Lycée polyvalent DIDEROT **Département IRIS**
61, rue David d'Angers
75019 PARIS
http://www.diderot.org

COURS

Java – Notions fondamentales

Auteur	Version - Date	Nom du fichier
Génaël VALET	Version 3.1 - Sept 2007	Cours N°1 - Structures fondamentales du langage JAVA - IRIS.docx

Apprentissage des notions de bases du langage JAVA : Les classes, les objets, la syntaxe, les tableaux, les boucles, …

Sommaire

Chapitre 1 : Introduction

Le langage JAVA est un langage évolué et objet et dont le succès est du à son architecture permettant une portabilité et une souplesse adaptée aux exigences de la programmation orientée objet.
Ce cours va aider le débutant à connaître la structure du langage telle que les variables et les types de données ou encore les opérateurs, les commentaires et la manipulation de chaînes.
Il initiera le débutant à l'architecture objet des classes contenant des méthodes et des champs.

Chapitre 2 : Structure d'un programme JAVA

A. L'environnement d'exécution d'un programme

Un programme écrit en JAVA est stocké dans un fichier dont l'extension est *.java* . Ce fichier est d'abord compilé et le résultat donne un autre fichier binaire dont l'extension est *.class*.

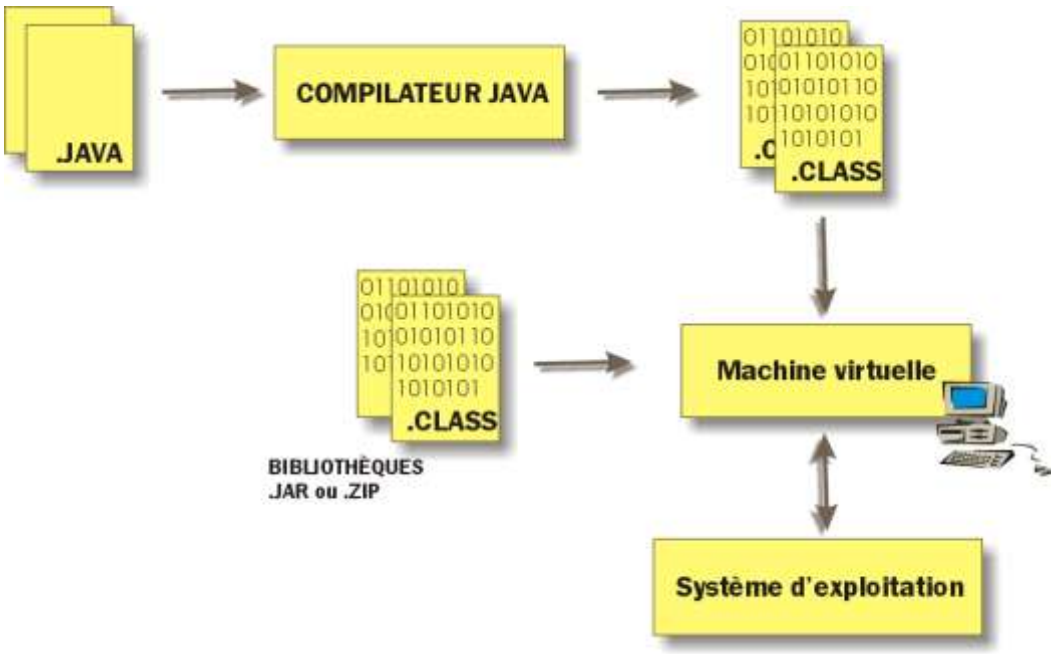


Schéma 1 - Explicatif sur la compilation et l'exécution d'un programme JAVA

A.1. La compilation

Le compilateur JAVA vérifie qu'il n'y a pas d'erreur de syntaxe ou que le programme ne présente pas d'incohérence par rapport à l'utilisation des classes et des objets. La sécurité qu'offre JAVA fait qu'aucune erreur ne peux passer inaperçue dès la compilation. En fait, les concepteurs de JAVA sont partis du principe qu'il vaut mieux prévenir que guérir et la conséquence est que le programmeur ne pourra pas exécuter son programme tant qu'il n'a pas levé toutes les interdictions et restrictions du langage.

En C/C++ , le compilateur ne vérifie pas l'utilisation qui est faite de la mémoire et des pointeurs. En java, les pointeurs n'existent pas et on utilise alors des références aux objets.

A.2. L'exécution

L'exécution du programme peut alors s'effectuer sur n'importe quelle plate-forme puisque le fichier *.class* généré regroupe un ensemble de *primitives*[1] qui seront alors interprétées par la *machine virtuelle*[2] du système.

A.3. L'utilisation des *packages*

Par souci d'organisation, les classes sont classées dans des packages. Cette organisation est totalement indépendante de l'emplacement physique des fichiers *.class*. En effet, rien n'empêche 2 classes situées dans des répertoires distincts de faire partie du même package.
Les packages sont définis de manière arborescente exactement comme dans un système de fichier classique. Par exemple, la classe *String* se situe dans le package « *java.lang* »
Ces packages se présentent soit sous la forme d'un répertoire contenant des sous répertoires, ou soit sous la forme d'archives. Dans le cas des archives, les classes d'un même package peuvent être regroupées dans fichier *.jar* ou *.zip* , formant ainsi des sortes de bibliothèques dans lesquelles vous pouvez puiser à tout moment.
Lors de la compilation, des liens sont effectués vers ces bibliothèques grâce au mot clé **`import`** inséré dans le code du programme.

Avec Java 2 , il existe environ plus de 5000 classes utilisables dans vos programmes livrés avec l'environnement de développement de base appelé le J2SE (Java 2 Standard Edition)

Voici un exemple d'utilisation des *packages* dans un programme qui permettra l'utilisation de toutes les classes situées dans le package *java.io* et de la classe *GregorianCalendar* du package *java.io :*

```
/* Extrait d'un programme JAVA */

import java.io.*;          // Importe toutes les classes de java.io

import java.util.GregorianCalendar ; //  Importe seulement la classe
                                              GregorianCalendar
```

A.4. Le fameux CLASSPATH

Dans un environnement JAVA, le « CLASSPATH » défini un ensemble de répertoires physiques contenant des classes ou des packages. Cet ensemble va permettre au compilateur et à la machine virtuelle de rechercher les classes utilisées dans tous ces répertoires.
Le terme « CLASSPATH » désigne en fait 2 méthodes différentes pour définir cet ensemble de répertoires : La variable d'environnement et le paramètre *–classpath* des outils du *J2SE* tels que *java* ou *javac*

[1] Fonctions basiques compréhensibles par tous les systèmes compatibles JAVA
[2] Logiciel intégré au système d'exploitation permettant de comprendre les primitives JAVA

a. La variable d'environnement

Cette variable se présente sous forme d'une liste de répertoires séparés par un « ; » ou un « : » selon les systèmes. Elle est gérée par le système d'exploitation et peut-être sauvegardée même après le redémarrage du système.
Pour les systèmes *Windows* , la variable d'environnement peut-être définie grâce à la commande suivante :

```
SET CLASSPATH=.;C:\DEV\;C:\PROJETS\LIBS
```

La procédure d'affectation des variables d'environnement peut varier selon les versions de Windows. L'utilisation de la commande SET n'est pas toujours appropriée pour les systèmes Windows 2000/XP qui différencient les variables systèmes et les variables utilisateurs (Voir Panneau de configuration -> Système -> Avancés -> Variables d'environnement

La 1ère valeur (.) signifie que la recherche des classes s'effectuera également dans le répertoire courant. N'oubliez pas de mettre ce point , surtout si vous utilisez les outils de commande tels que javac ou java

Pour les systèmes sous *Unix (sh/bash/ksh) :*

```
setenv CLASSPATH /home/dev:/home/projets:.
```

b. Le paramètre –classpath

Les outils Java tels que *javac* et *java* peuvent s'utiliser avec un paramètre *–classpath* qui défini à chaque utilisation, un CLASSPATH différent. Voici un exemple qui compile le fichier *Controller.java* :

```
javac -classpath c:\java\classes\javagently.jar;. Controller.java
```

L'inconvénient d'une telle méthode est de vous obliger à saisir à chaque compilation les chemins du CLASSPATH. Néanmoins tous les outils graphiques du marché savent tirer profit de cette syntaxe en incluant avec le paramètre tous les répertoires constituant votre projet complet.

Si vous utilisez le paramètre classpath , la variable d'environnement préalablement définie ne sera pas prise en compte.

B. Une classe est un fichier

Le programme est contenu dans un fichier. Il représente une *classe*, tout comme celles livrées avec JAVA que nous avons vu ci-dessus. En effet, vos programmes seront compilés en *.class* et seront utilisables par d'autres programmeurs ou par vous même.
Dès que vous souhaitez créer un programme en Java , il est indispensable de créer une classe , un fichier.

Le nom de la classe et le nom du fichier doivent être identiques. Le fichier aura .java comme extension. Par convention, vous prendrez garde à mettre une majuscule sur la première lettre du nom de la classe. Ceci parce qu'il est plus facile de différencier une instance d'un objet et la classe par cette majuscule.

C. Un premier programme

Voici un premier programme JAVA qui vous aidera à comprendre la structure du langage. Même si vous ne comprenez pas tout le code, cela ne vous empêchera pas de saisir l'importance de ce qui suit :

```
import java.lang.*;       // Importe tous les objets de la
                                    //    bibliothèque


public class MonPremierProgramme {

            public static void main ( String[] args ) {
                 System.out.println("Ceci est mon premier programme");
            }

}
```

Ce programme permet d'envoyer une chaîne de caractère à l'écran console. Le nom de la classe est *MonPremierProgramme* et il contient *une méthode*[3] spéciale nommée *main* dont nous reparlerons plus tard.

ATTENTION : JAVA est sensible à la casse, ce qui signifie que MonPremierProgramme et monPremierProgramme sont 2 objets différents. Il convient de faire extrêmement attention à la casse sous peine de se voir passer des heures inutiles et précieuses à corriger des erreurs de compilation.

C.1. Les classes

Une classe se déclare de la manière suivante :

```
<modificateur d'accès> class <Nom de la classe>
```

Ce qui donne dans notre exemple :

```
public class MonPremierProgramme
```

Ensuite, tout ce qui est contenu entre les accolades fait partie de la classe.

En résumé, la plupart des programmes JAVA ont cette structure. Il est important de se familiariser avec.

C.2. Les méthodes

Dans l'exemple, la méthode ***main*** , appartient à votre classe et peut-être utilisée comme une fonctionnalité supplémentaire de votre classe. Dans ce cas précis, la méthode ***main*** a une signification particulière qui permet à la classe *MonPremierProgramme* de n'avoir nul besoin d'être *instanciée*[4]. Ce concept sera détaillé plus loin.

[3] Une méthode est un ensemble d'instructions qui va s'exécuter dès que cette méthode est appelée. Elle appartient à la classe
[4] Instanciation : Opération qui consiste à créer une nouvelle référence d'un objet à partir d'une classe

C.3. Les modificateurs d'accès

Dans le programme précédent, nous avons parlé des modificateurs. Ils caractérisent la façon dont vous allez pouvoir accéder aux objets de la classe depuis l'extérieur. Nous reparlerons des modificateurs d'accès plus tard.

C.4. Compilation du programme

Lorsque ce programme est contenu dans le fichier *MonPremierPogramme.java*, il ne vous reste plus qu'a le compiler par la commande :

```
javac MonPremierProgramme.java
```

C.5. Exécution du programme

L'exécution du programme se fait avec la commande *java* mais sans préciser l'extension *.class* :

```
java MonPremierProgramme
```

C.6. Les commentaires

Présents dans le code pour des besoins de lisibilité et de compréhension, ils vont prendre toute leur importance lors de la maintenance et la modifications des programmes. Il existe 2 manières d'écrire des commentaires dans un programme :

```
// Commentaire sur une seule ligne

/** Commentaire
 *          sur plusieurs
 *          lignes    */
```

Avec l'une ou l'autre méthode, les caractères situés derrière les signes // ou /* ne seront pas lus par le compilateur. Attention cependant à ne pas mettre ces signes à l'intérieur d'un commentaire. L'exemple suivant produira des erreurs de compilations :

```
/* Le signe */ met fin au
            commentaire sur plusieurs lignes */
```

*Il existe un 3ème type de commentaire /** qui est utilisé par l'outil javadoc pour générer automatiquement la documentation au format HTML de vos programmes. Cet outil est assez puissant et sera étudié plus tard.*

D. L'API Java

Il s'agit de tout ce qui est fourni avec l'environnement de programmation *SDK* par *Sun*. *API* signifie « *Application Programming Interface* » qui, en bon français donne *Interface de programmation d'applications*. En un mot, l'API contient une vraie mine d'or pour les développeurs qui peuvent aller puiser dans ces ressources.

Les concepteurs du langage étant des développeurs chevronnés, ils sont sensibilisés aux problématiques inhérentes à la programmation. C'est pourquoi, ils ont développé des objets *clé en main* dont vous pourrez vous servir à volonté pour créer vos propres objets.
Par exemple, le « package » *java.sql* permet à vos programmes, d'accéder aux bases de données sans difficulté. Il en va de même pour développer des programmes réseaux avec le package « *java.net* »

D.1. Utilisation de la documentation API Java

Une documentation sur l'API Java est fournie sur le site internet de *Sun* (http://www.javasoft.com) au format *HTML* pour vous aider dans le développement d'applications Java.
Cette documentation, en anglais, vous sera vite indispensable pour connaître les réelles capacités des objets Java. Néanmoins, l'API Java est réellement utile lorsque l'on sait ce que l'on cherche. Vous apprendrez très vite à parcourir cette documentation lorsque vous serez à l'aise avec la programmation orientée objet que nous verrons ultérieurement.

D.2. Exemple de documentation

Voici une copie d'écran de la documentation sur l'objet *String :*

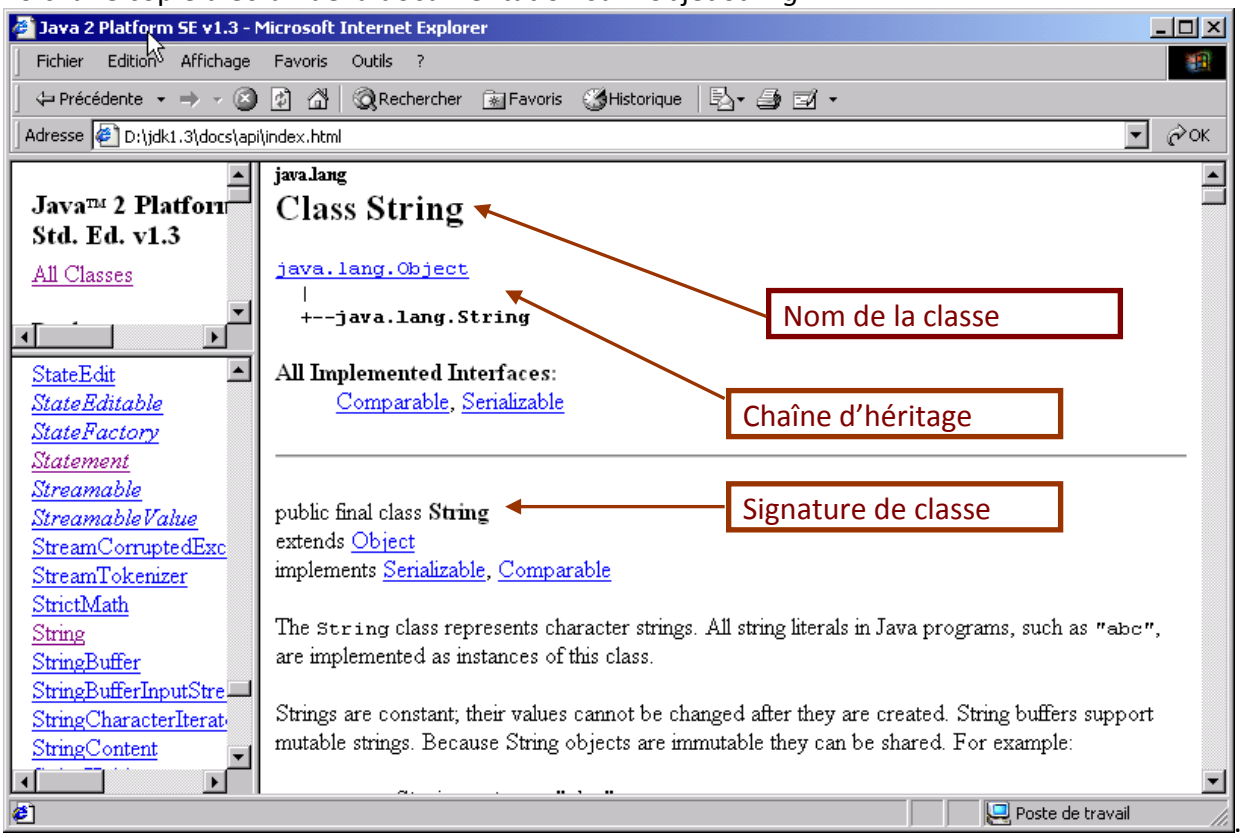


Schéma 2 – Documentation API de Java 2

Chapitre 3 : Types de données

A. Introduction

En Java, toute variable doit être déclarée et un espace mémoire doit être réservée pour stocker ses données. L'emplacement ne peut-être réservé que si le compilateur sait à quel type de variable il a à faire. Donc, la déclaration va spécifier le type et le nom de la variable, ainsi qu'un modificateur d'accès précisant si tout le monde peut y accéder ou non.
Il existe huit *type primitifs*[5], dont 6 six sont numériques (4 types d'entiers et 2 types de réels à virgule flottante) , 1 pour les caractères (*char*) et un booléen (*boolean*)

B. Les entiers

Voici un tableau représentant les 4 types d'entiers disponibles en JAVA :

Type	Taille en mémoire	Intervalles (limites incluses)
int	4 octets (32 bits)	**-2147483648** à **2147483647**
short	2 octets (16 bits)	**-32768** à **32767**
long	8 octets (64 bits)	**-9223372036854775808L** à **-9223372036854775807L**
byte	1 octet (8 bits)	**-128** à **127**

Pour déclarer ce type d'entiers, voici un panel d'exemples :

```
/* Exemples de déclaration d'entiers

int monEntier1 = 123478;  // Correct
int monEntier2 = 12.0 ;   // Incorrect , erreur de compilation

short monEntier3 = 32000; // Correct
short monEntier4 = 33000;      // Incorrect , erreur de compilation

long monEntier5 = 3000000000L ;     // Correct
long monEntier6 = 3000000000 ; // Incorrect , erreur de compilation

byte monEntier7 = -40;    // Correct
byte monEntier8 = 128;    // Incorrect , erreur de compilation
```

Les entiers longs ont un suffixe L (ex : 121020133L). Les entiers hexadécimaux sont codés ainsi :

```
int monEntierHexa = 0x12A ;
```

La façon dont sont codés les entiers ne dépend pas du système d'exploitation qui exécute le programme. C'est un des engagements de portabilité pris par les concepteurs du langage.

ATTENTION : Il ne faut pas confondre le type primitif int avec l'objet de JAVA Integer qui sont considérés différemment par le compilateur. Le premier est un type primitif alors que l'autre est une classe avec des méthodes, des champs et des constructeurs.

[5] Types prédéfinis par l'environnement JAVA comme les chaînes, les entiers ou les flottants

C. Les nombres à virgule flottante

Les deux types de nombres réels sont les *float* et les *double*

Type	Taille en mémoire	Intervalles (limites incluses)
float	4 octets (32 bits)	+/- **3.40282347^{E}+38F** environ
double	8 octets (64 bits)	+/- **1.797693133486231^{E}+308** environ

Il convient de faire très attention à la déclaration des ces nombres. En effet, si vous ne spécifiez pas le type flottant à la déclaration, c'est un *double* qui sera crée. Voici quelques exemples :

```
float monNombre = 12.34F ;          // F : précise qu'il s'agit d'un
// flottant

double monNombre2 = 13.45E14 ;
```

En général, la précision des float est insuffisante et vous préférerez utiliser des double qui se révèlent plus adaptés à la plupart des situations. L'utilisation des float peut être nécessaire dans le cas où la vitesse de calcul et l'encombrement de la mémoire sont prioritaires par rapport à la précision.

ATTENTION : Il ne faut pas confondre le type primitif double avec l'objet de JAVA Double qui sont considérés différemment par le compilateur. Le premier est un type primitif alors que l'autre est un objet avec des méthodes, des champs et des constructeurs.

