une méthode comprend le nom de la classe, le nom de la méthode et les types des paramètres. Cette facilité permet de mettre en oeuvre le polymorphisme.

Le polymorphisme est la capacité, pour un même message de correspondre à plusieurs formes de traitements selon l'objet auquel ce message est adressé. La gestion du polymorphisme est assurée par la machine virtuelle dynamiquement à l'exécution.

6.6. Les constructeurs

La déclaration d'un objet est suivie d'une sorte d'initialisation par le moyen d'une méthode particulière appelée constructeur pour que les variables aient une valeur de départ. Elle n'est systématiquement invoquée que lors de la création d'un objet.

Le constructeur suit la définition des autres méthodes excepté que son nom doit obligatoirement correspondre à celui de la classe et qu'il n'est pas typé, pas même void, donc il ne peut pas y avoir d'instruction return dans un constructeur. On peut surcharger un constructeur.

La définition d'un constructeur est facultative. Si elle n'est pas définie, la machine virtuelle appelle un constructeur par défaut vide créé automatiquement. Dès qu'un constructeur est explicitement défini, Java considère que le programmeur prenne en charge la création des constructeurs et que le mécanisme par défaut, qui correspond à un constructeur sans paramètres, est supprimé. Si on souhaite maintenir ce mécanisme, il faut définir explicitement un constructeur sans paramètres.

Il existe plusieurs manières de définir un constructeur :

1. Le constructeur simple : ce type de constructeur ne nécessite pas de définition explicite : son existence découle automatiquement de la définition de la classe.

Exemple :

```
public MaClasse() {}
```

2. Le constructeur avec initialisation fixe : il permet de créer un constructeur par défaut

Exemple :

```
public MaClasse() {
nombre = 5;
}
```

3. Le constructeur avec initialisation des variables : pour spécifier les valeurs de données à initialiser on peut les passer en paramètres au constructeur

Exemple :

```
public MaClasse(int valeur) {
nombre = valeur;
}
```

6.7. Le destructeur

Un destructeur permet d'exécuter du code lors de la libération de la place mémoire occupée par l'objet. En java, les destructeurs appelés finaliseurs (finalizers), sont automatiquement appelés par le garbage collector.

Pour créer un finaliseur, il faut redéfinir la méthode finalize() héritée de la classe Object.

6.8. Les accesseurs

L'encapsulation permet de sécuriser l'accès aux données d'une classe. Ainsi, les données déclarées private à l'intérieur d'une classe ne peuvent être accédées et modifiées que par des méthodes définies dans la même classe. Si une autre classe veut accéder aux données de la classe, l'opération n'est possible que par l'intermédiaire d'une méthode de la classe prévue à cet effet. Ces appels de méthodes sont appelés « échanges de message ».

Un accesseur est une méthode publique qui donne l'accès à une variable d'instance privée.

Pour une variable d'instance, il peut ne pas y avoir d'accesseur, un seul accesseur en lecture ou un accesseur en lecture et un en écriture. Par convention, les accesseurs en lecture commencent par get et les accesseurs en écriture commencent par set.

Exemple :

```
private int valeur = 13;
public int getValeur(){
return(valeur);
}
public void setValeur(int val) {
valeur = val;
}
```

7. L'héritage

7.1. Le principe de l'héritage

L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution. Grâce à l'héritage, les objets d'une classe ont accès aux données et aux méthodes de la classe parent et peuvent les étendre. Les sous classes peuvent redéfinir les variables et les méthodes héritées. Pour les variables, il suffit de les redéclarer sous le même nom avec un type différent. Les méthodes sont redéfinies avec le même nom, les mêmes types et le même nombre d'arguments, sinon il s'agit d'une surcharge. L'héritage successif de classes permet de définir une hiérarchie de classe qui ce compose de super classes et de sous classes. Une classe qui hérite

d'une autre est une sous classe et celle dont elle hérite est une super classe. Une classe peut avoir plusieurs sous classes. Une classe ne peut avoir qu'une seule classe mère : il n'y a pas d'héritage multiple en java.

Object est la classe parente de toutes les classes en java. Toutes les variables et méthodes contenues dans Object sont accessibles à partir de n'importe quelle classe car par héritage successif toutes les classes héritent d'Object.

7.2. La mise en oeuvre de l'héritage

On utilise le mot clé extends pour indiquer qu'une classe hérite d'une autre. En l'absence de ce mot réservé associé à une classe, le compilateur considère la classe Object comme classe parent.

Exemple :

```
class Fille extends Mere { ... }
```

Pour invoquer une méthode d'une classe parent, il suffit d'indiquer la méthode préfixée par super. Pour appeler le constructeur de la classe parent il suffit d'écrire super(paramètres) avec les paramètres adéquats.

Le lien entre une classe fille et une classe parent est géré par le langage : une évolution des règles de gestion de la classe parent conduit à modifier automatiquement la classe fille dès que cette dernière est recompilée.

En java, il est obligatoire dans un constructeur d'une classe fille de faire appel explicitement ou implicitement au constructeur de la classe mère.

