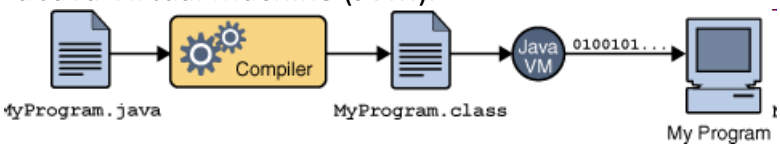
Chapitre I
Les bases du langage Java

Caractéristiques

- Java est interprété
- Le source est compilé en pseudo code (byte code) puis exécuté par un interpréteur Java
  - La Java Virtual Machine (**JVM**).



- Java est portable :
  - Indépendant de toute plate−forme
  - Pas de compilation spécifique pour chaque plate forme.
  - Le code reste indépendant de la machine sur laquelle il s'exécute.
  - Possibilité d'exécuter des programmes Java sur tous les environnements possédant une Java Virtual Machine.
- Java est orienté objet.
  - Comme la plupart des langages récents, Java est orienté objet.
  - Chaque fichier source contient la définition d'une ou plusieurs classes qui sont utilisées les unes avec les autres pour former une application.
- Java n'est pas complètement objet
  - il définit des types primitifs
    - entier, caractère, flottant, booléen,...
- Java est simple
  - Pas de notion de pointeurs
    - éviter les incidents en manipulant directement la mémoire,
  - Pas d'héritage multiple
- Java est fortement typé
  - Toutes les variables sont typées
  - Il n'existe pas de conversion automatique qui risquerait une perte de données.
- Java est sûr
  - La sécurité fait partie intégrante du système d'exécution et du compilateur.
- Un programme Java planté ne menace pas le système d'exploitation.
  - Il ne peut pas y avoir d'accès direct à la mémoire.

- L'accès au disque dur est réglementé dans une applet.

- Java est économe
  - Le pseudo code a une taille relativement petite car les bibliothèques de classes requises ne sont liées qu'à l'exécution.
- Java est multitâche
  - Il permet l'utilisation de threads qui sont des unités d'exécution isolées.
  - La JVM, elle même, utilise plusieurs threads.

Historique

Les principaux évènements de la vie de Java sont les suivants :

- **1995 mai** : premier lancement commercial
- **1996 janvier** : JDK 1.0.1
- **1996 septembre** : lancement du JDC
- **1997 février** : JDK 1.1
- **1998 décembre** : lancement de J2SE 1.2 et du JCP
- **1999 décembre** : lancement J2EE
- **2000 mai** : J2SE 1.3
- **2002 février** : J2SE 1.4
- **2004 septembre** : J2SE 5.0
- **2006**
  - **mai** : Java EE 5
  - **décembre** : Java SE 6.0

Application Vs Applet

Il existe 2 types de programmes avec la version standard de Java :

- Les applets
- Les applications.

- ***Une application***
  - autonome (stand alone program) est une application qui s'exécute sous le contrôle direct du système d'exploitation.
- ***Une applet***
  - une application qui est chargée par un navigateur et qui est exécutée sous le contrôle d'un plug in de ce dernier.

Principales différences entre une applet et une application

- Les applets n'ont pas de méthode main()
  - la méthode main() est appelée par la machine virtuelle pour exécuter une application.
- Les applets ne peuvent pas être testées avec l'interpréteur mais doivent être intégrées à une page HTML, elle même visualisée avec un navigateur

disposant d'un plug in sachant gérer les applets Java, ou testées avec l'applet viewer.

La syntaxe et les éléments de bases de Java

Les règles de base

- Java est sensible à la casse.
- Les blocs de code sont encadrés par des accolades { }
- Chaque instruction se termine par un '**;**'
- Une instruction peut tenir sur plusieurs lignes :

Exemple :

```
char
 code
 =
'D';
```

Les identificateurs

- Chaque objet, classe, programme ou variable est associé à un nom (Identificateur)
- Il peut se composer de tous les caractères alphanumériques et des caractères _ et $.
- Le premier caractère doit être une lettre, _ ou $.