D. Les caractères de type char

Les caractères se distinguent des chaînes par le biais de l'apostrophe. En effet, *'a'* signifie qu'il s'agit d'un caractère alors que *"a"* est une chaîne ne contenant qu'un seul caractère.
Le type *char* désigne des caractères en représentation *Unicode*[6]. Un caractère *Unicode* est constitué du caractère d'échappement *u* et d'un chiffre en hexadécimal sur 2 octets :

```
char monCaractere = '\u2122' ;      // Caractère ™ de marque déposée
```

Les caractères spéciaux sont classés dans le tableau suivant :

Caractère d'échappement	Signification	Valeur Unicode
\b	Effacement arrière	**\u0008**
\t	Tabulation horizontale	**\u0009**
\n	Saut de ligne	**\u000a**
\r	Retour chariot	**\u000d**
\"	Double apostrophe	**\u0022**
\'	Apostrophe	**\u0027**
\\	Antislash	**\u005c**

Pour tout savoir sur *l'Unicode* , **http://www.unicode.org .**

[6] Jeu de caractères offrant plus de possibilités que l'ASCII puisqu'il est possible de distinguer 65536 caractères différents contre 256 pour l'ASCII. Unicode est prévu pour coder l'ensemble des lettres de tous les pays

E. Les booléens

Le type primitif ***boolean*** peut avoir deux valeurs différentes : ***true*** ou ***false***. Il est employé chaque fois qu'un test conditionnel est effectué dans une structure de type *if…else* .

F. Les chaînes

Contrairement aux types précédents, il n'existe pas en JAVA, de type primitif gérant des chaînes de caractères. Par contre, JAVA propose un objet *String*. En conséquence, toute chaîne est une ***instance*** de la classe *String*.

```
String maChaine;
```

Les spécificités et les différentes manières d'utiliser des objets seront vus ultérieurement. Par conséquence, contentez-vous, pour l'instant, d'utiliser les exemples suivants sans chercher à comprendre le mécanisme des objets. Sachez que tout objet peut avoir des méthodes et que ces méthodes sont des sous programmes permettant de tirer partie du "savoir – faire" de l'objet.

F.1. Concaténation

L'opérateur + permet de concaténer des chaînes entre elles.

```
String maChaine = "Bonjour";

maChaine = maChaine + " à tous";
```

L'exemple ci-dessus permet de concaténer la chaîne *maChaine* avec la chaîne *" à tous"*. La chaîne résultante sera égale à : "*Bonjour à tous*"

F.2. Sous-chaînes

L'objet *String* dispose de ***méthodes*** très utiles pour extraire des morceaux de chaînes. La méthode *substring* s'utilise comme suit :

```
objetChaine.substring( int debut , int fin );
```

- **debut** : entier représentant le premier caractère inclus à prendre en compte pour la nouvelle chaîne. Le premier caractère d'une chaîne a l'indice 0
- **fin** : entier représentant le dernier caractère exclus à prendre en compte pour la nouvelle chaîne.

```
String maChaine = "Bonjour à tous";
String maSousChaine;

maSousChaine = maChaine.substring(0,7); // Donne "Bonjour"
```

La variable *maSousChaine* contiendra "Bonjour". En fait le premier caractère de la chaîne a l'indice 0. Dans l'exemple , le 2ème argument de *substring* est l'indice du premier caractère à **exclure** du résultat.

F.3. Manipulation de chaînes

Dans de nombreux cas, vous souhaiterez connaître la longueur d'une chaîne avant d'effectuer des manipulations. Il vous suffit d'utiliser la méthode *length()* pour l'obtenir :

```
String maChaine = "Bonjour";
int longueur;

longueur = maChaine.length();  // longueur contient 7
```

Contrairement à C++ , il n'est pas possible de modifier un caractère d'une chaîne en manipulant la chaîne comme un tableau (En JAVA, l'objet *String* n'est pas un tableau).
Par contre, la méthode *charAt()* permet de lire n'importe quel caractère de la chaîne :

```
String maChaine = "Bonjour à tous";
char c;

c = maChaine.charAt(3);   // c contient 'j'
```

F.4. Référence à un objet chaîne

Pour transformer la chaîne "*Bonjour à tous*" en "*Bonsoir à tous*" , vous pouvez néanmoins utiliser le code suivant :

```
String maChaine = "Bonjour à tous";

maChaine = maChaine.substring(0,3) + "soir" +
                                          maChaine.substring(7,14);
```

La variable *maChaine* fera référence à l'objet renvoyé par la méthode *substring* appliquée à l'objet *maChaine* lui-même.

Chapitre 4 : Les variables et opérateurs

A. Introduction

Comme nous l'avons précisé précédemment, toute variable doit être déclarée avant d'être utilisée. Par convention, la 1ère lettre du nom des variables n'est pas une majuscule. Cela permet de distinguer d'un seul coup d'œil une classe d'une instance d'un objet. Nous verrons les objets et les classes ultérieurement.

B. Déclaration d'une variable

B.1. Exemples

Voici quelques exemples de déclaration de variables :

```
int monEntier;

float monFlottant = 12.45F ;   // Déclaration et affectation

char monCaractere = 'z';
```

B.2. Noms des variables

Vous noterez que le nom des variables ne peuvent être des mots clés du langage comme *char , int, class …* . Ces noms peuvent être constitués de n'importe quel caractère *Unicode* contrairement à la plupart des autres langages. A l'exception de tous les caractères tels que *@ ,* les espaces , *+ ,* - ou encore */* qui ont une signification particulières dans les programmes.

Une astuce pour savoir si un caractère Unicode est considéré par JAVA comme un caractère valide pour les variables, consiste à utiliser les méthodes isJavaIdentifierStart et isJavaIdentifierPart de la classe Character.

C. Conversions de types

Dans beaucoup de situations, il peut être très utile d'effectuer des opérations mathématiques entre des types de données différents. Cela ne pose aucun problème à JAVA et lorsque les types de données mis en jeu sont différents, c'est toujours le plus précis qui l'emporte.

C.1. Conversion implicite

Elle se produit lorsque le compilateur décide lui-même, en fonction de règles précises, d'effectuer la conversion

Voici un exemple :

```
int a = 3;
float b = 4.3F;

b = b * a ;      // b*a donne un flottant ➔ résultat correct
```

```
a = b * a ;      // Incorrect car il y a perte de précision
                          // la compilation ne se fera pas
```

Dans le 1er exemple, une conversion de l'entier *a* est effectuée de telle manière que les 2 parties de l'opération contiennent des types identiques.

C.2. Conversion explicite

Il peut arriver que vous ayez besoin d'effectuer une conversion explicite soit pour éviter des erreurs de compilation en acceptant la perte de précision, soit pour des raisons inhérentes à votre programme. Cette opération se nomme le *transtypage* et est très utilisée en JAVA.
Voici l'exemple précédent qui ne se compilait pas et qui, grâce au *transtypage* , se compile sans problèmes :

```
int a = 3;
float b = 4.3F;


a = (int) b * a ;          // b est converti en entier avant l'opération
                                 // Il y a cependant une perte d'information
```

Dans cet exemple, le résultat sera *12* car la partie décimale de *b* a été éliminée.

C.3. Cas particulier de résultat tronqué

Il est intéressant de voir ce qu'il se passe lorsque le résultat de la conversion dépasse les dimensions du type du résultat :

```
short a;
int b= 53456;

a = (short) b;   // Conversion de b en short
```

Dans cet exemple, le résultat est tronqué car une variable de type *short* n'excède pas *32767*. Le résultat est alors tronqué et donne *–12080* .

D. Constantes

Pour déclarer une constante en JAVA, il suffit d'employer le modificateur *final* devant la déclaration de la variable.

```
final double EURO = 6.55957;        // Taux de change pour le franc
```

Par convention, les constantes sont en majuscules.

On utilise les modificateurs static et final devant une variable pour définir une constante de classe. Cette constante sera alors utilisable directement sans besoin d'instancier la classe. Les notions de classe et d'instanciation seront vues plus loin.

E. Opérateurs

Il existe plusieurs types d'opérateurs en JAVA :

- Opérateurs arithmétiques
- Opérateurs binaires
- Opérateurs d'incrémentation et de décrémentation
- Opérateurs relationnels

E.1. Opérateurs arithmétiques

Il en existe 5 :

- + : addition
- - : soustraction
- * : multiplication
- / : division
- % : reste de la division
-
- Voici quelques exemples :
-

```
int a = 23 + 12 ;          // a contient 35
int b = a * 2;        // b contient 70
int c = b / 3;        // c contient 23
double d = b / 3;          // d contient 23.0
double e = b / 3.0;   // e contient 23.333333333333332

int f = 50 % 12;      // f contient le reste de la division = 2
```

E.2. Opérateurs d'incrémentation

Ces opérateurs permettent d'effectuer des opérations arithmétiques prédéfinies pour incrémenter (ajouter 1) ou décrémenter (retirer 1) des variables. Selon l'endroit où se situe l'opérateur (avant ou après le nom de la variable), l'opération est effectuée avant ou après l'évaluation de l'expression.

Par exemple, pour incrémenter la variable *x* :

```
x++ ;                    // Incrémentation de x
```

ou

```
x--;                     // Décrémentation de x
```

Toute l'utilité des ces opérateurs réside dans le fait qu'ils peuvent être utilisés à l'intérieur d'une expression.
Par exemple, pour incrémenter la variable *x* après avoir affecté la variable z :

```
int x=10;
int z;
z = x++;                 //   z=10 après l'opération
                         //   x=11 après l'opération
```

Dans cet exemple, nous parlerons d'une *post incrémentation* étant donné que l'incrémentation s'effectue après que le contenu de *z* soit affecté au contenu de *x*.
La même opération mais cette fois-ci en pré incrémentation aurait donné :

```
int x=10;
int z;
z = ++x;                 //   z=11 après l'opération
                         //   x=11 après l'opération
```

Voici des exemples plus complexes :

```
// Fichier ex2.java

int a = 10;
int b = 13;

int resultat1, resultat2;

resultat1 = 3 * ++a;       // resultat1 = 33 , a=11 après l'opération
resultat2 = a++ * --b;          // resultat2 = 132 , a=12 , b=12 après
                                     // après l'opération
```

Exemple 2

L'emploi de ces opérateurs ne doit pas être démesuré. La première raison est qu'ils nuisent à la lisibilité du code. La deuxième est qu'il est source d'erreurs de la part du programmeur qui peut, dans des cas complexes, évaluer de manière incorrecte le contenu des variables dans une opération.

E.3. Opérateurs relationnels

Ils permettent de comparer des variables entre elles . Le signe == compare les variables situées de chaque côté du signe. Le résultat de la comparaison est un booléen dont les deux valeurs possibles sont *true* (Vrai) ou *false* (Faux) .
Par exemple :

```
(3 == 2)                 // Le résultat est false
(3 > 1)                  // Le résultat est true
```

Certains langages comme Visual Basic ne font pas la distinction entre = et ==. En JAVA, le fait d'employer deux signes différents évite les confusions. Le premier sert pour l'affectation alors que l'autre est utilisé pour le test d'égalité

Voici les principaux opérateurs relationnels

Opérateur	Signification
< , >	Inférieur à , supérieur à
<= , >=	Inférieur, supérieur ou égal à
== , !=	Egal à , différent de
&& , ||	et , ou logique

E.4. Opérateurs binaires

Ces opérateurs permettent d'effectuer des opérations binaires sur des variables de type entier. Les opérations binaires peuvent s'avérer utiles lorsque l'on manipule des données au niveau machine (registres internes d'un processeur, mémoire …)
Il existe des techniques de masquage de bits basées sur l'opérateur binaire *&*. Dans l'exemple suivant, les opérations binaires effectuées permettent d'isoler un bit et de récupérer sa valeur.

```
int a=9;              // 9 en décimal vaut 1001 en binaire
int b,c;

b = a & 4;       // ET entre 1001 et 0100 permet de masquer tous les
                   // bit sauf le troisième
                   // Le résultat vaut 0000
c = a & 8        // ET entre 1001 et 1000 permet de récupérer le 4ème
                     // bit qui vaut 1
                        // Le résultat vaut 1000
```

De manière générale, effectuer un ***ET* logique** entre un nombre binaire et 2^n permet de récupérer le n^{ième} bit de ce nombre.
Les opérateurs de décalage sont très utiles pour manipuler des nombres binaires. La pratique du *décalage à droite* ou du *décalage à gauche* sont assez courantes.

```
int a=10;   // a vaut 1010 en binaire
int b;

b = a >> 1  // b vaut 101 en binaire soit 5 en décimal
b = a << 2   // b vaut 101000 en binaire soit 40 en décimal
```

Voici un exemple de programme permettant de savoir si le bit n d'un entier est à 1 ou non. Si le résultat est égal à 1 alors le bit n est à 1 :

```
/* Programme permettant de tester le bit 2 d'un nombre
   Le programme affiche 1 si le bit est à 1 et 0 s'il est à zéro
*/
public class Ex3 {
public static void main (String[] args){
            int nombre=45;     // 101101 en binaire
            int resultat;


            resultat = (nombre & 4) >> 2; // Le ET logique met tous
                                          // les bits
                                          // à 0 mais conserve le bit 2
                                          // et décale le bit conservé
                                          // de 2 rangs vers la droite

            System.out.println("resultat="+resultat); // Affiche
                                                     // le résultat
 }
}
```

Exemple 3

Le programme de l'exemple 2 illustre la facilité d'emploi des opérateurs pour obtenir une imbrication des opérateurs dans une expression.

CONSEIL : Ne vous attardez pas sur la structure du programme et sur l'utilisation des classes d'affichage de l'exemple 2. Vous pouvez taper et compiler ce programme. Il fonctionnera sans problème.

E.5. Hiérarchie des opérateurs

Comme dans tous les langages de programmation, il est préférable d'employer des parenthèses pour indiquer l'ordre dans lequel les opérations doivent être accomplies. Voici un tableau récapitulant cette hiérarchie en JAVA:

Ordre	Opérateurs
1	[ ] , ( )
2	! , ++ , --
3	* , / , %
4	+ , -
5	<< , >> ,>>>
6	< , <= , > , >=
7	== , !=
8	&
9	^ (ou exclusif)
10	|
11	&&
12	||
13	?:
14	= , += , -= , *= , /= , &= , |= , <<= , >>= , >>>=

En résumé, si une expression contient plusieurs opérateurs présents dans ce tableau, chaque opération sera réalisée avec la priorité correspondante.
Voici quelques exemples :

```
int a,b,c,d;

a = 3;

b = 2 * a + 1;        // * prioritaire devant le + donc b vaut 7
c = 7 & 2*a;              // * prioritaire devant le & donc c vaut 6
d = a << 2 & 5;       // << prioritaire devant & donc d vaut 4
```

Chapitre 5 : Les flux d'exécution

Les flux d'exécution représentent la possibilité pour le programmeur de réaliser des boucles ou des test conditionnels. Tous les langages permettent de manipuler ces flux et JAVA n'échappe pas à cette règle

A. Bloc d'instructions et portée

Avant de parler de boucles, il faut bien comprendre ce que représentent un bloc d'instructions et la portée des éléments qu'il contient.
Un bloc est un ensemble d'instructions délimitées par des accolades. Les exemples vus précédemment contenaient des blocs :

```
public class Ex3 {

            public static void main (String[] args){


            int nombre=45;      // 101101 en binaire
            int resultat;


            resultat = (nombre & 4) >> 2; // Le ET logique met
                                          // tous les bits
                                        // à 0 mais conserve le bit 2
                                        // et décale le bit conservé
                                        // de 2 rangs vers la droite

            System.out.println("resultat="+resultat); // Affiche le résultat
 }
}
```

Toutes les instructions surlignées en gris appartiennent au bloc d'instructions de la méthode ***main***.
En dehors de ce bloc, les variables n'existent plus. Ce qui signifie que *resultat* et *nombre* ne sont pas reconnues ailleurs que dans leur bloc.
Dans l'exemple suivant, une erreur de compilation se produira :

```
public class Ex5 {

            public static void main (String[] args){


            int a;          // 101101 en binaire

                 {
                 int b = 3;
                 a = b * 2:
                 }
                 a = b + 7 ;    // Erreur car b n'est pas connu dans
                                // ce contexte
            }
}
```

La portée de la variable *b* est donc limitée , par contre celle de *a* est étendue à la méthode ***main***

ATTENTION : Il n'est pas possible en JAVA, de redéfinir une variable dans un bloc imbriqué si elle a déjà été définie dans le bloc de hiérarchie supérieure. Voici un exemple illustrant cette restriction

```
{
int r;
            {
            int r;          // ERREUR, r a déjà été définie au dessus
            int n;
            }
}
```

B. Les instructions conditionnelles

B.1. Instruction if … else

Il s'agit d'une des instructions la plus employée en JAVA. Elle permet d'effectuer une instruction ou un bloc d'instructions selon des conditions précises.
La forme la plus employée est :

```
if (condition)
            {
            bloc d'instructions
            }
```

Si la condition est vraie, le bloc d'instructions est exécuté sinon, le programme passe à la suite.
Voici une illustration de la forme :

```
if ( nombreDePassagers > 7 )
            {
            surcharge = true;
            mise_en_marche = false;
            }
```

Dans cet extrait de code, la mise en marche n'est faite que si le nombre de passagers est inférieure ou égale à 7.
else permet d'effecteur un autre bloc d'instruction au cas où la condition n'est pas vraie :

```
if ( nombreDePassagers > 7 )
            {
            surcharge = true;
            mise_en_marche = false;
            }
else
            {
            surcharge = false;
            mise_en_marche = true;
            }
```

Attention : Lorsque le bloc d'instruction ne contient qu'une seule instruction, vous pouvez omettre les accolades. Cependant, cela ne doit pas vous faire oublier que l'instruction qui suit ne fait pas partie du même bloc. Voici un exemple :

```
if (nombreDePassagers>7)
            surcharge = true;
            mise_en_marche = false; // Exécuté quoiqu'il arrive, même
                                    // si les passagers sont moins
                                    // que 7
```

Les imbrications d'instructions *if … else* sont également possibles :

```
if (solde<0)
            message = "Vous êtes débiteur";
else if (solde == 0)
            message = "Vos comptes sont à zéro";
else if (solde >= 10000)
            message = "Voulez-vous épargner ?";
```

B.2. Boucles *While*

Les boucles *While* permettent d'exécuter des blocs d'instructions de manière répétitives et cela, tant qu'une condition est vraie. Voici la forme de cette boucle :

```
while ( condition )
            {
            bloc d'instructions
            }
```

Cette boucle n'exécute pas le bloc d'instructions si la condition n'est pas vraie.
Voici un programme permettant le calcul x^n et stockant le résultat dans une variable :

```
public class Ex6 {

            public static void main (String[] args){


            int x=2, resultat=1;
            int n=12;

            int i=n;

            while ( i>0)                // Tant que I supérieur à 0
                 {
                 resultat *= x;              // resultat = resultat * x
                 i--;                        // Décrémentation
                 }
            System.out.println("Resultat="+resultat);// 4096
            }
}
```

Exemple 6

B.3. Boucles *for*

Elle permet d'effectuer une boucle avec un nombre d'itérations (passages dans le bloc) défini à l'avance.