7.3. L'accès aux propriétés héritées

Les variables et méthodes définies avec le modificateur d'accès public restent publiques à travers l'héritage et toutes les autres classes.

Une variable d'instance définie avec le modificateur private est bien héritée mais elle n'est pas accessible directement, elle l'est via les méthodes héritées.

Si l'on veut conserver pour une variable d'instance une protection semblable à celle assurée par le modificateur private, il faut utiliser le modificateur protected. La variable ainsi définie sera hérité dans toutes les classes descendantes qui pourront y accéder librement mais ne sera pas accessible hors de ces classes directement.

7.4. Le transtypage induit par l'héritage facilitent le Polymorphisme

L'héritage définit un cast implicite de la classe fille vers la classe mère : on peut affecter à une référence d'une classe n'importe quel objet d'une de ses sous classes.

Exemple : la classe Employe hérite de la classe Personne

```
Personne p = new Personne ("Dupond", "Jean");
Employe e = new Employe("Durand", "Julien", 10000);
p = e ; // ok : Employe est une sous classe de Personne
Object obj;
obj = e ; // ok : Employe herite de Personne qui elle même hérite de Object
```

Il est possible d'écrire le code suivant si Employé hérite de Personne

Exemple :

```
Personne[] tab = new Personne[10];
tab[0]= new Personne("Dupond","Jean");
tab[1]= new Employe("Durand", "Julien", 10000);
```

Il est possible de surcharger une méthode héritée : la forme de la méthode à exécuter est choisie en fonction des paramètres associés à l'appel.

Compte tenu du principe de l'héritage, le temps d'exécution du programme et la taille du code source et de l'exécutable augmentent.

7.5. La redéfinition d'une méthode héritée

La redéfinition d'une méthode héritée doit impérativement conserver la déclaration de la méthode parent (type et nombre de paramètres, la valeur de retour et les exceptions propagées doivent être identiques).

Si la signature de la méthode change, ce n'est plus une redéfinition mais une surcharge. Cette nouvelle méthode n'est pas héritée : la classe mère ne possède pas de méthode possédant cette signature.

7.6. Les interfaces et l'héritage multiple

Avec l'héritage multiple, une classe peut hériter en même temps de plusieurs super classes. Ce mécanisme n'existe pas en java. Les interfaces permettent de mettre en œuvre un mécanisme de remplacement.

Une interface est un ensemble de constantes et de déclarations de méthodes correspondant un peu à une classe abstraite. C'est une sorte de standard auquel une classe peut répondre. Tous les objets qui se conforment à cette interface (qui implémentent cette interface) possèdent les méthodes et

les constantes déclarées dans celle-ci. Plusieurs interfaces peuvent être implémentées dans une même classe.

Les interfaces se déclarent avec le mot clé **interface** et sont intégrées aux autres classes avec le mot clé **implements**. Une interface est implicitement déclarée avec le modificateur **abstract**.

Déclaration d'une interface :

```
[public] interface nomInterface [extends nomInterface1, nomInterface2 ... ]
{
// insérer ici des méthodes ou des champs static
}
```

Implémentation d'une interface :

```
Modificateurs class nomClasse [extends superClasse]
[implements nomInterface1, nomInterface 2, ...] {
//insérer ici des méthodes et des champs
}
```

Exemple :

```
interface AfficheType {
void AfficheType();
}
class Personne implements AfficheType {
public void afficheType() {
System.out.println("Je suis une personne");
}
}
class Voiture implements AfficheType {
public void afficheType() {
System.out.println("Je suis une voiture");
}
}
```

Exemple : déclaration d'une interface à laquelle doit se conformer tout individu

```
interface Individu {
String getNom();
String getPrenom();
Date getDateNaiss();
}
```

Toutes les méthodes d'une interface sont publiques et abstraites : elles sont implicitement déclarées comme telles.
Une interface peut être d'accès public ou package. Si elle est publique, toutes ses méthodes sont publiques même si elles ne sont pas déclarées avec le modificateur public. Si elle est d'accès package, il s'agit d'une interface d'implémentation pour les autres classes du package et ses méthodes ont le même accès package : elles sont accessible à toutes les classes du package.
Les seules variables que l'on peut définir dans une interface sont des variables de classe qui doivent être constantes : elles sont donc implicitement déclarées avec le modificateur static et final même si elles sont définies avec d'autres modificateurs.
Toute classe qui implémente cette interface doit au moins posséder les méthodes qui sont déclarées dans l'interface. L'interface ne fait que donner une liste de méthodes qui seront à définir dans les classes qui implémentent l'interface.
Les méthodes déclarées dans une interface publique sont implicitement publiques et elles sont héritées par toutes les classes qui implémentent cette interface. Une telle classe doit, pour être instanciable, définir toutes les méthodes héritées de l'interface.
Une classe peut implémenter une ou plusieurs interfaces tout en héritant de sa classe mère.
L'implémentation d'une interface définit un cast : l'implémentation d'une interface est une forme d'héritage. Comme pour l'héritage d'une classe, l'héritage d'une classe qui implémente une interface définit un cast implicite de la classe fille vers cette interface. Il est important de noter que dans ce cas il n'est possible de faire des appels qu'à des méthodes de l'interface.
Pour utiliser des méthodes de l'objet, il faut définir un cast explicite : il est préférable de contrôler la classe de l'objet pour éviter une exception ClassCastException à l'exécution.