Rappel : **Java est sensible à la casse.**

- Un identificateur ne peut pas appartenir à la liste des mots réservé du langage Java
- Pas d'espaces

Les commentaires

- Ils ne sont pas pris en compte par le compilateur donc ils ne sont pas inclus dans le pseudo code.
  a. Ils ne se terminent pas par un ;.
- Il existe trois types de commentaire en Java :
  a. Commentaire abrégé **//commentaire sur une seule ligne**
     - Exemple : int N=1; **// déclaration du compteur**
  b. Commentaire multiligne
     - *Exemple* :

```
/* commentaires ligne 1
 commentaires ligne 2 */
```

  c. Commentaire de documentation automatique
     - *Exemple*

```
/**
* commentaire de la methode
* @param val la valeur a traiter
* @since 1.0
* @return Rien
```

* @deprecated Utiliser la nouvelle méthode XXX
*/

Déclaration de variables

- Une variable possède
  - un nom,
  - un type
  - et une valeur.
- La déclaration d'une variable doit donc contenir
  - un nom
  - et le type de données qu'elle peut contenir.
- Une variable est utilisable dans le bloc ou elle est définie.
- La déclaration d'une variable permet de réserver la mémoire pour en stocker la valeur.
- Le type d'une variable peut être :
  - soit un type élémentaire dit aussi type primitif déclaré sous la forme ***type_élémentaire variable;***
  - soit une classe déclarée sous la forme ***classe variable ;***

- *Exemple* :
  - long nombre;
  - int compteur;
  - String chaine;
- Il est possible de définir plusieurs variables de même type en séparant chacune d'elles par une virgule.
  Exemple :
  - int jour, mois, annee ;
- Il est possible en une seule instruction de faire la déclaration et l'affectation d'une valeur à une variable ou plusieurs variables.
  - *Exemple* :

```
int i=3 , j=4 ;
```

Les types élémentaires

Type	***Désignation***	***Longueur***	***valeur***
boolean	valeur logique : true ou false	1 bit	true ou false
byte	octet signé	8 bits	−128 à 127
short	entier court signé	16 bits	−32768 à 32767
char	caractère Unicode	16 bits	\u0000 à \uFFFF
int	entier signé	32 bits	−2147483648 à 2147483647
Type	Désignation	Longueur	valeur
float	virgule flottante simple précision	32 bits	1.401e−045 à 3.40282e+038
double	virgule flottante double précision	64 bits	2.22507e−308 à 1.79769e+308
long	entier long	64 bits	−9223372036854775808 à 9223372036854775807

- ***Remarque***

Les types élémentaires commencent tous par une minuscule.

Initialisation des variables

Exemple :

```
int nombre; // déclaration
nombre = 100; //initialisation
```

OU

```
int nombre = 100; //déclaration et initialisation
```

- En Java, toute variable appartenant à un objet (définie comme étant un attribut de l'objet) est initialisée avec une valeur par défaut en accord avec son type au moment de la création.
  - Cette initialisation ne s'applique pas aux variables locales des méthodes de la classe.
- Les valeurs par défaut lors de l'initialisation automatique des variables d'instances sont :

Type	Valeur par défaut
boolean	false
byte, short, int, long	0
float, double	0.0
char	\u000
classe	null

L'affectation

- Le signe **=** est l'opérateur d'affectation et s'utilise avec une expression de la forme

variable = expression

- L'opération d'affectation est associative de droite à gauche :
  - il renvoie la valeur affectée ce qui permet d'écrire :

```
x = y = z = 0;
```

- Il existe des opérateurs qui permettent de simplifier l'écriture d'une opération d'affectation associée à un opérateur mathématique.

Opérateur	***Exemple***	***Signification***
=	a=10	a=10
+=	a+=10	a=a+10
-=	a-=10	a=a-10
***=**	a*=10	a=a*10
/=	a/=10	a=a/10
%=	a%=10	a=a%10
^=	a^=10	a=a^10

Attention

- Lors d'une opération sur des opérandes de types différents, le compilateur détermine le type du résultat en prenant le type le plus précis des opérandes.

Exemple

- Une multiplication d'une variable de type **float** avec une variable de type **double** donne un résultat de type **double**.
- Lors d'une opération entre un opérande **entier** et un **flottant**, le résultat est du type de l'opérande **flottant**.

Incrémentation & Décrémentation

- Les opérateurs d'incrémentation et de décrémentation sont : **n++ ++n n-- --n**

- Si l'opérateur est placé *avant* la variable (**préfixé**), la modification de la valeur est *immédiate* sinon la modification n'a lieu qu'à l'issu de l'exécution de la ligne d'instruction (**postfixé**)
- L'opérateur **++** renvoie la valeur *avant* incrémentation s'il est **postfixé**, *après* incrémentation s'il est **préfixé**.