Voici la forme générale :

```
for ( instruction1 ; expression1 ; expression2 )
            {
            bloc d'instructions
            }
```

- **instruction1** : initialisation d'une variable entière servant à donner l'état du nombre de passages effectués.
- **expression1** : condition de continuation de la boucle. Si la condition n'est plus vérifiée, la boucle s'arrête
- **expression2** : C'est l'instruction qui va permettre d'affecter le compteur

La boucle suivante effectue 10 itérations en affichant le décompte :

```
for (int i=10 ; i>0 ; i-- ) {
            System.out.println(i);

}
System.out.println("Boum !!!");
```

Vous auriez pu faire la même chose avec une boucle *while*

```
int i=10;

while (i>0)
            {
            System.out.println(i--);
            }

System.out.println("Boum !!!");
```

B.4. Instruction Switch-case

Lorsque le nombre d'états différents à tester d'une variable est trop important, il est préférable d'utiliser la structure ***switch..case..break***. L'utilisation de ce type de d'instruction est assez délicate mais rend bien des services notamment lors de la gestion des touches du clavier par exemple.

a. Structure

Le sélecteur, c'est à dire la variable sur laquelle est effectué le test, est situé derrière le mot ***switch*** **.** Ensuite, vient le bloc d'instructions ***case***. Chaque test effectué débute par ce mot clé. La présence de ***default*** permet de gérer les instructions à exécuter si aucun des cas n'est rencontré. Voici la structure complète :

```
switch selecteur {

            case valeur1 : instruction1;
                           instruction2;
                           // …
            case valeur2 : instruction3;
                           // …
            default : instruction4;

}
```

ATTENTION : Dès que la valeur testée correspond au contenu de la variable, TOUS les autres blocs CASE sont exécutés et ceci quel que soit la valeur testée. Dans le cadre d'une utilisation normale, il vaut mieux utiliser l'instruction break

b. Structure avec break

break permet de terminer le bloc ***switch*** sans autre forme de procès. Cela évite donc d'effectuer tout autre bloc case :

```
switch selecteur {

            case valeur1 : instruction1;
                           instruction2;
                           // …
                           break;
            case valeur2 : instruction3;
                           // …
                           break;
            default : instruction4;
                      break;       // Inutile puisque la dernière

}
```

c. Type des variables

Les variables pouvant être utilisées avec cette instruction sont les « ***char », « int », « byte »*** ou « ***short »***.

d. Exemples

```
int n=1;

switch n {

            case 0 :
                 System.out.println("Valeur nulle");
            case 1 :
                 System.out.println("Valeur positive");
            case 2 :
                 System.out.println("Valeur supérieure à 1");
}
```

L'affichage donnera :

```
Valeur positive
Valeur supérieure à 1
```

Avec break :

```
int n=1;

switch n {

            case 0 :
                 System.out.println("Valeur nulle");
                 break;
            case 1 :
                 System.out.println("Valeur positive");
                 break;
            case 2 :
                 System.out.println("Valeur supérieure à 1");
}
```

```
Valeur positive
```

L'utilisation de switch sans break est plus rare mais il peut être très utile de recourir à une telle structure pour la gestion des touches notamment.

Chapitre 6 : Les tableaux

Les tableaux sont très utilisés dans tout bon langage de programmation qui se respecte. Ils permettent de stocker les données dans un ensemble et d'accéder à ces données par l'intermédiaire d'indices. En JAVA, l'utilisation des tableaux est très simple. En règle générale, mieux vaut ne pas se préoccuper de la façon dont JAVA gère les tableaux mais plutôt de la manière des les utiliser.

A. Un tableau est un objet

Un tableau se comporte exactement comme un objet. Même si, à ce stade du cours, vous ne connaissez pas les objets, considérez les tableaux comme des objets pouvant contenir d'autres objets.

En JAVA, il n'est pas facile de changer la taille d'un tableau en cours de programme. Pour cela, il vaut mieux utiliser la classe Vector ou ArrayList.

B. Déclaration d'un tableau

Pour déclarer un tableau, vous pouvez utiliser plusieurs syntaxes. L'exemple suivant montre comment déclarer un tableau d'entiers :

```
int [] monTableau = new int[50];
```

ou

```
int monTableau[] = new int[50];
```

Vous remarquerez la présence de l'opérateur *new* qui sert à créer le tableau avec 50 entiers. Le premier élément du tableau a un indice de 0 et le dernier un indice de 49.
Pour changer la valeur du 43ème élément, vous utiliserez la syntaxe suivante :

```
monTableau[42] = 14;
```

Voici un exemple de tableau ayant 7 éléments indicés de 0 à 6 :

```
int[] monTableau =
          { 12,5,34,7,15,23,2 };
```

ou

```
int[] monTableau = new int[7];

monTableau[0] = 12;
monTableau[1] = 5;
monTableau[2] = 34;
…
```

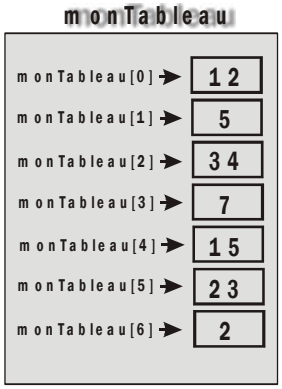


Schéma 3 – Tableau d'entiers

C. Tableaux d'objets

Un tableau peut contenir des objets. Il faut alors créer chaque objet séparément en utilisant l'opérateur ***new***.

Info C/C++ : L'opérateur new servant à allouer dynamiquement de la mémoire en C/C++ n'a pas la même signification en JAVA. En effet, cet opérateur est appelé pour toute instanciation d'objet en JAVA

C.1. Tableaux de *String*

Voici un exemple de programme permettant de déclarer un tableau de chaînes :

- **tab** : tableau de *String* contenant 5 éléments indicés de 0 à 4
-

Ensuite, le programme effectue une boucle qui stoppe lorsque la fin du tableau est atteinte. Ceci est rendu possible grâce à la propriété *length* de l'objet tableau

```
public class Ex7 {

            public static void main (String[] args){

            String [] tab = { "Chaine 1" , "Chaine 2" , "Chaine 3" };

            // Boucle chargee d'afficher tab2
            for (int i=0 ; i< tab.length ; i++ )
                 System.out.println("tab["+i+"] = "+tab[i]);
            }
}
```

Exemple 7

C.2. Tableaux d'*Object*

Le processus est le même pour manipuler des tableaux d'objets quelconques.

ATTENTION : Les tableaux, en JAVA, sont des références cachées. Ce qui signifie qu'un objet déclaré comme un tableau est en fait une référence à cet objet et non l'objet lui-même. L'exemple de copie de tableau qui suit propose une mise en pratique de cette particularité de JAVA.

D. Copie de tableaux

D.1. Introduction

La copie de tableaux peut-être très utile lorsqu'on souhaite trier ou effectuer des opérations de suppression ou de modification d'une partie du tableau.
Il paraîtrait naturel, pour faire une copie de tableau de faire comme dans l'instruction suivante mais hélas il n'en est rien :

```
int[] tab1 = {3,5,7};

int[] tab2 = tab1;

tab2[1] = 6;          // tab1[1] vaudra 6 et non plus 5
```

Dans cet exemple, vous constaterez que *tab1* et *tab2* font référence au même objet. Cela signifie que si vous modifiez un des éléments de *tab2* , vous modifiez par la même occasion celui de *tab1*.
Dans le dessin suivant , vous avez 2 références qui pointent sur le même objet en mémoire :

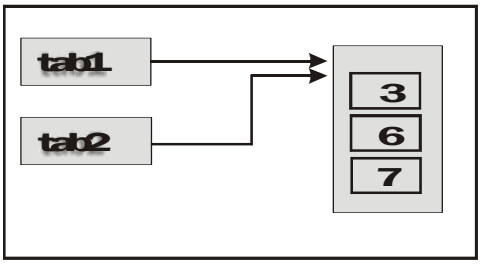


Schéma 4 – références aux objets

Pour plus d'explications sur les objets et les références, voyez le chapitre H.II Références aux objets

D.2. Méthode arraycopy

La classe *System* de l'API JAVA dispose d'une méthode *arraycopy* qui s'utilise comme suit :

```
System.arraycopy (source,indiceSource,cible,indiceCible,compte)
```

- **source** : Tableau source
- **indiceSource** : Indice à partir duquel on va commencer à copier
- **cible** : Tableau cible
- **indiceCible** : Indice à partir duquel on va commencer à écraser les éléments du tableau cible.
- **compte** : Nombre d'éléments du tableau source à copier

Voici un exemple de programme permettant de recopier le tableau *source* dans le tableau *cible* à partir de l'élément d'indice 2 de *cible*.

```
public class Ex8 {

            public static void main (String[] args){

            int [] source ={ 1,2,3,4,5 } ;
            int [] cible = { 0,0,0,0,0,0,0,0,0 };

            System.arraycopy(source ,0 , cible , 2 , 5 );


            // Boucle chargee d'afficher cible

            for (int i=0 ; i< cible.length ; i++ )
                 System.out.println("cible["+i+"] = "+cible[i]);
            }
}
```

Exemple 8

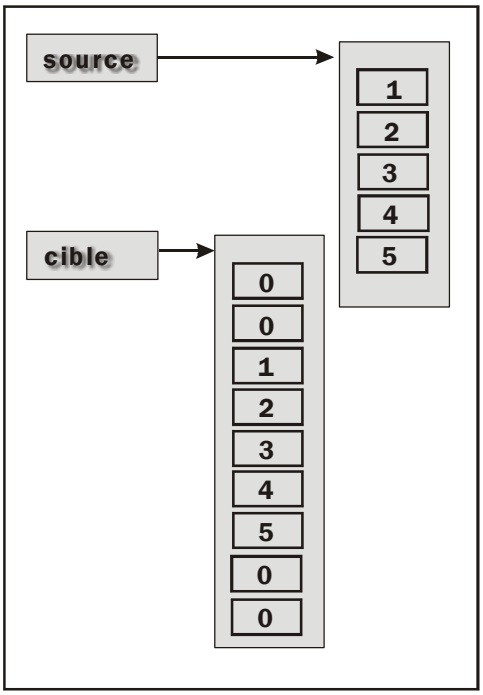


Schéma 5 – Copie de tableaux

Chapitre 7 : La programmation orientée objet

Ce chapitre propose une introduction à la programmation orientée objet en JAVA puis s'appuiera sur des objets JAVA existants dans les API pour argumenter la discussion autour de ce sujet incontournable.
N'oubliez pas que JAVA est un langage purement orienté objet et qu'il respecte les 3 règles fondamentales de la POO[7] :

- Encapsulation
- Héritage
- Polymorphisme

Derrière ces mots "barbares", se cache toute un attirail de règle et de vocabulaire qu'il va falloir apprendre avant de concevoir ses propres programmes en JAVA.
Ne sacrifiez pas la lecture de ce chapitre même si vous avez quelques notions de POO de part votre expérience dans d'autres langages. JAVA respecte les principes de la POO , ce qui n'est pas le cas de tous les langages.

A. Pourquoi autant de succès ?

Aujourd'hui , la POO fait quasi l'unanimité dans le monde des langages de développement. Elle remplace les techniques de programmation procédurale qui ont vu leur paroxysme dans les années 70. Le *basic* ou le *pascal* ont quasiment disparus pour laisser place à des langages de manipulation d'objets.
Avant, les développeurs étaient obligés de se concentrer d'avantage sur les algorithmes en proposant des structures de données complexes et plus difficiles à construire. Avec les objets, le travail sur l'algorithme est minimisé. Leur souplesse permet aux développeurs d'aller beaucoup plus loin dans la réalisation et la conception de programmes "intelligents" et modulaires.

B. Les objets

B.1. Le rôle d'un objet

Le développeur conçoit ses objets avec une idée bien précise sur l'utilisation qui en sera faite. Cet objet aura des fonctionnalités et des tâches à effectuer. En dehors de ces tâches, l'objet ne sait rien et se contente de faire ce qu'on lui demande.
Le temps où les développeurs refaisaient le monde à chaque nouveau programme est révolu. Aujourd'hui les objets peuvent fonctionner ensemble même si leur concepteur est différent.
Ne perdez jamais de vue qu'en tant que programmeur JAVA, vous devez toujours penser à la réutilisabilité de vos objets.
Les objets sont conçus de telle manière que le développeur n'a pas à se soucier de la manière dont ils sont implémentés. Lorsque vous démarrez votre voiture, peut vous importe de connaître la façon dont le déplacement des pistons est géré dans le moteur. La fonction principale de votre voiture est de se déplacer d'un point A vers un point B. Il en va de même pour les objets.
Reprenons l'exemple de l'automobile en imaginant de créer un objet appelé *Voiture*. En tant qu'utilisateur de cet objet, votre soucis est de trouver les fonctionnalités suivantes :

- Démarrer
- Avancer

[7] Programmation Orientée Objet

- Reculer
- Tourner
- Accélérer
- Freiner

Pour les objets, vous appliquerez le même principe. Vos objets auront des fonctionnalités et des savoir-faire que d'autres objets pourront réutiliser , ce qui leur évitera de recréer eux-mêmes.

B.2. Caractéristiques d'un objet

Un objet se caractérise par :

- **Un état** : Caractérise l'état dans lequel est l'objet (Le moteur de la voiture est-il démarré ? La voiture est-elle à l'arrêt ?
- **Un comportement** : Caractérise ce que l'objet est capable de faire
- **Une identité unique** : Précise son caractère unique. Un objet possède une identité propre, ce qui permet de le distinguer parmi les autres (Comme les humains)
-

C. Les classes

C.1. Introduction

La notion de classe, est sans doute la plus importante à connaître. Une classe est un modèle, une maquette ou un moule à partir de laquelle les objets vont être construits.

Ne confondez jamais une classe et un objet. L'objet est une instance de la classe qui représente son modèle.

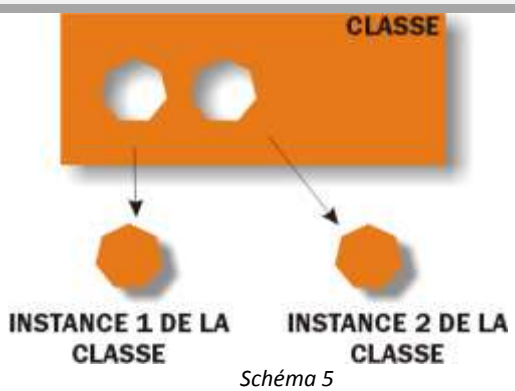


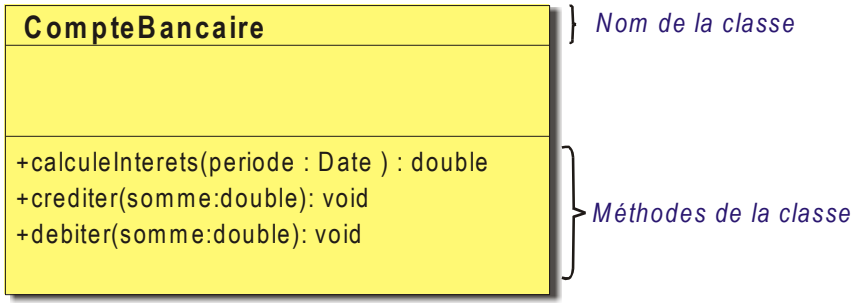
Schéma 5

Comme le montre le schéma précédent, l'opération qui consiste à sortir le nouvel objet de son moule est appelé *instanciation*. A ce moment, une place en mémoire est réservée pour l'objet par le compilateur. Comme le compilateur sait, grâce au modèle ce qu'est la classe, la taille de l'objet, il sera à même de réserver l'espace nécessaire dans la mémoire.

C.2. Les Méthodes de classe

Ce sont les éléments les plus intéressants puisqu'il s'agit de fonctions intégrées à la classe qui peuvent être appelées par l'utilisateur d'une instance de la classe selon les règles de *l'encapsulation* (voir plus loin dans ce chapitre).
Les méthodes définissent le *comportement* de l'objet.

Par exemple, imaginons une classe *CompteBancaire*. Une méthode *calculeInterets* qui renverrait le montant total des intérêts pour une période donnée serait très utile :



Les méthodes peuvent accepter des données en paramètres mais peuvent également renvoyer des données. Dans notre exemple, la méthode calculeInterets renvoie le montant total des intérêts avec l'argument periode représentant la période de calcul.

a. Le passage de types primitifs

Lorsque l'on passe des paramètres à une méthode, le passage par valeur consiste à faire une copie de la variable. Par conséquent, la méthode ne risque pas de modifier la variable utilisée lors de l'appel. Voici un exemple qui défini une classe *Classe* et un programme principal chargé d'appeler la méthode *additionner* :

```
public class Classe {


            public int additionner(int nombre1,int nombre2) {
                 nombre1 += 1;    // Modification du 1er paramètre
                 return (nombre1+nombre2);
            }
}
```

Voici le programme appelant :

```
public class Main {

            public static void main (String[] args) {

            Classe objet = new Classe();

            int a = 1;
            int b = 1;
            int resultat;

            System.out.println("a =" + a);
            System.out.println("b =" + b);

            resultat = objet.additionner(a,b);

            System.out.println("Operation effectuee");
            System.out.println("a =" + a);
            System.out.println("b =" + b);
            System.out.println("resultat =" + resultat);
            }
}
```

Les variables *a* et *b* restent inchangées même si on essaie de modifier ces valeurs à l'intérieur de la méthode.

b. Le passage d'objets

ATTENTION : Le passage par valeur est systématique pour les types primitifs tels que int mais le passage d'objets (ou plus précisément de références à des objets) est tout à fait différent. En effet , nous savons qu'une référence n'est pas un objet mais une sorte d'étiquette qui pointe vers l'objet lui-même. Or , deux références peuvent pointer vers un même objet. Que se passerait-il si on essayait de modifier un objet passé en paramètre ? Voici un exemple à 3 classes

```
public class Chaine {

private String chaine;

            public Chaine(String uneChaine) {
                 this.chaine = uneChaine;
            }

            public String getChaine() {
                 return this.chaine;
            }

            public void setChaine(String uneChaine) {

                 this.chaine = uneChaine;
            }

}
```

Chaine.java

Voici la classe chargée d'appeler les méthodes de chaine :

```
public class Classe {

            public String concatener(Chaine chaine1,Chaine chaine2) {
                 chaine1.setChaine(" xxxxx ");   // Modification
                 return chaine1.getChaine()+chaine2.getChaine();
            }
}
```

Classe.java

Et enfin le programme principal :

```
public class Main {

            public static void main (String[] args) {

            Classe objet = new Classe();

            Chaine ch1 = new Chaine("Il fait  ");
            Chaine ch2 = new Chaine("beau");
            String resultat;

            System.out.println("ch1 =" + ch1.getChaine());
            System.out.println("ch2 =" + ch2.getChaine());

            resultat = objet.concatener(ch1,ch2);

            System.out.println("Operation effectuee");
            System.out.println("ch1 =" + ch1.getChaine());
```

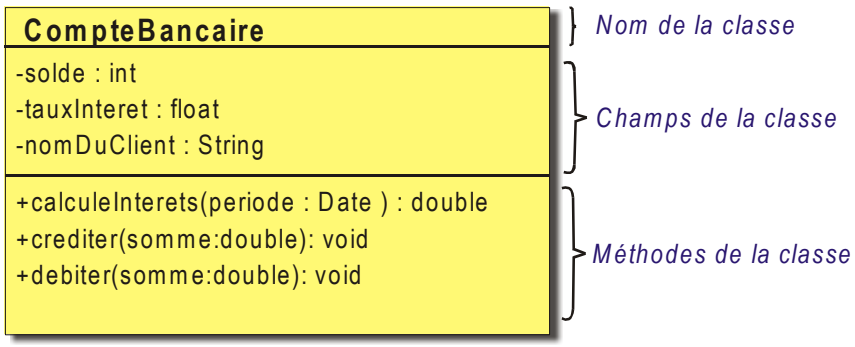
```
            System.out.println("ch2 =" + ch2.getChaine());
            System.out.println("resultat =" + resultat);
            }
}
```

Main.java

Vous constaterez que l'objet *ch1* a bien été modifié, ce qui peut poser beaucoup de problèmes dans un programme manipulant de nombreux objets.

C.3. Les champs ou données de la classe

Les champs ou données de la classe permettent de stocker *l'état* de l'objet. Pour notre classe *CompteBancaire* , le champ le plus intéressant sera sans doute celui qui contiendra le solde du compte :



D. Les 3 règles fondamentales de la POO

D.1. Encapsulation de données

C'est la caractéristique permettant à un objet de protéger ses données de l'extérieur et de contrôler leur manipulation par les autres objets.
Les modificateurs de type dont vous avez eu un aperçu au chapitre B permettent de fixer des règles quant à l'accès aux données.

D.2. Héritage

Vos classes sont construites à partir d'autres classes. Nous avons tous un patrimoine génétique issu de nos parents qui font que notre identité est unique , les classes et donc les objets aussi.

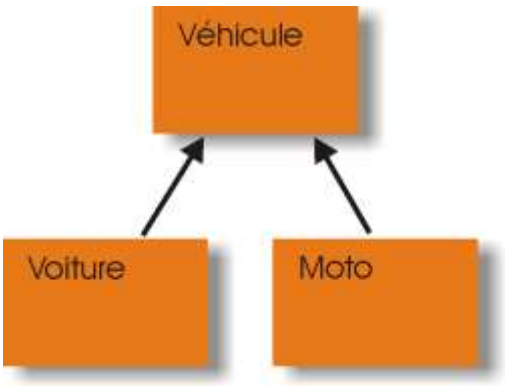


Schéma 6 – Exemple d'héritage

Dans le schéma 6, les classes *Voiture* et *Moto* héritent de la classe *Vehicule*. Nous pourrions même aller plus loin dans la démarche en considérant que les classes *Berline* et *Sportive* héritent de la classe *Voiture*.