7.7. Des conseils sur l'héritage

Lors de la création d'une classe « mère » il faut tenir compte des points suivants :

- la définition des accès aux variables d'instances, très souvent privées, doit être réfléchie entre protected et private ;
- pour empêcher la redéfinition d'une méthode (surcharge) il faut la déclarer avec le modificateur final. Lors de la création d'une classe fille, pour chaque méthode héritée qui n'est pas final, il faut envisager les cas suivant :
- la méthode héritée convient à la classe fille : on ne doit pas la redéfinir ;

- la méthode héritée convient mais partiellement du fait de la spécialisation apportée par la classe fille : il faut la redéfinir voir la surcharger. La plupart du temps une redéfinition commencera par appeler la méthode héritée (via super) pour garantir l'évolution du code ;
- la méthode héritée ne convient pas : il faut redéfinir ou surcharger la méthode sans appeler la méthode héritée lors de la redéfinition.

8. Les classes internes

Les classes internes (inner classes) sont une extension du langage java introduite dans la version 1.1 du JDK. Ce sont des classes qui sont définies dans une autre classe. Les difficultés dans leur utilisation concerne leur visibilité et leur accès aux membres de la classe dans laquelle elles sont définies.

Exemple très simple :

```
public class ClassePrincipale1 {
class ClasseInterne {
}
}
```

Les classes internes sont particulièrement utiles pour :

- permettre de définir une classe à l'endroit ou une seule autre en a besoin
- définir des classes de type adapter (essentiellement à partir du JDK 1.1 pour traiter des évenements émis par les interfaces graphiques)
- définir des méthodes de type callback d'une façon générale

Il est possible d'imbriquer plusieurs classes internes. Java ne possède pas de restrictions sur le nombre de classes qu'il est ainsi possible d'imbriquer. En revanche une limitation peut intervenir au niveau du système d'exploitation en ce qui concerne la longueur du nom du fichier .class généré pour les différentes classes internes.

Si plusieurs classes internes sont imbriquées, il n'est pas possible d'utiliser un nom pour la classe qui soit déjà attribuée à une de ces classes englobantes. Le compilateur génèrera une erreur à la compilation.

Exemple :

```
public class ClassePrincipale6 {
class ClasseInterne1 {
class ClasseInterne2 {
class ClasseInterne3 {
```

```
}
}
}
}
```

Accès aux classes internes :

Exemple :

```
public class ClassePrincipale8 {
public class ClasseInterne {
}
public static void main(String[] args) {
ClassePrincipale8 cp = new ClassePrincipale8();
ClassePrincipale8.ClasseInterne ci = cp. new ClasseInterne() ;
System.out.println(ci.getClass().getName());
}
}
```

Resultat :

```
java ClassePrincipale8
ClassePrincipale8$ClasseInterne
```

L'accessibilité à la classe interne respecte les règles de visibilité du langage. Il est même possible de définir une classe interne private pour limiter son accès à sa seule classe principale.

Exemple :

```
public class ClassePrincipale7 {
private class ClasseInterne {
}
}
```

Il n'est pas possible de déclarer des membres statiques dans une classe interne :

Exemple :

```
public class ClassePrincipale10 {
public class ClasseInterne {
static int var = 3;
}
}
```

Resultat :

```
javac ClassePrincipale10.java
ClassePrincipale10.java:3: Variable var can't be static in inner
```

```
class ClassePrincipale10. ClasseInterne. Only members of interfaces
and top-level classes canbe static.
static int var = 3;
^
1 error
```

Pour pouvoir utiliser une variable de classe dans une classe interne, il faut la déclarer dans sa classe englobante.

Il existe quatre types de classes internes :

- les classes internes non statiques : elles sont membres à part entière de la classe qui les englobent et peuvent accéder à tous les membres de cette dernière
- les classes internes locales : elles sont définies dans un block de code. Elles peuvent être static ou non.
- les classes internes anonymes : elles sont définies et instanciées à la volée sans posséder de nom
- les classe internes statiques : elles sont membres à part entière de la classe qui les englobent et peuvent accéder uniquement aux membres statiques de cette dernière

9. La gestion dynamique des objets

Tout objet appartient à une classe et Java sait la reconnaître dynamiquement.

Java fournit dans son API un ensemble de classes qui permettent d'agir dynamiquement sur des classes. Cette technique est appelée introspection et permet :

- De décrire une classe ou une interface : obtenir son nom, sa classe mère, la liste de ces méthodes, de ses variables de classe, de ses constructeurs et de ses variables d'instances ;
- D'agir sur une classe en envoyant, à un objet Class des messages comme à tout autre objet. Par exemple, créer dynamiquement à partir d'un objet Class une nouvelle instance de la classe représentée.