Exemple

```
/* test sur les incrémentations prefixées et postfixées */
class test {
public static void main (String args[]) {
int n1=0;
int n2=0;
System.out.println("n1 = " + n1 + " n2 = " + n2);
n1=n2++;
System.out.println("n1 = " + n1 + " n2 = " + n2);
n1=++n2;
System.out.println("n1 = " + n1 + " n2 = " + n2);
n1=n1++; //attention
System.out.println("n1 = " + n1 + " n2 = " + n2);
} }
```

- *Résultat* :

```
int n1=0;
int n2=0; // n1=0 n2=0
n1=n2++; // n1=0 n2=1
n1=++n2; // n1=2 n2=2
n1=n1++; // attention : n1 ne change pas de valeur
```

Les structures de contrôle

Les boucles

1.

```
While ( condition )
{... // code a exécuter dans la boucle }
```

- Le code est exécuté *tant que* la condition est vraie.
- Si avant l'instruction *while*, la condition est fausse, alors le code de la boucle ne sera jamais exécuté.

2.

```
do
{...}
while ( condition )
```

- Cette boucle est *au moins exécutée une fois* quelque soit la valeur de la condition

3.

```
for ( initialisation; condition; modification)
{...}
```

Exemples :

- for (i = 0 ; i < 10; i++) { ....}
- for (int i = 0 ; i < 10; i++) { ....}
- for (; ;) { ... } // boucle infinie

- L'initialisation, la condition et la modification de l'index sont optionnels.
- Dans l'initialisation, on peut déclarer une variable qui servira d'index et qui sera dans ce cas locale à la boucle.
- Il est possible d'inclure plusieurs traitements dans l'initialisation et la modification de la boucle
  - Chacun des traitements doit être séparé par une virgule.

Exemple :

for (i = 0 , j = 0 ; i * j < 1000;i++ , j+= 2) { ....}

- La condition peut ne pas porter sur l'index de la boucle :

Exemple :

```
boolean trouve = false;
for (int i = 0 ; !trouve ; i++ ) {
if ( tableau[i] == 1 )
trouve = true;
... //gestion de la fin du parcours du tableau
}
```

Les branchements conditionnels

1.

```
if (condition) {
...
} else if (condition) {
...
} else {
...
}
```

2.

```
switch (expression) {
case constante1 :
instr11;
instr12;
break;
case constante2 :
...
default : ... }
```

- On ne peut utiliser ***switch*** qu'avec des types primitifs d'une taille maximum de 32 bits (**byte, short, int, char**).

L'opérateur ternaire :

(condition) **?** valeur−vrai **:** valeur−faux

Exemple 1 :

```
if (niveau == 5) total = 10;
else total = 5 ;
// équivalent à total = (niveau ==5) ? 10 : 5;
```

Exemple 2 :

```
System.out.println((genre == " H ") ? " Mr " : " Mme ");
```

Les tableaux

La déclaration des tableaux

- Java permet de placer les crochets **après** ou **avant** le *nom du tableau* dans la déclaration.

Exemple :

```
int tableau[ ] = new int[50]; // déclaration et allocation
OU
int[ ] tableau = new int[50];
OU
int tab[]; // déclaration
tab = new int[50]; //allocation
```

Remarque

  - Le premier élément d'un tableau possède l'indice 0.
- Java ne supporte pas directement les tableaux à plusieurs dimensions :
  - Déclarer un tableau de tableau.

Exemple :

```
float tableau[][] = new float[10][10];
```

- La taille des tableaux de la seconde dimension peut ne pas être identique pour chaque occurrence.

Exemple :

```
int dim1[ ][ ] = new int[3][ ];
dim1[0] = new int[4];
dim1[1] = new int[9];
dim1[2] = new int[2];
```

- Chaque élément du tableau est initialisé selon son type par l'instruction new :
  - 0 pour les numériques,
  - '\0' pour les caractères,
  - false pour les booléens
  - et nil pour les chaines de caractères et les autres objets.