La classe parent est appelée la superclasse

Certains langages comme le C++ supporte *l'héritage multiple* qui permet à une classe d'hériter de plusieurs classes parents. En JAVA, il n'y a pas d'héritage multiple néanmoins, les *interfaces* permettent de bénéficier de tous les avantages de l'héritage multiple en évitant certains inconvénients.

D.3. Polymorphisme

Cette notion souligne le fait qu'un objet (une instance de classe) va pouvoir s'adapter au contexte dans lequel il est utilisé en proposant au développeur plusieurs méthodes permettant d'atteindre le même objectif. Le développeur sera alors à même de choisir la méthode qui lui convient selon la situation.
Prenons l'exemple d'un compte bancaire qu'il est possible de créditer avec une somme soit en francs, soit en euros. Il serait très utile que les instances de la classe *CompteBancaire* soient capables, de créditer en francs et en euros. Le concepteur de classe va donc proposer deux méthodes *crediter(…)* différentes. L'une prenant des francs et l'autre des euros.

Vous comprendrez mieux la notion de polymorphisme lorsque que vous verrez des exemples en JAVA. Pour l'instant, considérez que cette notion s'applique à tout objet ayant des fonctionnalités multiples et redondantes dépendantes du contexte dans lequel il se trouve.

En fait, le polymorphisme s'applique particulièrement aux *méthodes* qui peuvent être *surchargées*. (Voir plus loin dans le cours)

Chapitre 8 : Structure objet de JAVA

JAVA étant un langage purement orienté objet, il respecte les règles énoncées au chapitre précédent. Nous allons voir, dans ce chapitre, comment JAVA implémente toutes les fonctionnalité d'héritage, encapsulation et polymorphisme.

A. Les classes

Les quelques exemples vus dans la 1ère partie de ce cours ont montré la structure de petites classes. De manière générale, une classe est stockée dans un fichier portant le même nom.
Pour la classe *MaClasse* , un fichier nommé *MaClasse.java* va contenir le corps de la classe.

```
public class MaClasse {

// Corps de la classe
}
```

Le modificateur d'accès *public* va permettre à d'autres utilisateurs d'instancier cette classe pour créer des objets.

A.1. Créer des objets (Instancier)

La première opération que vous serez amenez à faire, est l'utilisation des classes existantes. Vous allez donc créer vos objets à partir de ces classes. La création d'objets à partir de classes s'appelle *l'instanciation*.
Ce processus va créer le nouvel objet grâce à l'opérateur *new* :

```
CompteBancaire monCompte = new CompteBancaire(400.0) ;
```

ou

```
CompteBancaire monCompte;                // déclaration
monCompte = new CompteBancaire(400.0) ; // création et appel au
                                              // constructeur
```

L'exemple ci-dessus va créer le nouvel objet nommé *monCompte* à partir de la classe *CompteBancaire*. Ce qui suit l'opérateur *new* , c'est l'appel au *constructeur* de l'objet qui initialiser le compte avec 400 frs.
Nous reverrons le principe des constructeurs dans la suite du chapitre.

Vous remarquerez au passage que les noms des classes ont une majuscule sur la 1ère lettre alors que les instances n'en ont pas. Ceci est une convention qu'il faut absolument respecter car elle permet de distinguer un objet d'une classe.

A.2. Les champs ou données de la classe

a. Etat de l'objet

Ils représentent l'état de l'objet. Une classe peut contenir des données qui peuvent être des objets (*relation d'agrégation voir plus loin*) ou des types primitifs tels que des entiers, des tableaux, des chaînes de caractères.
Voici la classe contact qui contient les données nécessaires pour stocker l'état d'un objet:

```
public class Contact {

String nom;
String prenom;
String adresse;

int age;

}
```

b. Encapsulation de données

Ces données ou ces champs sont contenus dans la classe. Tout objet instancié à partir de cette classe pourra se servir de ces données. Par contre, la manière dont les données seront protégées ou non des autres objets va dépendre du type de modificateur employé. Il existe 3 types de modificateurs d'accès :

- **public** : Le champ est accessible et modifiable par tout le monde
- **private** : Le champ est hors d'atteinte. Seul l'objet lui-même peut y accéder
- **protected** : Le champ est accessible seulement par les classes enfants et la classe elle-même.

Si la classe fait partie d'un package, les classes appartenant au même package auront également accès aux champs protected

Si vous voulions protéger les données de l'exemple précédent, les données de la classe deviennent inaccessibles :

```
public class Contact {

private String nom;
private String prenom;
private String adresse;

private int age;

}
```

Le programme principal suivant n'aura pas accès à ces données :

```
public class Principal {

            public static void main (String[] args) {

            // Création d'un nouvel objet à partir de la classe
            Contact monContact = new Contact();

            monContact.age = 24;    // ERREUR , DONNEES PRIVEE !!!
```

```
            }
}
```

Vous devez vous demander l'intérêt de protéger les données d'une classe. Si nous reprenons l'exemple de notre classe *CompteBancaire* , nous nous apercevons que donner un accès public au champ *solde* peut avoir des conséquences fâcheuses surtout si une personne mal intentionnée venait à modifier cette valeur.

En général, les développeurs JAVA rendent leurs données privées et créent des méthodes permettant d'y accéder. Ces méthodes fixent les règles de l'encapsulation et permettent de protéger les données. Nous appelleront ces méthodes des accesseurs de données.

A.3. Les méthodes

Chaque classe possède un ensemble de méthodes qui représentent ce que l'objet sera capable de faire (son comportement). Les méthodes sont en fait, des blocs de code, réalisant un certain nombre d'opérations sur les différentes ressources d'une classe.
Une méthode peut prendre des arguments mais aussi renvoyer des données à l'objet appelant.
Imaginons une classe *CompteBancaire* , disposant d'une méthode *crediter* prenant comme argument la somme à créditer et ne renvoyant rien.
Voici la signature de la méthode :

```
public void crediter ( double montant )
```

La signature d'une méthode correspond à sa forme : Son nom , ce qu'elle renvoie, ce qu'elle prend comme arguments ainsi qu'un ou plusieurs modificateurs. le mot clé void sert à préciser que la méthode ne renvoie rien. Sa présence est obligatoire dans ce cas. Si aucun modificateur n'est précisé, la méthode sera accessible via les instances de classes venant du même package

Dans la pratique, voici le code source partiel de la classe *CompteBancaire* :

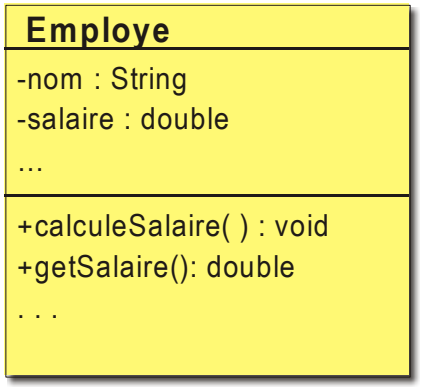
```
public class CompteBancaire {

private double solde;      // Champ de la classe

            public void crediter ( double montant ) { // méthode

            solde = solde + montant;

            }
}
```

Voici le programme principal chargé d'utiliser cette classe pour créer un objet *CompteBancaire* :

```
public class Ex9 {

            public static void main (String[] args) {

            // Création d'un nouvel objet à partir de la classe
            CompteBancaire monCompteBancaire = new CompteBancaire();

            monCompteBancaire.crediter(500.0);  // Crédit de 500 Frs

            }
}
```

A.4. Accesseurs de données

Pour illustrer notre propos sur la protection des données par l'encapsulation, voici un exemple d'une classe *Employe* qui contient une donnée *salaire* :



Le champ *salaire* est privé et la méthode *getSalaire()* permet de lire le salaire mais pas de l'écrire.
Pour modifier le salaire, il faut appeler la méthode *calculeSalaire( )* qui va recalculer le salaire en fonction de critères très précis et non modifiables.
Ici, le savoir-faire de l'objet et l'encapsulation des données permettent de ne montrer que ce qui peut l'être en évitant toute manipulation hasardeuse des salaires.

```
public class Employe {

private double salaire;
private double tauxHoraire = 350.0;

public void calculeSalaire(int heures) {

            salaire = heures * tauxHoraire;
}

public double getSalaire() {        // Renvoie un double

            return salaire;              // return utilisé pour retourner
            }                            // une valeur.
}
```

Voici le programme principal chargé d'afficher le salaire :

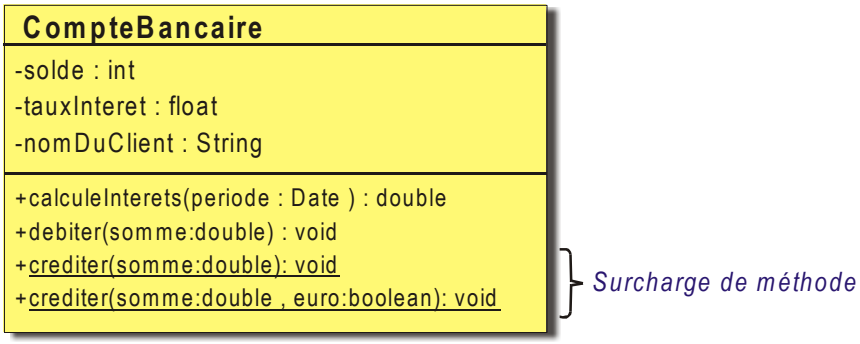
```
public class Ex10 {

public static void main (String[] args) {

            Employe dupont = new Employe();

            dupont.calculeSalaire(120);  // Appel à la méthode de l'objet
            System.out.println ( dupont.getSalaire());
                                        // Affichage du salaire
            }
}
```

Exemple 10

Dans cet exemple, l'utilisateur de la classe pourra modifier le salaire mais n'aura accès qu'au nombre d'heures sans connaître pour autant le taux horaire.

A.5. Surcharge de méthodes

Il arrive fréquemment qu'une classe permette d'accéder à ses données de manières différentes. Elle implémente alors des méthodes qui portent le même nom mais avec des signatures différentes. Cette technique est appelée la ***surcharge de méthodes***.
L'utilisateur de la classe peut alors, choisir la méthode qui correspond le mieux au contexte dans lequel il travaille.
Voici un exemple permettant d'illustrer l'intérêt de la surcharge de méthodes.



La classe *CompteBancaire* dispose de 2 méthodes *crediter* permettant de créditer le compte en euros ou en francs. La variable booléenne *euro* transmise signifie que la somme à créditer est en euro lorsqu'elle vaut *true* et en francs si elle vaut *false* :

```
public class CompteBancaire {

private double solde=0.0;

            public void crediter ( double montant ) {
            solde = solde + montant;
            }

            public void crediter ( double montant , boolean euro ) {

            if (euro==true)
                 crediter(montant*6.55957); // Appel à la méthode
                                            // ci-dessus
            else
                 crediter(montant);
            }
}
```

Remarquez que l'appel à la 2ème méthode provoque l'appel à la 1ère mais avec un montant converti en euro juste avant.

L'appel à une méthode à l'intérieur de la classe est possible. Pour éviter toute confusion, on fait appel à l'opérateur this. Cet opérateur est une référence à l'objet courant (L'instance courante de la classe).

Lorsque la classe sera instanciée, l'utilisateur pourra à loisir appeler l'une ou l'autre des méthodes. Dans ce cas, le compilateur recherche au sein de la classe, une méthode ayant la même signature.

Attention : Le simple fait d'appeler une méthode avec des types de paramètre différents provoque une erreur de compilation car le compilateur ne peut pas trouver de correspondance entre la méthode telle que l'écrit l'utilisateur et celle qui se trouve dans la classe. Voici un exemple :

L'appel suivant provoquera une erreur de compilation :

```
CompteBancaire monCompte = new CompteBancaire();

monCompte.crediter( 345.0 ,  "true" ); // ERREUR , un String est transmis
```

Dans ce cas, le compilateur va rechercher une méthode ayant la signature suivante:

```
public void crediter ( double , String )
```

Etant donné que la bonne méthode possède la signature suivante,

```
public void crediter ( double , boolean )
```

le compilateur ne peut pas effectuer de correspondance.

Le compilateur, peut, à l'occasion, effectuer un transtypage (une conversion) dans le cas où le type recherché est plus précis que le type fourni. Lors de l'appel à la méthode crediter(double montant ,boolean euro), l'appel suivant aurait engendré un transtypage du float ; crediter(132.34F , true)

A.6. Les constructeurs

a. Définition d'un constructeur

Un constructeur est une méthode un peu particulière qui porte le **même nom** que la classe , qui **ne renvoie rien** (pas même void) et qui ne pourra être exécuté **qu'une seule fois.** Comme toute méthode, un constructeur peut prendre des paramètres et également être surchargé.
Voici un exemple de constructeur pour la classe *CompteBancaire* :

```
public class CompteBancaire {

            private double solde ;

            public CompteBancaire(double monSolde) {
                 solde = monSolde ;
}
… // Suite de la classe
}
```

b. Objet implicite this

L'objet implicite *this* fait référence à l'objet lui-même. Comme une classe est destinée à engendrer des objets, il est bien évident qu'il n'est pas possible de connaître à l'avance tous les noms des futures références à cet objet. C'est pourquoi , on utilise *this* à l'intérieur d'une classe pour désigner l'objet lui-même lorsqu'il sera contruit.
L'exemple suivant démontre l'intérêt d'utiliser *this* pour empêcher toute ambiguïté sur deux variables ayant le même nom :

```
 public class CompteBancaire {

            private double solde ;

            public CompteBancaire(double solde) {
                 this.solde = solde ;
}
… // Suite de la classe
}
```

Ici, il faut distinguer la variable *solde* de la classe et la variable *solde* passée en paramètre du constructeur.

c. Mécanisme de construction

Les constructeurs sont des méthodes particulières. D'abord, elles portent **le même nom que la classe** (Avec une majuscule, comme la classe) . Ensuite, ces méthodes sont chargées d'**initialiser** l'objet (Tous ses champs) lors de l'instanciation.

> *Le terme constructeur n'est pas très approprié puisqu'ils ne servent pas à construire l'objet mais à initialiser les données qu'il contient.*

Vous avez déjà rencontré sans le savoir des constructeurs dans de nombreux exemples de ce cours. En effet, l'appel au constructeur se fait conjointement avec l'opérateur ***new***.
Voici un exemple d'utilisation :

```
CompteBancaire monCompte = new CompteBancaire(1230.5);
```

d. Constructeur par défaut

Comme tout objet se doit d'être instancié avant d'être utilisé, la présence d'un constructeur par défaut est indispensable. Java vous laisse deux alternatives en ce qui concerne les constructeurs. La première consiste à n'implémenter aucun constructeur dans la classe. Dans ce cas, vous confiez à Java le soin de définir lui-même un constructeur par défaut qui ne fera rien. La seconde alternative vous oblige à créer au moins un constructeur. Dans ce cas, Java vous considère responsable de la construction de vos objets et n'essaiera pas d'appeler le constructeur par défaut.
Voici un exemple d'une classe sans constructeur :

```
public class Classe {

            private int champ;

            public void methode() {
                  // Contenu de la méthode
            }
}
```

et le programme principal chargé d'instancier un objet :

```
public class Main {

            public static void main ( String[] args){

                  Classe objet = new Classe();
                  objet.methode() ;
            }
}
```

Ici, même si aucun constructeur n'est présent, Java s'occupe de tout (Sauf bien sûr d'initialiser le champ *champ*.

Le deuxième exemple, provoquera une erreur de compilation :

```
public class Classe2 {

            private int champ;

            public Classe2(int unChamp) {
                  this.champ = unChamp;
            }

}
```

voici le programme principal chargé d'instancier un objet :

```
public class Main2 {

            public static void main ( String[] args){

                  Classe2 objet = new Classe2();
            }
}
```

Le compilateur renvoie une erreur car le constructeur par défaut n'existe pas. Java vous confie le soin de construire vous-même vos objets et par conséquent ne fabrique pas de constructeur par défaut :

```
Main2.java:6: cannot resolve symbol
symbol  : constructor Classe2 ()
```

```
location: class Classe2
                Classe2 objet = new Classe2();
                                ^
1 error

Processus terminé avec code quitter 1
```

Si nous reprenons notre exemple du compte bancaire, il peut être très utile de prévoir un constructeur avec le solde de départ :

```
public class CompteBancaire {

private double solde;

            public CompteBancaire ( double solde ) { // Constructeur

            this.solde = solde;
            }

            public void crediter ( double montant ) {
            solde = solde + montant;
            }

            public void crediter ( double montant , boolean euro ) {

            if (euro==true)
                 this.crediter(montant*6.55957); // Appel à la méthode
                                                 // ci-dessus
            else
                 this.crediter(montant);
            }
}
```

Voici plusieurs remarques :

- Les constructeurs ne renvoient rien , pas même *void*. **Ne rajoutez jamais *void* devant le constructeur !!! Il ne serait plus considéré comme un constructeur mais comme une méthode classique**
- l'opérateur *this* permet de ne pas confondre la variable transmise lors de l'appel au constructeur et le nom du champ appartenant à la classe.
- Le constructeur est , en général, utilisé pour initialiser toutes les données de la classe.
-

Dans certains cas, il faut employer this de manière explicite pour éviter toute confusion avec d'autres variables qui porteraient le même nom. En tout cas , si this n'est pas présent et qu'un des paramètres porte le même nom qu'un champ de classe, le compilateur considérera qu'il s'agit du paramètre et non du champ de classe

- Voici le programme principal :
-

```
public class Principal {

public static void main (String[] args) {

            CompteBancaire monCompte = new CompteBancaire(450.0);
            }
}
```

-

Notez l'appel au constructeur de la classe *CompteBancaire*.Le constructeur étant une méthode comme les autres, il est possible de le surcharger pour proposer à l'utilisateur plusieurs mécanismes d'initialisation d'objets.

Au lieu de modifier directement le champ solde dans le constructeur nous aurions pu utiliser la méthode crediter(double montant) , ce qui aurait donné le code ci-dessous. Vous remarquerez que , comme la méthode crediter appartient à la classe elle-même, vous employez directement le nom de la méthode nom d'objet le précédant :

```
public CompteBancaire ( double solde ) { // Constructeur

this.solde = 0;
crediter( solde ); // appel à la méthode crediter
}
```

e. Surcharge des constructeurs

La surcharge des constructeurs est très utilisée en JAVA. Elle consiste à proposer plusieurs formes d'initialisation d'objets, ce qui devient très pratique pour le développeur.
En effet, n'oubliez pas que la majeure partie du travail du développeur est la réutilisation de classes existantes pour créer les siennes. Donc, plus cette réutilisation est simple et moins son travail est fastidieux.
Voyons un exemple simple pour lequel la surcharge des constructeurs est très utile. Il s'agit de la classe *JButton* permettant de créer des boutons dans une application graphique. Il existe pour cette classe, 5 constructeurs différents :

Sommaire des constructeurs de *JButton*
JButton() Crée un bouton sans texte ni icône
JButton(D:\jdk1.3docsapijavaxswingAction.html Action a) Crée un bouton dont les propriétés sont tirées de l'objet Action passé en paramètre
JButton(Icon icon) Crée un bouton avec l'icône passé en paramètre
JButton(String text) Crée un bouton avec le texte passé en paramètre
JButton(String text, Icon icon) Crée un bouton avec le texte et l'icône passés en paramètres

L'instruction permet de créer ce bouton avec du texte :

```
JButton monBouton = new JButton ( "Quitter" );
```

ou sans texte :

```
JButton monBouton = new JButton ();
```

ou avec un icône :

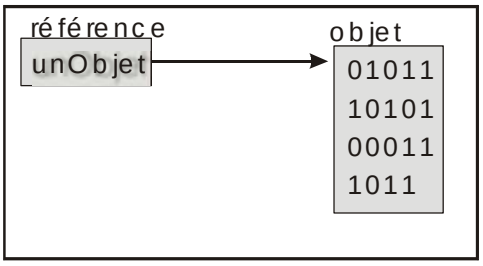
```
JButton monBouton = new JButton ("Quitter" , monIcone);
```

B. Les références aux objets

En Java, de nombreux objets peuvent être fabriqués à partir d'une classe. Lorsque l'on crée un objet, il est indispensable de pouvoir appeler ses méthodes. Pour cela , on utilise les références :

```
Classe unObjet = new Classe() ; // unObjet est une référence
```

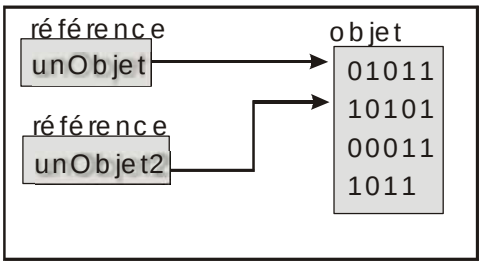
A l'exécution, la machine virtuelle va donc stocker cet objet quelque part dans la mémoire puis créera une référence pointant sur cet objet :



L'opération suivante aura pour effet de créer une nouvelle référence mais pas un nouvel objet :

```
Classe unObjet2 = unObjet ; //
```

Le schéma précédent devient alors :



Evidemment, si on essaie de modifier l'état de l'objet, il suffit d'utiliser l'une ou l'autre des références.