➢ **L'initialisation explicite d'un tableau**

- *Exemples* :

int tableau[5] = {10,20,30,40,50};
int tableau[3][2] = {{5,1},{6,2},{7,3}};

- La taille du tableau n'est pas obligatoire si le tableau est initialisé à sa création.
  - *Exemple* :

int tableau[] = {10,20,30,40,50};

- Le nombre d'éléments de chaque lignes peut ne pas être identique :
  - *Exemple* :

int[][] tabEntiers = {{1,2,3,4,5,6},{1,2,3,4},{1,2,3,4,5,6,7,8,9}};

Les conversions de types

- Lors de la déclaration, il est possible d'utiliser un *cast* :
  Exemple :
  int entier = 5;
  float flottant = **(float)** entier;
  - La conversion peut entraîner une perte d'informations
- Il n'existe pas en Java de fonction pour convertir :
  - Les conversions de type se font par des méthodes.
- La bibliothèque de classes API fournit une série de classes qui contiennent des méthodes de manipulation et de conversion de types élémentaires.
  - **String** pour les chaînes de caractères Unicode
  - **Integer** pour les valeurs entières (integer)
  - **Long** pour les entiers long signés (long)
  - **Float** pour les nombres à virgules flottante (float)
  - **Double** pour les nombres à virgule flottante en double précision (double)
- Les classes portent le même nom que le type élémentaire sur lequel elles reposent avec la **première lettre en majuscule**.
  La conversion d'un entier int en chaîne de caractère String
  Exemple :
  int i = 10;
  String montexte = new String();
  montexte =montexte.valueOf(i);
- *valueOf* est également définie pour des arguments de type *boolean*, *long*, *float*, *double* et *char*
  La conversion d'une chaîne de caractères String en entier int
  Exemple :
  String montexte = new String(" 10 ");
  Integer monnombre=new Integer(montexte);

int i = monnombre.intValue(); //conversion d'Integer en int

La manipulation des chaînes de caractères

- Les chaînes de caractères ne sont pas des tableaux
  - Il faut utiliser les méthodes de la classe *String* d'un objet instancié pour effectuer des manipulations.
- Il est impossible de modifier le contenu d'un objet String construit à partir d'une constante.
- Cependant, il est possible d'utiliser les méthodes qui renvoient une chaîne pour modifier le contenu de la chaîne.
  - *Exemple* :

String texte = " Java Java Java ";
texte = texte.replace('a','o');

Les caractères spéciaux dans les chaînes

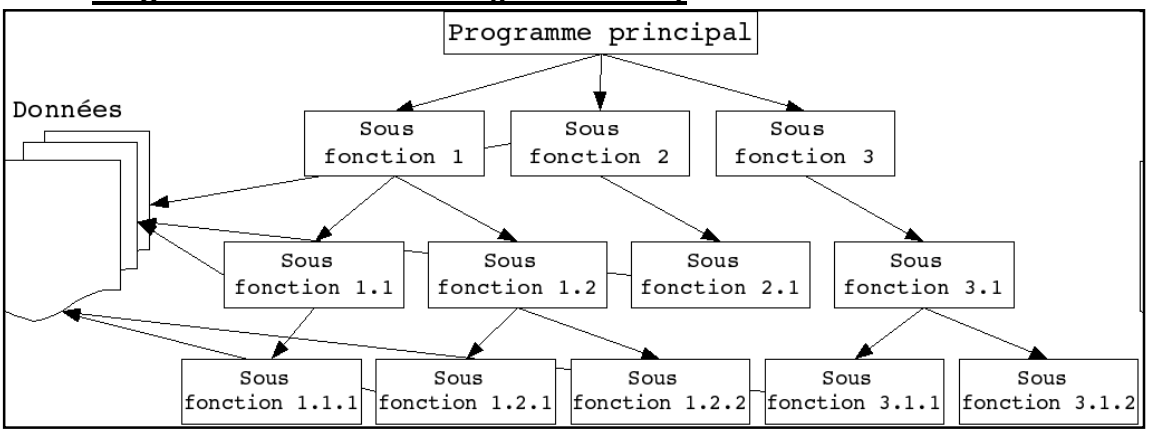
- Dans une chaîne de caractères, plusieurs caractères particuliers doivent être utilisés avec le caractère d'échappement ****.

Caractères spéciaux	Affichage
\'	Apostrophe
\"	Guillemet
\\	anti slash
\t	Tabulation
\b	retour arrière (backspace)
\r	retour chariot
\f	saut de page (form feed)
\n	saut de ligne (newline)
\0ddd	caractère ASCII ddd (octal)
\xdd	caractère ASCII dd (hexadécimal)
\udddd	caractère Unicode dddd (hexadécimal)

Chapitre II
Programmation Orientée Objet Appliquée au langage JAVA

POO Vs Programmation structurée

- **Programmation structurée (procédurale)**



  - Diviser un programme en sous-programmes,
  - S'intéresser aux traitements puis aux données
  - ***Que doit faire le programme ?***
- **Objectifs de la POO**
  - Facilité la réutilisation de code,
    - *encapsulation* et *abstraction*
  - Facilité de l'évolution du code
  - Améliorer la conception et la maintenance des grands systèmes
  - Programmation par « composants ».
  - Conception d'un logiciel à la manière de la fabrication d'une voiture

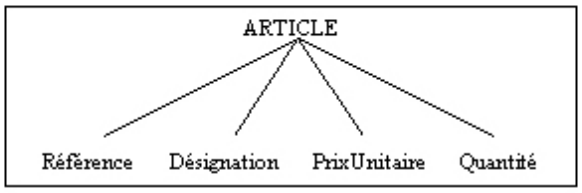
Définition de l'approche orientée objet

- Une méthode de modélisation basée sur une représentation *abstraite* des entités du **monde réel**
- Une méthode qui regroupe les données et les traitements sur ces données au sein d'une entité unique : ***l'objet***.
- Une méthode qui exprime les fonctionnalités sous la forme de collaborations entre les objets

La notion d'objet

- La Programmation Orientée Objet consiste à modéliser informatiquement un ensemble d'éléments d'une partie du monde réel en un ensemble d'entités informatiques.
- Ces entités informatiques sont appelées ***objets***.
- Il s'agit de données informatiques regroupant les principales caractéristiques des éléments du monde réel (taille, couleur, ...).
- Cette modélisation consiste à créer une représentation abstraite, sous forme d'objets, d'entités ayant une *existence matérielle* (chien, voiture, ampoule, ...) ou bien *virtuelle* (sécurité sociale, temps, ...).
- Un objet est caractérisé par plusieurs notions :
  - **Les attributs**: les données caractérisant l'objet.
    - Des variables stockant des informations d'état de l'objet
  - **Les méthodes** (appelées parfois *fonctions membres*)
    - caractérisent son comportement,
    - l'ensemble des actions (appelées *opérations*) que l'objet est à même de réaliser.
    - Ces opérations permettent de faire réagir l'objet aux sollicitations extérieures (ou d'agir sur les autres objets).
    - les opérations sont étroitement liées aux attributs,
    - leurs actions peuvent dépendre des valeurs des attributs, ou bien les modifier
  - **L'identité**:
    - Permet de le distinguer un objet des autres objets, indépendamment de son état.
    - Exemple un produit pourra être repéré par un code,
    - une voiture par un numéro de série,
    - etc.
- Un objet est un ***groupe de données structurées caractérisé par un identifiant unique*** représentant le monde réel.

Exemple : Un article en stock est un objet, il est caractérisé par une référence, une désignation, un prix unitaire et une quantité.



- Un objet est constitué ***attributs*** qui caractérisent la structure de celui-ci.

- *Exemple : Référence, Désignation, PrixUnitaire et Quantité sont les attributs de l'objet de type ARTICLE.*
- L'objet est manipulé par des procédures appelées ***méthodes*** qui sont caractérisées par une entête définissant leur nom, les paramètres d'appel et de retour pour chacune d'elle.
- *Exemple :* On peut définir comme méthodes de l'objet ARTICLE
  - *PrixTtc :* Méthode permettant de calculer le prix TTC d'un article
  - *SortieArticle :* Méthode permettant de diminuer la quantité en stock

Remarque

On ne peut exécuter une méthode sans préciser l'objet sur lequel elle s'applique.