B.1. Le ramasse-miettes (garbage collector)

En JAVA, lorsqu'un objet ne sert plus à rien, il n'est pas nécessaire de l'effacer de la mémoire. En effet, un outil très efficace nommé le *garbage collector* s'occupe de vérifier l'utilité de garder en vie les objets et les supprime le cas échéant. La conséquence directe est que le développeur ne se soucie plus de cet aspect tant redouté par les programmeurs C++ qu'est la destruction des objets.
Dans le code suivant :

```
{
String s = "chaine";
}
System.out.println(s);  // Erreur à la compilation car s n'existe plus
```

La référence s a une portée limitée aux accolades. Cette référence n'existe pas en dehors de ces limites , par contre l'objet String lui continue d'exister au délà. Heureusement, le ramasse-miettes va s'occuper de supprimer l'objet.

B.2. La méthode finalize()

La classe *Object* dispose d'une méthode *finalize()*. Cette méthode, n'est pas, comme on pourrait le penser, un destructeur d'objet comme le sont les destructeurs en C++. En fait, cette méthode est appelée par le ramasse-miettes au moment où ce dernier cherche à libérer de la mémoire.

> Le ramasse-miettes ne cherchera à libérer de la mémoire que si celle-ci lui fait défaut. Ce qui signifie que les objets ne seront pas forcément détruits. Donc, la méthode finalize() ne sera pas forcément appelée.

Etant donné que le ramasse-miettes s'occupe de tout, l'intérêt de la méthode *finalize()* n'est pas totalement justifié. Son intérêt réside dans le fait que l'opérateur *new* n'est le seul moyen de créer des objets en Java. En effet, Java peut faire appel à du code natif (en c ou c++) qui va allouer de la mémoire pour sa propre consommation. Cette mémoire allouée ne sera pas effacée par le garbage collector. Voilà pourquoi les concepteurs de Java ont crée la méthode *finalize()*.
Cette méthode sera utilisée uniquement si vos objets ont écrit dans la mémoire par un autre moyen que l'opérateur *new*.

C. L'héritage

Vous voilà rendu à l'un des aspects le plus intéressant et le plus riche des langages orientés objet. L'héritage vous permettra d'enrichir des classes existantes , dont vous êtes le créateur ou non , pour construire des classes plus spécifiques qui hériteront du savoir-faire de leur parent.
Cette technique est très utilisée en JAVA et permet d'aller beaucoup plus loin dans la résolution de problèmes complexes.

C.1. Classes enfants

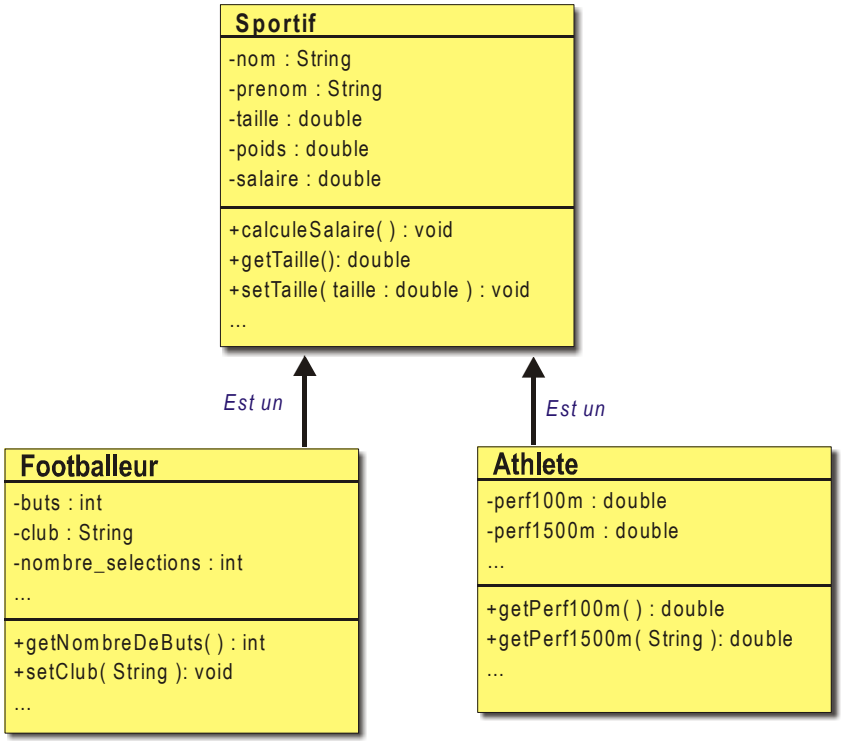
Les classes enfants bénéficient de toutes les caractéristiques de leur parent. C'est à dire qu'elles peuvent utiliser leurs méthodes, leurs champs (Seulement les champs protégés et public mais pas les champs privés). Les enfants peuvent également profiter des constructeurs de leur parent.
Pour créer une relation d'héritage, vous utiliserez le mot clé ***extends***. En effet, l'en-tête de la classe va se présenter de la manière suivante :

```
class Footballeur extends Sportif { …
```

La relation d'héritage entre *Footballeur* et *Sportif* sera clairement établie.

> *Considérez maintenant que tout footballeur EST un sportif , ce qui n'a rien d'illogique après tout ! Le fait de créer des classes enfants vient du besoin d'avoir des objets plus spécifiques. En effet, les caractéristiques d'un sportif ne préjugent pas de ses capacités à jouer au football. Il faut donc créer une classe Footballeur qui permettra de stocker des données sur ses capacités par rapport à son sport.*

Voici un exemple d'héritage qui vous propose 3 classes avec des relations d'héritage :



C.2. Classe parent et hiérarchie

La classe parent rend certaines méthodes et champs disponible pour ses enfants. L'enfant peut alors utiliser ou redéfinir les méthodes si celle ci ne leur conviennent pas.
Pour justifier l'intérêt de l'héritage, l'enfant peut dire à son parent :

"Ta méthode est bien mais je veux la redéfinir pour mes besoins spécifiques"
ou encore

"J'utilise tes méthodes mais j'y ajoute des fonctionnalités"

Voici le code JAVA pour les classes *Footballeur* et *Sportif* :

```
public class Sportif {

            private String nom,prenom;
            private double taille, poids;
            private double salaire;

                 // Constructeur
                 public Sportif( String nom , String prenom , double
                                                  taille ,double poids) {

                      this.nom = nom;
                      this.prenom = prenom;
                      this.taille = taille;
                      this.poids = poids;
                 }

                 // Méthode d'accès à la taille
```

```
                public double getTaille() {

                     return taille;
                     }

                // Méthode d'accès au poids
                public double getPoids() {

                     return poids;
                }


}
```

Sportif.java

- La classe *Sportif* dispose de plusieurs champs qui seront initialisés à l'appel du constructeur.
- L'appel à l'opérateur *this* évite les confusions avec les arguments transmis avec le constructeur.

```
public class Footballeur extends Sportif {

            private int nombreDeButs;
            private String club;

            // Constructeur
            public Footballeur( String nom, String prenom, double taille ,
                                double poids, int buts , String club ) {

                  super(nom,prenom,taille,poids);      // Utilisation du
                                                       // constructeur du parent
                  nombreDeButs = buts;
                  this.club = club;
                  }

            // Méthode d'accès au club
            public String getClub() {
                  return club;
                  }

            // Méthode d'accès au nombre de buts marqués
            public int getNombreDeButs() {

                  return nombreDeButs;
                  }
}
```

Footballeur.java

Le constructeur appelle le constructeur de la super-classe grâce au mot clé *super*. Ce qui permet de bénéficier du savoir faire de son parent. Ce processus est très intéressant et permet au développeur de se concentrer sur sa nouvelle classe tout en bénéficiant du travail accompli sur la classe parent.

Le rôle du constructeur de la super-classe est d'initialiser les données de la super-classe. Les classes enfants sont tenues d'appeler le constructeur de la super-classe sous peine de travailler avec des données de super-classe non initialisées. En JAVA, une donnée d'instance que l'on utilise sans une initialisation préalable provoque une erreur de type NullPointerException.

Voici le programme principal qui utilise les caractéristiques de l'héritage :

```
public class Ex11 {

public static void main (String[] args) {

Footballeur zizou = new Footballeur (
                     "Zidane","Zinédine",1.82,80.0,124,"Real Madrid");


            System.out.println( zizou.getTaille()); // Utilisation de la
                                                    // méthode du parent
            }
}
```

Ex11.java

C.3. Transtypage (Casting)

Le transtypage est l'action qui consiste à convertir les objets en objets plus ou moins spécifiques. Le transtypage implique une relation d'héritage entre les classes impliquées. Il est possible, sous certaines

conditions de convertir un *Sportif* en *Footballeur*. Par contre , il est impossible de convertir une *String* en *Sportif* puisque aucune relation d'héritage existe entre les deux classes.
Pour mieux comprendre l'intérêt du transtypage, considérons l'exemple suivant :

```
public class Main2 {

public static void main (String[] args) {


Sportif [] liste = new Sportif[2];

liste[0] = new Footballeur (
                    "Zidane","Zinédine",1.82,80.0,124,"Real Madrid");

liste[1]= new Sportif("DUPONT","Jean",1.70,75.0);


            /* ERREUR de compil : Sportif de contient pas de méthode
             getNombreDeButs() */
            int buts = liste[0].getNombreDeButs();
            }
}
```

Dans cet exemple, nous créons un tableau de deux *Sportif*. Chaque élément du tableau est une instance de *Sportif* ou une instance d'une des classes enfants de *Sportif* (*Footballeur* dans l'exemple). La ligne 16 provoque une erreur de compilation car même si *liste[0]* a été crée comme un *Footballeur* , il est considéré comme un *Sportif*.
Pour pouvoir à nouveau utiliser *liste[0]* comme un *Footballeur* , il faut le **transtyper** :

```
Footballeur f = (Footballeur) liste[0];
int buts = f.getNombreDeButs();
```

Le transtypage qui consiste à convertir un objet en un objet plus spécifique s'appelle le transtypage descendant , ou spécialisation (downcasting). L'action qui consiste à considérer un objet comme une instance de la classe parent s'appelle le transtypage ascendant ou généralisation ou surtypage (upcasting) . Cet opération est implicite dans le cas du transtypage ascendant.

Voici quelques exemples possibles et impossibles d'utilisation du transtypage :

```
public class Main {

public static void main (String[] args) {


Sportif [] liste = new Sportif[2];

liste[0] = new Footballeur (
                    "Zidane","Zinédine",1.82,80.0,124,"Real Madrid");

liste[1]= new Sportif("DUPONT","Jean",1.70,75.0);


            /* ERREUR de compil : Sportif de contient pas de méthode
             getNombreDeButs() */
            int buts = liste[0].getNombreDeButs();

            /* OK : un footballeur EST UN sportif */
            Footballeur f = (Footballeur)liste[0];
            int buts = f.getNombreDeButs();

            /* ERREUR EXECUTION : sportif[1] n'est pas un footballeur */
            Footballeur f = (Footballeur)liste[1];
            int buts = f.getNombreDeButs();
            }
}
```

C.4. Empêcher l'héritage

a. Modificateur final pour une classe

Afin de protéger certaines classes et d'empêcher qu'elles puissent avoir des enfants, on utilise le mot clé ***final***. Ce modificateur est placé dans la déclaration de la classe :

```
public final class MaClasse { … }
```

L'avantage d'empêcher une classe d'avoir des enfants est certes de protéger ses propres classes mais cette fonctionnalité permet également d'améliorer les performances d'un programme. Sans rentrer dans les détails, il faut savoir qu'une classe *final* donnera moins de travail au microprocesseur et contribuera à améliorer les performances d'un programme. D'ailleurs , la classe *String* de JAVA est une classe qui n'est pas possible de dériver.

b. Modificateur final pour une méthode

Lorsque vous appliquez ce modificateur à une méthode mais pas forcément à sa classe, les classes enfant ne pourront en aucun cas surcharger cette méthode.
Cette fonctionnalité vous permet de verrouiller certaines méthodes dites sensibles en empêchant tout utilisateur de votre classe de redéfinir des méthodes.
A titre d'exemple, la méthode *crediter* de la classe *CompteBancaire* doit être déclarée *final* pour ne pas que d'autres développeurs puissent créditer un compte d'une manière différente.

```
final public crediter( float montant) { … }
```

c. Modificateur final pour un champ

Pour un champ, la signification n'est pas la même. Il désigne un champ qui ne peut être modifié. Vous appliquerez le modificateur *final* aux constantes. Vous veillerez également à mettre en majuscule le nom de ces constantes afin de respecter la convention :

```
private final int VALEUR_EURO = 5.55957;
```

Il existe, dans les classes JAVA, de nombreux exemples de champs constants. En général, ces champs constants sont également dits "statiques" grâce à l'attribut static dont nous reparlerons plus tard.

C.5. Mécanisme de l'héritage

a. Chaîne d'héritage

Il importe de bien comprendre ce qui se passe lorsqu'un appel de méthode est appliqué à des objets de types différents dans la hiérarchie des classes :

- La sous classe vérifie qu'elle possède bien une méthode de ce nom avec la même signature. Si c'est le cas, elle l'utilise
- Sinon, la classe parent est interrogée et détermine si elle possède une telle méthode. Si c'est le cas elle l'utilise.

Ce processus peut remonter ainsi toute la hiérarchie des classes jusqu'à ce qu'un méthode correspondante soit retrouvée. Si aucune ne correspond, le compilateur génère une erreur.

b. Masquage de méthodes

La conséquence directe de ce mécanisme est que si une méthode a le même nom et la même signature dans une classe enfant et dans la classe parent, celle-ci masque automatiquement les autres.

c. Exemple de la méthode toString()

Un exemple qui illustre parfaitement ce mécanisme est la méthode *toString()*. Cette méthode est chargée pour tous les objets, de donner une représentation sous forme de chaîne de caractère de l'objet.
Si vous essayez d'appeler la méthode *toString()* à un objet, vous obtiendrez une chaîne de caractère représentant l'objet :

```
public class Ex12 {

public static void main (String[] args) {

Object monObjet = new Object();

System.out.println(monObjet.toString() );
}
}
```

Vous obtiendrez une chaîne à peu près comme celle ci :

```
java.lang.Object@73d6a5
```

Cette chaîne ne signifie pas grand chose pour nous. Voilà pourquoi, il peut être très utile de masquer cette méthode dans vos propres classes. En effet, n'oublions pas que toute classe est une classe enfant de la classe *Object*.
Par exemple, pour notre classe *Footballeur* vue au chapitre précédent, il peut être utile de masquer la méthode *toString()* par votre propre méthode qui affichera le nom, le prénom, la taille et le poids du joueur :

```
public class Sportif {

            private String nom,prenom;
            private double taille, poids;
            private double salaire;

                  // Constructeur
                  public Sportif( String nom , String prenom , double
                                                taille , double poids) {

                        this.nom = nom;
                        this.prenom = prenom;
                        this.taille = taille;
                        this.poids = poids;
                  }

                  // Méthode d'accès à la taille
                  public double getTaille() {

                        return taille;
                        }

                  // Méthode d'accès au poids
                  public double getPoids() {

                        return poids;
                  }

                  public void setTaille(double taille) {

                              this.taille= taille;
                        }

            // Méthode donnant une représentation du sportif
            public String toString() {

            String chaine;

                  chaine = "Joueur : " + prenom + " " + nom + "\n";
                  chaine += "Taille : " + taille + " m, Poids : " +
                                                      poids + " kg";
                  return chaine;
            }
}
```

Sportif.java

Voici le programme principal :

```
public class Ex12 {
```

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

Sportif monSportif = new Sportif ("Dupont","Jean",1.80,78.0);

            System.out.println( monSportif.toString());
            }
}
```

Ex12.java

L'appel à la méthode *toString()* renverra une chaîne de caractère représentant les caractéristiques du joueur, ce qui est beaucoup plus intéressant qu'une chaîne comme celle–ci :

```
java.lang.Object@73d6a5
```

ATTENTION : Pour masquer une méthode, il faut qu'elle aie strictement la même signature que la méthode de la classe parent. Donc, si la classe parent renvoie une String , la méthode masquante renverra également une String

d. Exemple de la méthode equals()

Cette méthode permet de savoir si les deux références *objet1* et *objet2* pointent sur le même objet en mémoire:

```
if ( monObjet1.equals(monObjet2))
            System.out.println("Ces objets sont identiques");
```

Cela ne présente pas vraiment d'intérêt. Néanmoins, il peut être fort utile de surcharger la méthode *equals()* dans les sous classes de manière à effectuer un traitement d'égalité qui aurait une réelle signification pour ces objets.
Par exemple, nos objets de type *Footballeur* seraient alors comparés en fonction du nombre de buts qu'ils ont marqués et non de leur emplacement en mémoire.

e. La classe Object

Comme nous l'avons vu précédemment, cette classe est parente de toutes les autres classes. Elle représente l'ancêtre de toute nouvelle classe. Elle est considérée comme la classe parent par défaut de toute nouvelle classe. Ce qui explique qu'il n'est pas nécessaire d'écrire :

```
public class MaClasse extends Object { … }
```

Une variable de type *Object* peut référencer n'importe quel type d'objet :

```
Object monObjet = new Footballeur ( "Zidane" , "Zinédine" , 1.81 , 78 , 145 ,
"Real de Madrid");
```

Naturellement, pour pouvoir utiliser ce footballeur, il faut effectuer un transtypage pour accéder à ses champs propres :

```
FootBalleur zizou = (Footballeur) monObjet ;
```

L'intérêt de considérer le footballeur comme un objet de type *Object* est de pouvoir effectuer des traitements génériques. Un traitement générique est un traitement qui peut être effectué pour n'importe quel type d'objet. Dans ce cas, il n'est pas nécessaire de réécrire du nouveau code pour chaque nouvel objet.
Prenons l'exemple d'un tableau contenant des objets. Si l'on souhaite retrouver l'endroit du tableau où se situe un objet en particulier, nous sommes en présence d'une situation qui requiert un traitement générique.
La méthode *trouver* prend comme argument le tableau d'objets où chercher ainsi que l'objet à trouver :

```
int trouver( Object [] a , Object obj ) {

{
            for (int i=0 ; i < a.length; i++)
                 if (obj.equals(a[i]))
                      return i;
            return -1;
}
```

La méthode *equals(..)*, qu'elle soit surchargée ou non, sera exécutée. Si l'objet courant a une méthode surchargée, elle sera choisie, sinon, c'est celle *d'Object* qui sera exécutée. Dans un cas comme dans l'autre, l'appel à cette méthode vérifiera la correspondance entre les 2 objets.
Evidemment, afin d'effectuer un test d'égalité ayant un sens, il vaudrait mieux que la méthode *equals* soit surchargée dans toutes les classes susceptibles d'être comparées.
Vous voyez bien que grâce à la surcharge, l'objet sait ce qu'il a à faire lorsqu'on lui dit de se comparer à un autre objet.

> La méthode equals ne doit pas pouvoir être appelée si les 2 objets à comparer ne font pas partie de la même famille d'héritage. En effet, comparer des Sportif et des String est incohérent. Lorsque vous surchargez la méthode equals, il ne faut pas oublier qu'une exception de type ClassCastException sera générée si l'objet fourni en paramètre n'est pas du même type que l'objet courant.

La méthode *equals*() de la classe *Sportif* , surchargée est la suivante :

```
public class Sportif {

// … Contenu de la classe

public boolean equals ( Object obj ) {

if ( obj instanceof Sportif ) {
            Sportif sp = (Sportif) obj;      // Transtypage obligatoire
            if (sp.nom.equals(nom) && sp.prenom.equals(prenom)
                                    && sp.taille==taille
                                           && sp.poids == poids  )
                 return true;
            else
                 return false;
            }
else
            return false;
}
```

La comparaison va être effectuée sur le nom, le prénom, la taille et le poids du sportif.

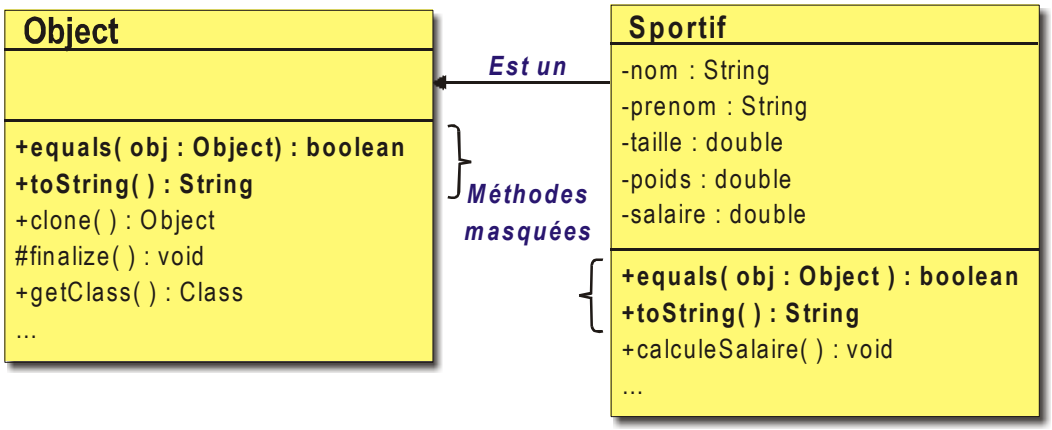


Diagramme Objet des classes Object et Sportif

Lors de l'appel à la méthode ***trouver()*** , le compilateur va choisir la méthode la plus appropriée, c'est à dire celle de la classe ***Sportif***. Cette méthode ***trouver()*** pourra alors être réutilisée pour n'importe quelle type d'objet à condition que le concepteur de l'objet aie pris la peine de surcharger la méthode ***equals()***.

f. Polymorphisme et héritage

Ce terme désigne la capacité d'un objet à décider quelle méthode il doit appliquer sur lui-même et selon sa position dans la hiérarchie. Le polymorphisme est fondé sur le simple fait qu'un objet peut réagir différemment à un même message ou appel.
Par exemple, l'appel à la méthode *toString* n'aura pas le même effet selon la position de l'objet dans la hiérarchie.

g. Constructeur de la superclasse

Il peut être utile de faire appel au constructeur de la superclasse. Pour le faire, on utilise le mot clé *super* en prenant bien soin de respecter la signature. Comme les constructeurs sont des méthodes spéciales, on

utilise ce mot clé pour bien spécifier au compilateur qu'il s'agit bien du constructeur de la superclasse et non le constructeur de la classe enfant

```
super( param1, param2 );
```

C.6. Classes abstraites

a. Nécessité et intérêt

Généralement, plus on remonte dans la hiérarchie des classes et plus ces classes paraissent abstraites. Néanmoins, elles peuvent se charger de certaines opérations communes à toutes les classes enfants. Evidemment, plus on descend dans la hiérarchie et plus les classes deviennent spécifiques.
En JAVA, il existe des classes dites *abstraites*, qui ne font rien, mais dont le rôle est destiné à avoir des classes enfants qui elles, auront des tâches à accomplir. Ces classes contiennent en général des méthodes abstraites qui ne font rien mais qui constituent une promesse faite que toutes les classes enfants implémenteront obligatoirement ces méthodes. Par contre une classe abstraite peut contenir des données ou des champs concrets.
Les classes abstraites sont là pour orienter la conception des classes enfants ainsi que pour faciliter la compréhension générale.

Les classes abstraites ne peuvent pas être instanciées et donc, aucun objet ne peut en être issu. Néanmoins, il est possible de créer une variable objet à partir d'une classe abstraite seulement si cet objet fait référence à une classe concrète :

```
Vehicule monAuto = new Voiture();
```

b. Exemple

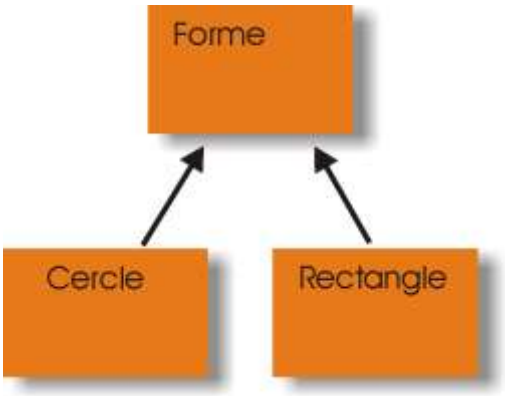


Schéma 7 – Exemple d'héritage

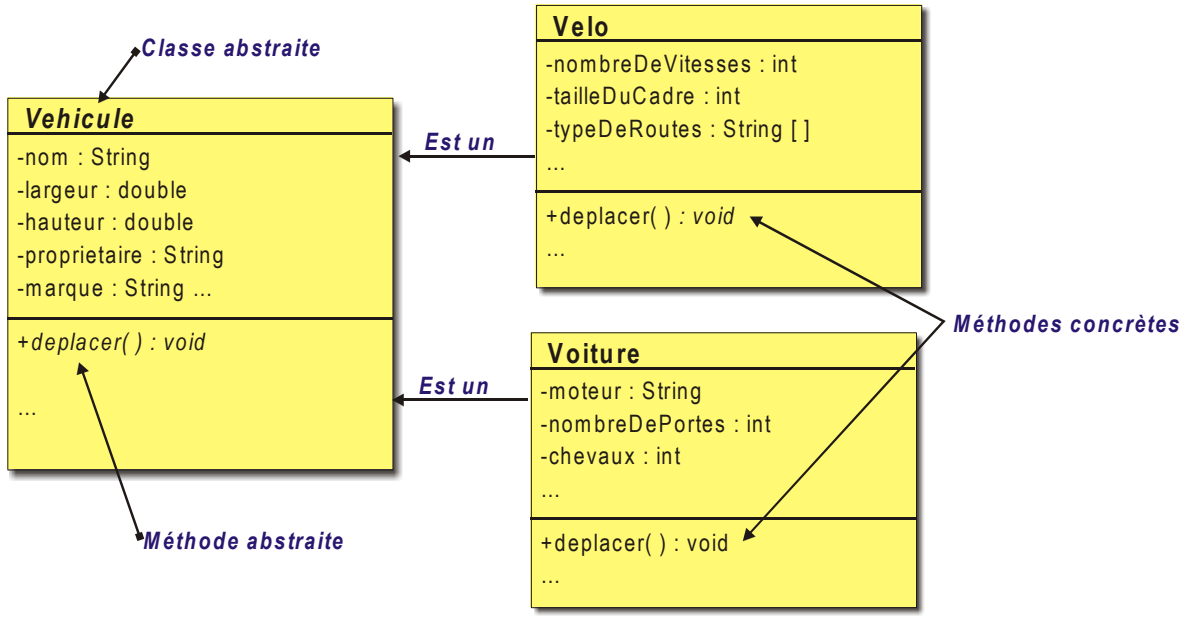
Par exemple, si on demande à un objet issu de la classe *Forme* quel est sa surface ? Que répondra t-il ? Cela dépend de la forme. Pour un cercle, c'est $\pi \cdot rayon^2$ alors que pour un rectangle, c'est $largeur \times hauteur$. Donc, la classe *Forme* ne servira pas directement pour créer des objets mais chaque *enfant* de *Forme* aura sa manière propre de calculer sa surface.

c. Méthodes abstraites

Les méthodes abstraites sont des promesses faîtes que toutes les classes enfants implémenteront ces méthodes. Une méthode abstraite ne contient pas d'implémentation. Cette implémentation est à la charge de la classe enfant.

d. Exemple 1

Voici un exemple avec la classe abstraite *Vehicule* , dont le nom apparaît en italique , et des classes *Velo* et *Voiture* . Pour un véhicule, la méthode *deplacer()* ne signifie pas grand chose. Le mode de déplacement va dépendre du type de véhicule utilisé. D'où l'utilité de rendre la méthode *deplacer()* abstraite :



Exemple de classe abstraite

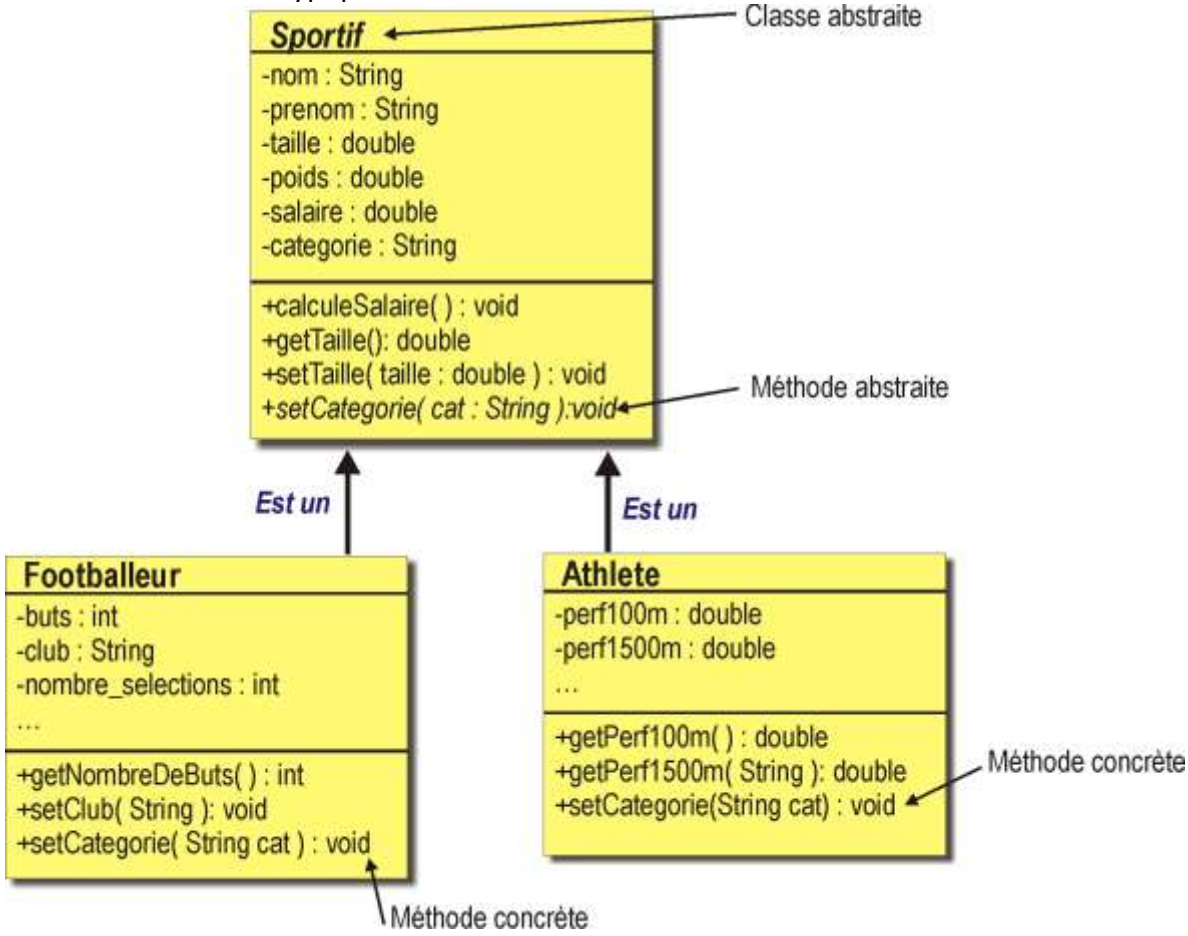
> *Une classe abstraite peut contenir des méthodes concrètes. Une classe qui contient une ou plusieurs méthodes abstraites doit obligatoirement être déclarée abstraite. De même une classe enfant qui hérite d'une classe abstraite doit implémenter toutes les méthodes abstraites de son parent (A moins qu'elle ne soit elle-même déclarée abstraite).*

Pour rendre une méthode ou une classe abstraite, on utilise le mot clé *abstract* :

```
public abstract class Vehicule {

public abstract void deplacer( );

// …
}
```

e. Exemple 2

Reprenons l'exemple des footballeurs et des sportifs. Imaginons que l'on souhaite prendre en considération le fait qu'un sportif, quelque soit son sport , fait partie d'une catégorie. Même si tout sportif a une catégorie, le nom de cette catégorie dépend du sport dans lequel il évolue. Un footballeur peut évoluer dans la catégorie *CFA* alors qu'un basketteur évoluera dans la catégorie *Promotion d'excellence*.
Nous sommes devant un cas typique de l'utilisation des classes abstraites. Le modèle deviendrait :



La généricité de la classe Sportif permet de considérer toutes les instances de Footballeur et Athlete comme des formes. Cela devient très utile lorsque, par exemple, on souhaite afficher les noms des sportifs sans se soucier de savoir s'il s'agit d'un footballeur ou d'un athlète.

f. Conclusion

L'emploi des méthodes et des classes abstraites survient surtout dans le haut de la hiérarchie de vos propres classes. Même si au début, leur emploi ne vous semble pas porter un très grand intérêt , elles facilitent la lecture et la compréhension de vos objets.

Voici quelques exemples de classes abstraites existantes dans les API de JAVA 2 :

- **Calendar** : Classe utilisée pour gérer les dates en JAVA. Elle contient un certain nombre de champs utilisés avec les dates. Elle est la superclasse de *GregorianCalendar* qui représente le calendrier de type Grégorien.
- **Collator** : Classe utilisée pour la recherche et la comparaison de chaînes dans des documents. Elle fourni les méthodes de base ainsi que quelques champs statiques pour la manipulation de texte. Sous classe connue : *RuleBasedCollator*
- **FileView** : Classe utilisée pour gérer la charte graphique (Le Look & Feel) de la classe *JFileChooser* . Sous classe connue : *BasicFileChooserUI.BasicFileView*
-

De manière générale, tout ce qui représente un concept abstrait se modélise par une classe abstraite.

D. Champs et méthodes statiques

D.1. Le principe

Lorsque nous avons parlé des constantes en JAVA, nous avons abordé le modificateur *final*. En JAVA, il existe un autre type de modificateur qui permet aux champs et aux méthodes qui en sont affectés, d'être utilisés directement sans le besoin d'avoir une instance.
Cette fonctionnalité peut être très utile pour réaliser des traitements ou récupérer des données de la classe qui ne nécessitent pas la création d'un objet. C'est le cas, par exemple de la mise en place de boîte de dialogues dans les applications graphiques en JAVA.

Une donnée ou une méthode statique appartient à la classe et non plus à l'objet comme c'est le cas pour une donnée ou une méthode non-statique.

D.2. Exemple 1

La classe *Double* possède plusieurs méthodes statiques. Par exemple, la méthode *parseDouble* permet de convertir une chaîne de caractères en une variable de type *double*. Cette méthode s'utilise comme ceci :

```
String chaine = "134.456" ;
double d = Double.parseDouble(chaine) ;   // chaine est convertie
```

D.3. Exemple 2

La classe *JOptionPane* , utilisée pour afficher des boîtes de dialogue, possède des champs et des méthodes statiques. Le rôle des champs statiques est généralement de contenir des valeurs utilisées pour paramétrer les boîtes de dialogue. Jetons un coup d'œil à la documentation de l'API JAVA de *JOptionPane* :

static int	**YES_NO_CANCEL_OPTION** Utilisé pour la méthode showConfirmDialog.
static int	**YES_NO_OPTION** Utilisé pour la méthode showConfirmDialog.

Champs statiques de la classe JOptionPane

Ces 2 entiers stockés vont permettre à la méthode *showConfirmDialog* de savoir s'il faut afficher les boutons *YES, NO* ou *YES, NO et CANCEL* :

Exemple de fenêtre obtenue avec JOptionPane

Dans la pratique, le programme suivant va permettre l'affichage de cette boîte :

```
import javax.swing.*;

public class Ex13  {

// Programme principal
public static void main (String[] args) {

JFrame myFrame = new JFrame(); // Fenêtre de taille nulle

JOptionPane.showConfirmDialog(myFrame , "Message de confirmation",
                                    "Message" , JOptionPane.YES_NO_OPTION);

            }
}
```

Ex13.java

Ne prêtez pas attention à l'utilisation de la classe JFrame. Nous la reverrons dans le cours sur les interfaces graphiques de JAVA et la notion de conteneur. En fait les boîtes de dialogue générées à l'aide de JOptionPane nécessitent un conteneur pour être affichées.

La méthode statique *showConfirmDialog* ne nécessite pas l'instanciation d'un objet de type *JOptionPane*. L'appel à cette méthode se fait directement.
D'autre part, le dernier paramètre envoyé à l'appel de la méthode est un entier , plus précisément un champ statique ,récupéré directement dans la classe *JOptionPane*.
Vous pouvez changer ce paramètre en utilisant :

```
JOptionPane.showConfirmDialog(myFrame , "Message de confirmation",
```

```
                                    "Message" , JOptionPane.YES_NO_CANCEL_OPTION);
```

Vous obtiendrez alors la boîte de dialogue suivante :

*Exemple de fenêtre obtenue avec JOptionPane et l'option **YES_NO_CANCEL_OPTION***

Voici 2 des signatures de la méthode *showConfirmDialog* :

`static int`	**`showConfirmDialog`**`(Component parentComponent, Object message)` Brings up a modal dialog with the options Yes, No and Cancel; with the title, "Select an Option".
`static int`	**`showConfirmDialog`**`(Component parentComponent, Object message, String title, int optionType)` Brings up a modal dialog where the number of choices is determined by the `optionType` parameter.

Remarquez la présence de mot clé *static* devant la déclaration de la méthode.

D.4. Modificateur static

Ce modificateur sera donc appliqué à un champ ou une méthode :

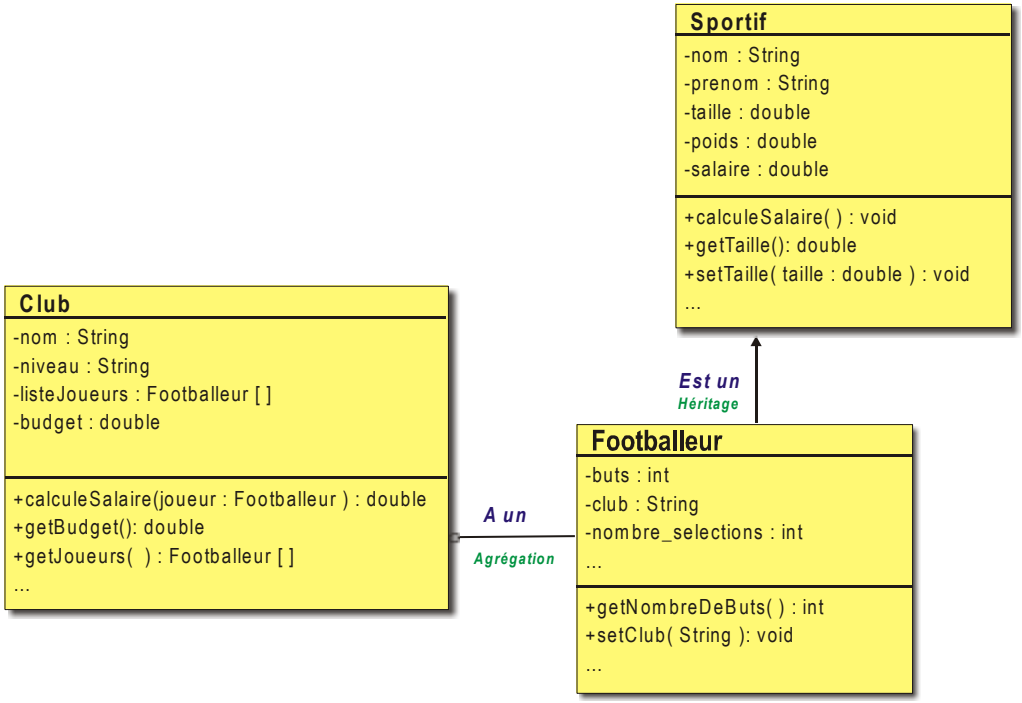
```
public static int CHAMP_STATIQUE;
```

ou pour une méthode :

```
public static void methodeStatique() { //… };
```

E. Agrégation et héritage

L'agrégation est la possibilité qu'un objet puisse contenir d'autres objets. Nous parlerons alors d'agrégation.
Dans ce cas l'objet contenu appartient à l'objet contenant et devient un champ de cette classe. Il se déclare donc comme un champ avec les mêmes règles de l'encapsulation grâce aux modificateurs ***private, protected*** et ***public***.
Imaginons que notre classe *Club* puisse contenir un tableau d'objets *Footballeur*
.



Exemple de relation d'agrégation entre Footballeur et Club

Pour l'héritage , nous dirons qu'un footballeur ***"est un"*** sportif. Pour l'agrégation, nous dirons que le club ***"a un"*** footballeur (ou plusieurs)

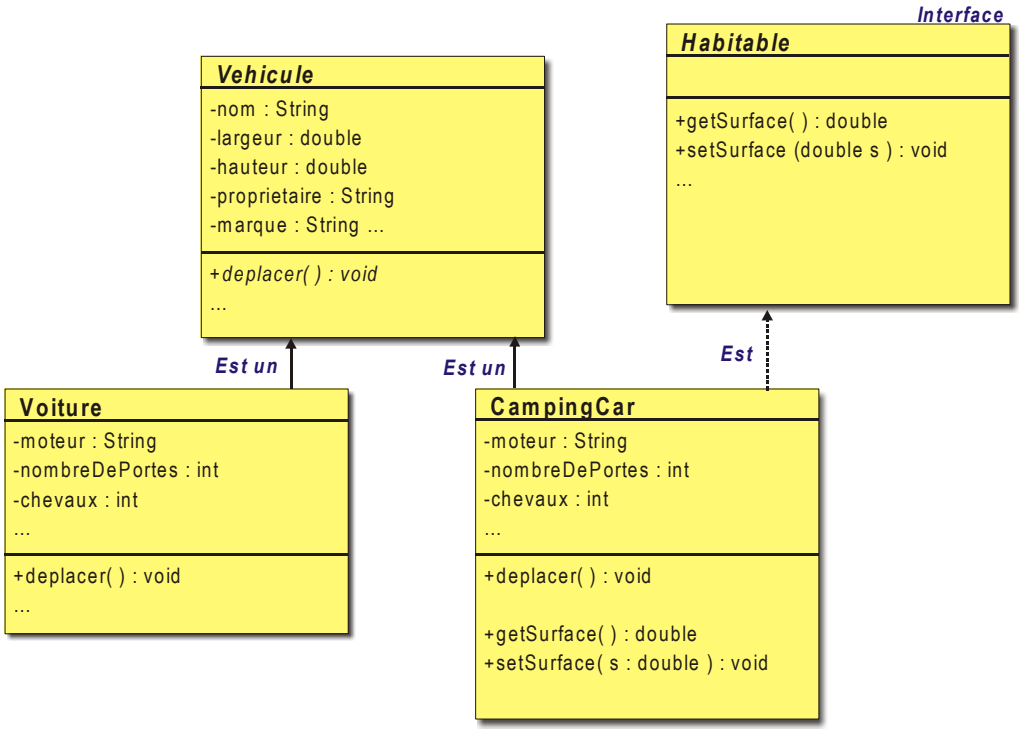
F. Les interfaces

F.1. Héritage multiple

Dans les langages orientés objets, la notion d'héritage multiple stipule qu'une classe peut posséder plusieurs parents. En C++, par exemple, cette fonctionnalité est disponible.
L'héritage multiple présente de nombreux avantages d'un point de vue conceptuel. En effet, il est plus facile de construire ses propres objets en utilisant un travail déjà effectué sur les objets parents.
Il faut savoir que l'héritage rend les compilateurs soit, peu performants ou soit complexes.
En JAVA, l'héritage multiple n'existe pas mais les interfaces en reprennent les principaux avantages. Par contre, leur emploi est plus simple, mais on ne sait pas toujours quand les utiliser et surtout pourquoi

F.2. Pourquoi utiliser des interfaces

Nous avons vu dans les chapitres précédents les différentes fonctionnalités des objets JAVA. Vous avez pu constater que la construction de ses propres classes n'est pas simple surtout si vous voulez obtenir du code de qualité professionnelle et réutilisable. A part pour les classes abstraites, les objets que nous avons abordé se contentaient de représenter des fonctions concrètes *(Footballeur, Sportif, CompteBancaire* ..).
Maintenant, il est temps de modéliser des fonctions un peu plus abstraites (Un *Footballeur* est-il *Transferable* ou encore un *Fruit* est-il *Stockable* ?).
Les interfaces peuvent répondre à toutes ces questions. Elles traduisent la capacité d'un objet à adopter un comportement en plus de ce que son parent peut lui fournir :



Dans le schéma ci-dessus, la classe *CampingCar* est une classe *Vehicule* mais elle est également *Habitable*. Vous remarquerez également que *CampingCar* possède toutes les méthodes de l'interface *Habitable*.

En fait, on dit que CampingCar implémente l'interface Habitable

F.3. Caractéristiques des interfaces

Une interface est une promesse que toutes les classes qui implémenteront l'interface implémenteront également toutes les méthodes de cette interface. Evidemment, la façon dont fonctionne les méthodes va dépendre de chaque classe.
L'intérêt des interfaces est que toute classe qui l'implémente devra adopter un comportement identique même si chacun aura un façon différente de le faire.

a. Déclarer une interface

Pour déclarer une interface, il suffit de créer un nouveau fichier contenant le mot clé ***interface*** :

```
public interface Habitable {

public void setSurface( double s);
public double getSurface();

}
```

b. Utiliser une interface

Pour utiliser ou implémenter une interface, il suffit d'utiliser le mot clé ***implements*** :

```
public class CampingCar extends Vehicule implements Habitable  {

public void setSurface( double s)
{
// Contenu de la méthode
```

```
}
public double getSurface() {
// Contenu de la méthode
}

// Reste de la classe
}
```

Il est possible d'ajouter des champs à l'intérieur des interfaces. Néanmoins, ces champs devront être des champs statiques et non modifiables grâce aux modificateurs static et final

Une classe peut également implémenter plusieurs interfaces. On met alors des virgules entre chaque nom d'interface :

```
public class CampingCar extends Vehicule
                         implements Habitable, Cloneable  {

// Contenu de la classe
}
```

c. Exemple dans l'API JAVA

Voici quelques exemples de l'utilisation des interfaces dans les API de JAVA. L'interface *Cloneable* est une interface qui précise, grâce à la méthode *clone()* que l'objet en question peut être cloné en un autre objet identique.
Evidemment, si par exemple, je souhaite cloner un objet de type *Voiture* et que la classe *Voiture* implémente l'interface *Cloneable* , le compilateur me fera utiliser la méthode *clone()* surchargée pour l'occasion. Ce qui a pour conséquence que seul un objet de type *Voiture* sait comment se cloner :

```
Voiture vt = new Voiture ("ZX",108);

Voiture vt2 = (Voiture) vt.clone();
// Transtypage nécessaire car la méthode clone() renvoie
// un objet de type Object
```

En règle générale, il vaut mieux masquer la méthode clone( ) plutôt que de laisser celle de Object faire le clonage. Tout simplement parce que la classe Object ne sait rien de l'objet et va se contenter d'effectuer une copie binaire de l'objet. Si l'objet ne contient aucune autre référence à d'autres objets, tout se passera bien. Par contre, si l'objet contient d'autres objets, les deux clones feront référence aux mêmes objets.

Par défaut, l'appel à la méthode clone() déclenche une exception. Nous reverrons le principe des exceptions. Pour activer cette méthode, il suffit d'implémenter l'interface Cloneable et de masquer la méthode clone(). Mais il faut aussi intercepter l'exception CloneNotSupportedException.

F.4. Conclusion

Même si l'intérêt et les fonctionnalités des interfaces ne vous semblent pas très clairs, nous y reviendrons en détail lorsque nous aborderons la gestion des évènements qui en est friande.

G. Classes internes

Il s'agit de classes définies à l'intérieur d'autres classes (Dans le même fichier). Voici une liste des caractéristiques et des avantages que procurent les classes internes :

- Les objets définis dans la classe interne peuvent accéder aux méthodes et champs privés de la classe d'appartenance comme ils en faisaient partie.
- Les classes internes peuvent être cachées des autres classes du même package
- Les classes internes anonymes (Sans nom) sont très pratiques pour la gestion des évènements.

G.1. Principe

La définition de la classe interne doit être placée au sein de la classe d'appartenance (Avant la dernière accolade de la classe).
De nombreux développeurs utilisent les classes internes en relation avec les interfaces. Dans la gestion des évènements , un objet qui doit provoquer un événement se référence auprès d'un autre objet appelé "l'écouteur". L'écouteur, sait réagir aux évènements qui sont provoqués par l'objet écouté.

G.2. Exemple

Voici l'exemple d'une fenêtre de type *JFrame* et de la classe interne *WindowsCloser* qui étend la classe abstraite *WindowAdapter* :

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

public class ExH6_1 extends JFrame  {

public ExH6_1() {          // Constructeur par défaut

            setTitle("Exemple de classe interne");
            setSize(300,200);
            // Référencement de la fenêtre auprès de l'objet écouteur
            addWindowListener( new WindowCloser() );
}


// Programme principal
static void main (String[] args) {

            ExH6_1 myFrame = new ExH6_1();
            myFrame.show();
            }

// Classe interne
public class WindowCloser extends WindowAdapter {

public void windowClosing ( WindowEvent e ) {
            System.exit(0);
            }
}
}
```

ExH6_1.java

Cet exemple propose d'afficher une fenêtre et de quitter l'application lorsqu'on clique sur la croix à droite de la fenêtre. En effet, ceci est très utile car pour un objet de type *JFrame* , le fait de cliquer sur la croix ne quitte pas vraiment l'application mais cache la fenêtre. Le programme reste tout de même en mémoire. Avec cet exemple, l'application est réellement quittée lorsqu'on clique sur la croix.

G.3. Conclusion

Nous ne nous attarderons pas sur les concepts de gestion des évènements puisqu'il s'agira d'un chapitre complet du polycopié ***Programmation d'interfaces graphiques utilisateur*** . pour cet exemple, retenez simplement la possibilité de définir une classe à l'intérieur d'une autre classe.
Notez également la possibilité à cette classe d'accéder aux diverses implémentations de la classe à laquelle elle appartient (Méthodes, champs).

H. Les collections

Les collections sont des objets dont le rôle est de contenir d'autres objets sous forme d'ensembles cohérents. Les collections sont très différentes des tableaux car elles n'ont pas de taille définitive. En effet, contrairement à un tableau, une collection peut contenir un nombre non fini d'objets de type différents.

H.1. Qu'est-ce qu'une collection ?

Une collection permet de traiter plusieurs objets comme un seul ensemble. Il en résulte une certaine « généricité » permettant de manipuler simplement des objets de types et de complexité différents.
Imaginons que nous ayons à traiter un ensemble de chaines de caractères. Nous utiliserons donc la classe *String*. Pour stocker, trier et gérer un ensemble de chaînes contenant la liste des jours de la semaine, nous utiliserons une classe *ArrayList*

```
package javacsharp;

import java.util.ArrayList;
import java.util.Iterator;


public class ExempleCollections {

    public static void main(String[] args) {

        ArrayList maCollection = new ArrayList();

        // Ajoute Lundi, mardi, mercredi
        maCollection.add("Lundi");
        maCollection.add("Mardi");
        maCollection.add("Jeudi");

        // Ajoute Mercredi en 3eme position
        maCollection.add(2,"Mercredi");
        maCollection.add("Vendredi");
```

```
        maCollection.add("Samedi");
        maCollection.add("Dimanche");

        maCollection.add("toto");

        // Retire le dernier élément de la collection
        maCollection.remove("toto");

        /* Classe Iterator est une classe d'aide permettant
         * de parcourir la collection
         */
        Iterator monIt = maCollection.iterator();

        // Affiche à l'écran le contenu de la collection
        while (monIt.hasNext()) {
            String cur = (String) monIt.next();
            System.out.print(cur+", ");
        }
    }
}
```

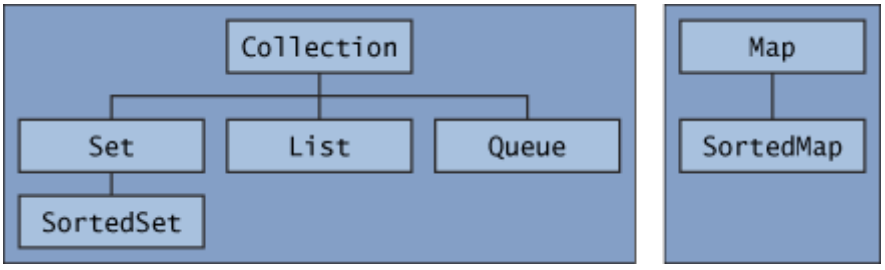
ExempleCollections.java

Dans l'exemple précédent, nous remarquons plusieurs choses :

- La classe *ArrayList* est une classe gérant les collections
- Il n'est pas nécessaire de définir une taille pour la collection. Le constructeur par défaut fixe la taille à 10 éléments et agrandit la taille en fonction des besoins.
- On peut ajouter et retirer des éléments.
- La classe *Iterator* est utile pour parcourir cette collection. C'est une classe permettant d'extraire chaque élément l'un après l'autre pour l'affichage
- Une collection est un ensemble d'objets de type *Object*. Il faut donc effectuer un transtypage vers *String* pour récupérer l'objet d'origine. Cette contrainte a d'ailleurs été levée à partir du JSE 1.5, ce que nous verrons plus tard.

H.2. Le framework des collections

Voici les classes et interfaces constituant le framework des collections en Java :



Le framework des collections Java

a. Collection

Racine de la hiérarchie du framework, l'interface *Collection* défini la manière de gérer un ensemble d'éléments. Il n'y a pas d'implémentation directe de cette interface mais il existe beaucoup de sous-interface ou de classes abstraites. Cette interface dispose de méthodes de base permettant la manipulation de ses éléments.

b. Set

Collection qui contient des éléments qui doivent être uniques. Cette interface ne supporte pas les doublons. Elle dispose du même jeu de méthodes qui ont été redéfinies pour éviter les doublons.
L'interface *SortedSet* s'arranger pour permettre un parcours de la collection selon l'ordre naturel des objets contenus.

c. List

Collection ordonnée d'éléments autorisant la duplication. L'utilisateur d'une *List* offre d'avantage de contrôle sur l'insertion de nouveau éléments. L'accessibilité aux éléments peut également se faire via un entier déterminant sa position dans la collection.
Les classes *Vector* et *ArrayList* sont des classes très utilisées par les développeurs.

> Il convient d'éviter d'utiliser la classe Vector qui offre des performances moyennes sur des grosses collections. Vector est une version conçue différemment à l'origine. Elle a été adaptée à List après sa conception

d. Queue

L'interface *Queue* est plus centrée sur la façon dont les éléments sont insérés et retirés plutôt que sur les performances de traitement. La plupart des collections basées sur *Queue* utilise la méthode FIFO (First In First Out) pour gérer l'insertion et la suppression d'éléments.
Les classes *LinkedList* ou encore *PriorityQueue* sont des classes basées sur *Queue*. Elles sont utilisées principalement pour gérer des éléments selon une priorité ou un ordre précis.
On utilisera en priorité la méthode *offer* plutôt que *add* pour ajouter des éléments dans ce type de collection.

e. Map

Type de collection permettant de stocker les éléments sous forme de paires « nom-valeur ». Chaque élément porte un nom unique. Ce nom est utilisé pour accéder à l'élément. Le nom de l'élément est appelé la clé (key)
Ce type de collection est très utile pour accélérer les recherches sur les éléments contenus puisqu'ils sont accessibles par la clé.
Notons la présence de la classe *HashMap* qui propose les mêmes fonctionnalités que les tables de hachage.

H.3. Le principal problème des collections

Le fait qu'une collection contienne des éléments qui sont des objets de n'importe quel type n'est pas sans conséquence sur la lisibilité du code et sur les sources d'erreurs. En effet, lorsqu'un l'utilisateur d'une collection ajoute des éléments, aucune vérification n'est effectuée sur le type d'objet fourni.
Reprenons le programme précédent en ajoutant autre chose que des objets de type *String*.

```
package javacsharp;

import java.util.ArrayList;
```

```
import java.util.Iterator;


public class ExempleCollections2 {

    public static void main(String[] args) {

        ArrayList maCollection = new ArrayList();

        // Ajoute Lundi, mardi, mercredi
        maCollection.add("Lundi");
        maCollection.add("Mardi");
        maCollection.add("Jeudi");

        // Ajoute Mercredi en 3eme position
        maCollection.add(2,"Mercredi");
        maCollection.add("Vendredi");
        maCollection.add("Samedi");
        maCollection.add("Dimanche");

        maCollection.add("toto");

        // Retire le dernier éléments de la collection
        maCollection.remove(7);

        // Ajoute un objet qui n'est pas de type String
        maCollection.add(new Integer(5));

        /* Classe Iterator est une classe d'aide permettant
         * de parcourir la collection
         */
        Iterator monIt = maCollection.iterator();

        // Affiche à l'écran le contenu de la collection
        while (monIt.hasNext()) {
            String cur = (String) monIt.next();
            System.out.print(cur+", ");
        }

    }
}
```

ExempleCollections2.java

Une exception se produit pendant l'exécution de la ligne 39 :

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer
  at javacsharp.ExempleCollections2.main(ExempleCollections2.java:39)
```

Lors que la boucle essaie de transtyper le dernier élément en objet de type *String*, une erreur de type *ClassCastException* se produit ce qui est tout à fait normal étant donné qu'il s'agit d'un objet de type *Integer*.

Un utilisateur de collection doit donc veiller au type d'objet présent dans les collections et faire un transtypage correct des objets présents.

Face à cet inconvénient, les concepteurs de Java ont mis en place dans la version 1.5 un lot d'améliorations permettant de typer les collections : les génériques

I. Les « generics »

I.1. Introduction

Arrivés avec la version 1.5 du J2SE, les « generics » proposent de nouvelles fonctionnalités destinées à réduire la taille du code et augmenter sa « maintenabilité ». Voici une liste des fonctionnalités proposées :

- Ajout des « types paramétrés » aux classes et aux méthodes
- Nouvelle méthode de manipulation des collections
- Typage des collections
- Utilisation étendue des « wildcards » pour émettre des restrictions sur les types utilisés lors de l'instanciation

I.2. Le principe

Les « generics », également appelés « types paramétrés », permettent d'éviter les transtypages nécessaires à la récupération des objets dans les collections. En effet, les collections en java sont implémentées autour des « Object ». Avant l'arrivée des « generics », il fallait effectuer un transtypage (Casting) pour récupérer l'objet :

```
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.iterator().next();
```

- **Ligne 1** : Déclaration et instantiation d'une collection d'objets
- **Ligne 2** : Ajout d'un objet de type *Integer* à la collection
- **Ligne 3** : Récupération du 1er objet de la collection nécessitant un transtypage en *Integer* .

Avec les « generics », le type d'objet contenu dans la collection est défini lors de l'instanciation de la collection. Le type choisi est mis entre < > :

```
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

- Ligne 1 : La présence de <*Integer*> permet de typer la collection
- Ligne 3 : Il n'est plus nécessaire de transtyper pour récupérer l'entier

Hormis les avantages évoqués ci-dessus, les « generics » permettent de détecter les problèmes de type au moment de la compilation. En effet, **une erreur de transtypage se produit nécessairement pendant l'exécution du programme**. Donc, si un développeur ne teste pas tout son code, l'erreur risque de se produire chez l'utilisateur du programme, ce qui pose d'énormes problèmes de maintenance du code.
Voila un programme qui commet volontairement une erreur d'ajout d'un entier dans une collection typée en « String » :

```
public class ExempleGenerics {

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

```
        ArrayList<String> maCollection = new ArrayList<String>();

        // Ajoute Lundi, mardi, mercredi
        maCollection.add("Lundi");
        maCollection.add("Mardi");
        maCollection.add("Jeudi");

        maCollection.add(new Integer(4));    //ERREUR DE COMPILATION

        /* Classe Iterator est une classe d'aide permettant
         * de parcourir la collection
         */
        Iterator monIt = maCollection.iterator();

        // Affiche à l'écran le contenu de la collection
        while (monIt.hasNext()) {
            String cur = (String) monIt.next();
            System.out.print(cur+", ");
        }
    }
}
```

ExempleGenerics.java

Voici l'erreur de compilation provoquée :

```
"ExempleGenerics.java": impossible de trouver le symbole ; symbole  : méthode add(java.lang.Integer),
emplacement : classe java.util.ArrayList<java.lang.String> at line 18, column 22
```

L'erreur précédente indique que la méthode *add(java.lang.Integer)* prenant comme paramètre un objet de type *Integer* est inconnue.

L'erreur se produit donc à la compilation et non à l'exécution, ce qui évite au développeur des erreurs inévitables de transtypage dans des projets de grande échelle.

I.3. Parcourir les collections typées

Avec les « generics », il est possible de parcourir une collection en utilisant une forme spéciale de boucle « for » :

```
for ( ClassType variable : collection) {… }
```

Valeurs des paramètres :

- ClassType : Le type de variable
- Variable : Le nom de la variable dans la boucle
- Collection : L'objet collection à parcourir

Voici un exemple de parcours d'une collection à l'aide d'une boucle « for » :

TestCollections.java

```
1
2 package org.cours.n1.generics;
```

```
 3
 4 import java.util.ArrayList;
 5
 6
 7 public class TestCollections {
 8
 9
10
11  public static void main(String[] args) {
12
13    ArrayList <String> maCollection = new ArrayList <String>();
14
15    maCollection.add("chaine1");
16    maCollection.add("chaine2");
17    maCollection.add("chaine3");
18
19          for (String c : maCollection) {
20               System.out.println(c);
21          }
22     }
23 }
24
```

- **Ligne 13** : Déclaration d'une collection d'objets « String »
- **Ligne 19** : Boucle de parcours de la collection. Le code est très simple et aucune erreur de transtypage ne peut se produire à l'exécution

I.4. Les « generics » et le transtypage

Comment se comporte une collection avec type paramétré vis-à-vis du transtypage ?
Voici un petit programme qui tente de répondre à cette question :

```
package javacsharp;

import java.util.ArrayList;
import java.util.Iterator;


public class ExempleGenericsSubtype {

    public static void main(String[] args) {

        ArrayList<String> maCol1 = new ArrayList<String>();
        ArrayList<Object> maCol2 = maCol1;
    }
}
```

ExempleGenericsSubtype.java

- **Ligne 12** : Les deux types sont incompatibles. L'un fait référence à une collection de *String* alors que l'autre est une collection de « *Object* ». Même si *String* est un *Object* , la réciproque n'est pas vraie.

I.5. Définir des classes « génériques »

Les classes « génériques » acceptent de travailler avec des objets dont le type n'est pas encore connu au moment de la compilation. Cela permet de définir des méthodes travaillant avec un contenu « typé » dont le type réel sera connu plus tard.

Le code avec « generics » est dont potentiellement plus léger puisque le code est conçu pour travailler avec tout type d'objet. Il n'est donc plus nécessaire de prévoir un code différent pour chaque type stocké dans les collections.

Voici la définition d'une interface destinée à travailler avec un type d'objets non défini à l'avance :

```
public class MaClasse <E> {

public void add( E x) {
…
}
public E get() {
…
}
}
```

Le « <E> » signifie : Tout type d'objet déclaré lors de l'instanciation

Le type définitif de la classe sera déterminé lors de l'instanciation :

```
MaClasse <String> aMaClasse = new MaClasse<String> ();
```

Ou

```
MaClasse <Integer> aMaClasse = new MaClasse<Integer> ();
```

I.6. Définir des méthodes génériques

Il est possible d'appliquer le même principe aux méthodes afin de les rendre génériques. Dans ce cas, il suffit de préciser le type générique d'objet dans la signature de la méthode :

```
public <T> void maMethode(Collection T) { … }
```

Imaginons que nous souhaitions implémenter une méthode statique capable de copier un tableau depuis une collection (Le rêve de tous les développeurs Java…). La déclaration de la méthode ressemblerait à ceci :

```
public static <T> void arrayToCollection(T[] tableau , ArrayList<T> col) { //
Contenu de la méthode
```

```
}
```

Voici un exemple de programme utilisant une méthode générique :

TestCollections.java

```
1  package org.cours.n1.generics;
2
3  import java.util.ArrayList;
4
5  public class TestCollections {
6
7      public static <T> void arrayToCollection(T[] tableau , ArrayList<T> col) {
8          // Boucle de parcours du tableau
9          for (T element : tableau) {
10             // Ajout de chaque élément à la collection
11             col.add(element);
12         }
13     }
14
15     public static void main(String[] args) {
16
17         // Définition et instanciation d&apos;une collection de chaine
18         ArrayList <String> maCollection = new ArrayList <String>();
19
20         // Création du tableau à transférer
21         String[] tableau = {"chaine1","chaine2","chaine3"};
22
23         // Appel de la méthode permettant le transfert
24         arrayToCollection(tableau,maCollection);
25
26         // Boucle d&apos;affichage de la collection
27         for (String c : maCollection) {
28             System.out.println(c);
29         }
30     }
31 }
```

I.7. Restrictions sur les « generics »

a. Les « generics » face à l'héritage

L'héritage, qui consiste à faire bénéficier à une classe enfant, des attributs ou méthodes de son parent peuvent amener une certaine confusion lorsqu'il est appliqué aux « generics ». En effet, lorsqu'on qu'une classe « Conducteur » hérite de la classe « Personne », on considère à raison qu'un conducteur « est une » personne.

Pour les « generics », une collection de « Conducteur » n'est pas une collection de « Personne » car toutes les personnes ne sont pas des conducteurs. On ne peut donc pas les considérer de la même manière. Le traitement qui consisterait à envoyer à la préfecture une collection de « Personne » serait dangereux car toutes les personnes seraient candidates pour le permis de conduire alors que seuls les conducteurs y ont droit.

Voici comment le compilateur réagit lorsqu'on essaie d'afficher une liste de conducteurs, alors que la méthode « afficher » est seulement capable de traiter une collection de personnes :

AfficheurDePersonne.java

```
1  package org.cours.n1.generics;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class AfficheurDePersonne {
7
8      // Affiche la collection de personnes
9      public static void afficher(List <Personne> personnes) {
10         for (Personne p : personnes) {
11             System.out.println(p);
12         }
13     }
14
15     public static void main(String[] args) {
16
17     // Collection de personnes
18     List <Personne> pliste = new ArrayList<Personne>();
19     // Collection de conducteurs
20     List <Conducteur> cliste = new ArrayList<Conducteur>();
21
22     pliste.add(new Personne("VALJEAN","Jean"));
23     cliste.add(new Conducteur("CALMAN","Jeanne",1212354));
24
25     //OK
26     afficher(pliste);
27     // ERREUR DE COMPILATION
28     afficher(cliste);
29     }
30 }
```

Donc, malgré le fait **qu'un conducteur est aussi une personne**, **les « generics » empêchent l'affichage d'une collection de conducteurs**. Nous allons voir que dans certains cas, il peut être utile d'élargir ou de restreindre les types génériques supportés avec **le principe des « wildcards »**.

b. Le principe des types « wildcards »

Le principe des « generics » étant de pouvoir choisir le type utilisé au moment de l'instanciation, il peut être utile de limiter, voire d'étendre le type utilisable. Dans un cas, on autorisera tout type d'objet et dans l'autre, on limitera à certaines classes.

Par exemple, nous pourrions définir une collection de tout type avec le type < ? > :

```
public void afficher(Collection <?> c) {
    for (Object e : c) {
        System.out.println(e) ;
    }
}
```

Le type d'objets contenu dans la collection pourra être n'importe lequel. Nous pouvons caractériser la collection par le terme « collection de type inconnu ».

Le code de la méthode « afficher » de l'exemple précédent devrait être réécrit de la manière suivante :

AfficheurDePersonne.java

```
1      // Affiche la collection de personnes
2      public static void afficher(List <?> personnes) {
3          for (Object p : personnes) {
4              System.out.println(p);
5          }
6      }
```

ATTENTION cependant à l'utilisation abusive du caractère < ? >. Que se passerait-il si le développeur utilisait d'autres types que « Conducteur » ou « Personne » ? Cela produirait des incohérences à l'affichage et à terme, des erreurs d'exécution. Le problème des collections non typées se repose comment avant l'arrivée des « generics »

c. Restreindre les « wildcards »

Afin de résoudre ce problème d'utilisation abusive des « wildcards », il est possible de restreindre les types utilisés. De cette manière, on réserver l'usage d'une méthode ou d'une classe à une partie de la chaîne d'héritage.

```
public static void afficher(List <? extends Personne> personnes)
```

Le paramètre fourni sera donc **obligatoirement une classe dérivée de « Personne »**. Les instances de « Conducteur » seront donc acceptées.

ATTENTION : Il y a un prix à payer pour l'utilisation de cette syntaxe. Dès lors que l'on restreint les types de classes, on perd la possibilité de modifier la collection passée en paramètres dans le corps de la méthode. Le code suivant provoquera donc une erreur de compilation.

```
// Affiche la collection de personnes
public static void afficher(List <? extends Personne> personnes) {
    personnes.add(new Personne("HUGO","Victor")); //ERREUR DE COMPILATION
    for (Object p : personnes) {
        System.out.println(p);
    }
}
```

Pourquoi perd t on la possibilité de modifier la collection passée en paramètres ?

Dans l'exemple, on tente d'ajouter un objet « Personne » alors même qu'on ignore ce que la collection contient : des objets « Personne » ou des objets « Conducteur ». Le compilateur ne peut pas prendre le risque d'ajouter des objets hétérogènes à une collection.

N'oublions pas que, même si les « generics » nous autorisent à choisir le type d'une collection, dès lors que ce type est choisi, il n'est pas possible d'y ajouter un autre type d'objet

Chapitre 9 : Les exceptions

A. Introduction

A.1. Un monde parfait

Imaginons un monde utopique où les programmes informatiques comme les ordinateurs fonctionnent parfaitement avec une compatibilité de tout les instants. En poussant un peu plus loin notre imagination, considérons que les utilisateurs se comportent de manière prévisible, raisonnable et avertie.
Evidemment, ce monde n'existe pas et les sources d'erreurs d'exécution sont multiples. En effet, un fichier qui n'existe pas, une division par zéro, une base de données qui ne répond pas ou encore une connexion réseau qui se termine prématurément, sont autant de causes d'échecs qui feront de votre programme, un chemin de croix pour l'utilisateur

A.2. Qu'est-ce qu'une exception ?

Les exceptions correspondent à des situations où le compilateur ne décèle pas d'erreur conceptuelle ou de syntaxe mais où pourtant, il n'y a aucune certitude sur l'exécution du programme. Par exemple, lors d'une division de 2 nombres, si le diviseur est égal à zéro, une exception se produit et le programme ne peut continuer à s'exécuter étant donné que la division par zéro est interdite. Mais il existe beaucoup de situations où l'on ne peut prévoir le résultat d'une action. Les accès aux fichiers illustrent bien l'incertitude dans le cas où le fichier désigné est déjà ouvert par un autre utilisateur ou qu'il a été effacé.

A.3. Inconvénients

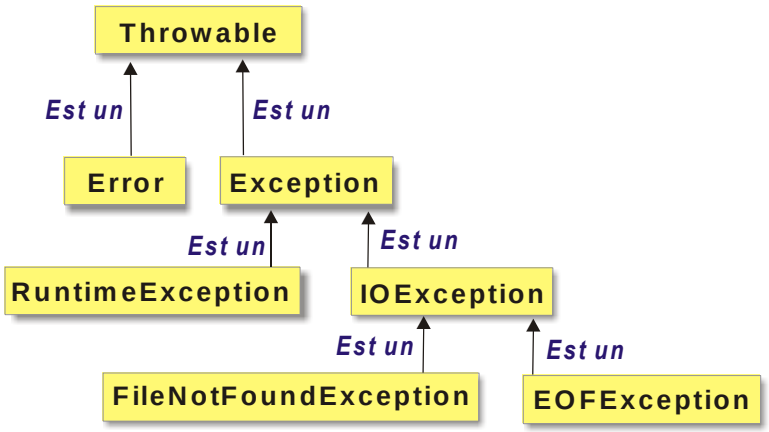
L'inconvénient d'une exception lorsqu'elle se produit est l'arrêt sans autre forme de procès du programme. Le mécanisme des exceptions en JAVA, va vous permettre d'éviter ce genre de désagrément par un travail préparatoire basé sur la prévention plutôt que la guérison.

B. Mécanisme des exceptions

En JAVA, les exceptions sont modélisées par des objets. Chaque exception d'un type particulier est caractérisée par un gestionnaire d'exception qui connaîtra la marche à suivre lorsque l'exception va se produire.
Par exemple, lorsqu'un utilisateur sauvegarde le fruit de son travail, il peut arriver que le disque dur soit plein et ne puisse accepter des données supplémentaires. Le gestionnaire d'exception adapté, doit lui permettre, dans une telle situation, de sauvegarder ses données ailleurs et lui laisser le loisir d'effacer d'autres fichiers pour faire de la place.

B.1. Classes d'exceptions

Toutes les classes d'exception sont issues de la classe *Throwable*.



Exemples de classes d' exception en JAVA

Dans le schéma très partiel ci-dessus, vous remarquerez que chaque exception correspond à un besoin spécifique. Plus on descend dans la hiérarchie et plus l'erreur à traiter est spécifique.

> *En général, les exceptions de type RuntimeException ne sont pas traitées car il s'agit souvent d'une erreur de programmation que le développeur peut résoudre en modifiant son programme (Un dépassement d'indice pour un tableau par exemple). Par contre les exceptions de type IOException correspondent à une situation inattendue indépendante de programme lui même (Un fichier qui n'existe pas ou encore un serveur web qui ne répond pas)*

B.2. Méthodes générant des exceptions

Si, à l'intérieur d'une méthode, vous jugez qu'une action peut mal se passer, il faut permettre à cette méthode de lancer une exception. L'en-tête de la méthode a la syntaxe suivante :

```
public String readData(BufferedReader in) throws EOFException {
// …
}
```

Ici, la méthode *readData* , chargée de lire dans un fichier , peut échouer parce la lecture du fichier s'est terminée prématurément . L'emploi du mot *throws* permet à cette méthode de prévenir qu'elle est susceptible de renvoyer une exception.

La conséquence directe est que tout programmeur désireux d'utiliser votre méthode devra intercepter cette exception et prévoir le code à exécuter si jamais cette exception se produit. Nous allons voir plus loin la manière d'intercepter les exceptions avec le bloc try…catch.

Revenons à notre méthode où nous pouvons décider à quel endroit il faut lancer une exception :

```
public String readData(BufferedReader in) throws EOFException {

while ( n<tailleDuFichier )
{
     if (ch==-1)          // Se produit si la fin de fichier atteinte
            if (n < tailleDuFichier)
                 throw new EOFException(); // Lance l'exception
}
```

L'instruction de la ligne 7 permet de lancer une exception si l'entier *n* ne correspond pas à la taille réelle du fichier. Rencontrer une fin de fichier alors que le nombre de caractères lus ne correspond pas au nombre réel, revient à dire que la lecture a été interrompue inopinément (Le fichier a subi des dommages ou a été modifié…).

Le programme ayant appelé votre méthode sera alors informé de l'exception et devra réagir en conséquence. Par exemple, il devra informer l'utilisateur du problème rencontré. En aucun cas, la méthode ne doit informer l'utilisateur car ça n'est pas son rôle et la plupart du temps, elle ne le peut pas. Il faut laisser le mécanisme des exceptions faire son travail.

Dans certains cas, il peut être utile d'informer le code appelant avec des informations plus pertinentes :

```
public String readData(BufferedReader in) throws EOFException {

while ( n<tailleDuFichier )
{
     if (ch==-1)          // Se produit si la fin de fichier atteinte
            if (n < tailleDuFichier)
                 {
                 String mess = "Echec au " + n + " ième caractère";
                 throw new EOFException(mess); // Lance l'exception
}
```

Un constructeur de la classe *EOFException( )* prenant comme argument une chaîne de caractère permet de passer l'information.

B.3. Intercepter les exceptions

Voyons maintenant ce qui se passe du côté de l'appelant. L'appelant est le morceau de programme chargé d'appeler la méthode en question. Lorsque le compilateur rencontre une méthode d'un objet capable de renvoyer une exception, il va vérifier que vous avez tout prévu dans le cas où l'exception se produit.

Si vous n'interceptez pas l'exception, la compilation va échouer avec un message d'erreur. Vous pouvez le prévoir en implémentant un bloc de type *try…catch…finally*.

a. Bloc try..catch

Ce bloc doit comporter les mots clés *try* , *catch* et éventuellement *finally*. Nous pouvons résumer en une phrase :

Essaie ce bout de code et attrape l'exception si elle se produit.

```
try {
            votre code pouvant provoquer l'exception
 } catch (ExceptionType e)
 {
   Code gérant l'exception
 }
```

Si une exception du type spécifié dans le bloc *catch* est lancée dans le bloc *try* , le programme exécute le code du bloc *catch*. Par contre si aucune exception n'est lancée, le programme ignore la clause *catch*.

b. Lever plusieurs exceptions

Certaines actions peuvent lancer plusieurs types d'exceptions. Le programmeur soigneux aura en charge de lever chaque exception avec plusieurs clause *catch* :

```
try {
            //code pouvant lancer des exceptions
 }
  catch ( MalformedURLException e1)
 {  // Action pour les URL incorrectes
 }
  catch (UnknownHostException e2)
 {  // Action pour les hôtes inconnus
 }
  catch (IOException e3)
 {  // Action pour les autres cas
 }
```

Dans le cas où plusieurs exceptions sont levées, il faut toujours prendre soin d'ordonner les blocs catch en allant de l'exception la plus spécifique à la plus générale (Dans l'ordre montant de la hiérarchie des classes).

c. S'informer sur l'origine de l'exception

Toute exception, dispose d'une méthode *getMessage()* permettant d'obtenir un message d'information sur l'origine de l'exception :

```
System.out.println(e3.getMessage());
```

d. Relancer les exceptions

Dans la plupart des cas, vous ne pourrez pas traiter complètement l'origine de l'exception. Il faudra alors la relancer de manière à informer le code appelant que l'exception s'est produite mais que vous avez partiellement réglé le problème. Le code appelant devra alors traiter l'exception avec les informations que vous lui avez donné.

```
try {
            code pouvant lancer des exceptions
 }
  catch ( MalformedURLException e1)
 {
            … traitement relevant de votre compétence
            throw e1;  // Vous passez le relais au code appelant
 }
```

B.4. Clause finally

Lorsqu'une exception se produit, elle interromps la méthode et sort de celle-ci. Si votre méthode utilise des ressources telles que la mémoire ou un fichier, elles ne seront pas libérées et votre méthode est la seule habilitée à les gérer.
La clause *finally* fixe le problème en proposant un bloc de code qui sera exécuté quoiqu'il arrive, qu'une exception ait été levée ou non :

```
try
{
            // Code susceptible de lancer une exception
} catch (IOException e)
{
            // Code traitant l'exception
} finally
{
            // Code s'exécutant de toute façon
}
```

Notez que si le bloc try n'est pas suivi d'un bloc catch , il faudra prévoir un bloc finally sous peine de provoquer une erreur de compilation

C. Créer ses propres classes d'exception

Il peut arriver qu'aucune exception disponible dans les API JAVA ne vous convienne et qu'il vous faille les créer. Dans tous les cas, vous aurez à charge d'hériter d'une classe d'exception existante pour créer la votre.
Imaginez que vous souhaitiez créer une classe d'exception capable de générer une exception lorsque vous constatez que le fichier que vous êtes en train de lire n'a pas le bon format. La création d'une classe *BadFileFormatException* serait judicieux :

```
class BadFileFormatException extends IOException
{
 public BadFileFormatException() {}
 public BadFileFormatException( String gripe )
            { super(gripe);
            }
}
```

Il suffit maintenant de lancer votre exception si nécessaire :

```
String readFile(BufferedReader in) throws BadFileFormationException
{
 // Code effectuant la lecture dans le fichier
if ( badFormat == true )
            throw new BadFileFormatException();
}
```

Le code appelant n'aura qu'à traiter l'exception :

```
try
{
  myFile.readFile(in);
} catch { BadFileFormatException e)
{
```

```
 // Code chargé de traiter l'exception
}
```