Notion de classe

- Les objets qui ont les mêmes états et les mêmes comportements sont regroupés :
  - c'est une **classe**
- Une classe est un *modèle* de définition pour des objets
  - Ayant même structure (même ensemble d'attributs)
  - Ayant même comportement (même méthodes)
  - Ayant une sémantique commune

- Les **objets** sont des représentations dynamiques, du modèle défini pour eux au travers de la classe (**instanciation**)
  - Une classe permet d'**instancier** (créer) plusieurs objets
  - Chaque objet est ***instance*** d'une et une seule classe

- **Exemple : classe "*Voiture*"**

Programmation Orientée Objet

- Un programme OO est constitué de classes qui permettent de créer des objets qui s'envoient des messages
- L'ensemble des interactions entre les objets défini un algorithme
- Les relations entre les classes reflètent la décomposition du programme

POO avec Java

- Un programme en Java se présente comme un ensemble de classes :
  - une classe par fichier.
  - le fichier doit porter le *même nom* que la classe
    - (sans l'extension Java)
- Faire attention aux majuscules,
  - La première lettre de l'identificateur de la classe est en ***majuscule***,
  - La première lettre des méthodes est en ***minuscule***.

Conventions de noms

- CeciEstUneClasse
- celaEstUneMethode(…)
- jeSuisUneVariable
- JE_SUIS_UNE_CONSTANTE

Affectation et comparaison

- **Affecter un objet**
  - "**a = b**" signifie a devient *identique* à b
  - Les deux objets a et b sont identiques et toute modification de a entraîne celle de b
- **Comparer deux objets**
  - "**a == b** " retourne « *true* » si les deux objets sont identiques
  - C'est-à-dire si les **références** sont les mêmes,
  - cela ne compare pas les attributs

- Le test de comparaison(== et !=) entre objets ne concerne que les références et non les attributs!!!!
- Recopier les attributs d'un objet « **clone()** »
  - Les deux objets a et b sont distincts
  - Toute modification de a n'entraîne pas celle de b
- Comparer le contenu des objets :
  - « **equals(Object o)** »
- Renvoyer « true » si les objets a et b peuvent être considéré s comme *identiques* au vu de leurs attributs

Cycle de vie d'un objet

Création

- Usage d'un ***Constructeur***
- L'objet est créé en mémoire et les attributs de l'objet sont initialisés

Utilisation

- Usage des Méthodes et des Attributs Les attributs de l'objet peuvent être modifiés
- Les attributs (ou leurs dérivés) peuvent être consultés

Remarque **:L'utilisation d'un objet non construit provoque une exception de type *NullPointerException***

Notion de constructeur

- La création d'un nouvel objet est obtenue par l'appel à

new *Constructeur*(paramètres)

- Il existe un constructeur par défaut qui ne possède pas de paramètres
  - si aucun autre constructeur avec paramètre n'existe
- **NB : Les constructeurs portent le même nom que la classe**

Exemple

```
class Point
{
//Attributs
double x;
double y;
//Constructeur sans paramètres
public Point ()
{
x=0;
y=0;
}
//Constructeur avec un seul paramètre
public Point (double a)
```

```
    {
    x=a;
    y=0;
    }
    //Constructeur avec deux paramètres
    public Point(double a, double b)
    {
    x=a;
    y=b;
    }
    //Méthode affiche : pour afficher les coordonnées
d'un point
    void affiche()
    {
    System.out.println("p=("+x+", "+y+")");
    }
    }
    class TestPoint
    {
    //Méthode main: point d'entrée du programme
    public static void main(String []args)
    {
    //Déclaration puis création
    Point p1;//déclaration
    p1=new Point();/*création et allocation mémoire
                              en utilisant le premier
                                      constructeur*/

    //Déclaration et création en une seule ligne
    Point p2=new Point(3.5);
    Point p3 = new Point(4.6, 8D);
    /*Appel de la méthode affiche
    pour afficher les coordonnées des points*/
    p1.affiche();
    p2.affiche();
    p3.affiche();
    }
    }
```

On obtient sur l'écran:

p=(0.0,0.0)
p=(3.5,0.0)
p=(4.6,8.0)

Constructeur : définition

- Un constructeur est une méthode particulière appelée au moment de la *création* pour *initialiser* l'objet en fonction des paramètres fournis par l'utilisateur.
- Règles à respecter:
  - Le nom du constructeur doit correspondre ***exactement*** au nom de la classe.
  - Aucun type de retour déclaré par le constructeur.
- Une classe peut avoir plusieurs constructeurs avec des listes d'arguments différents,
  - La liste des arguments qui détermine quel constructeur doit être utilisé lors de l'appel de l'un des constructeurs.
- Toute classe a au moins un constructeur,
  - Si aucun constructeur n'a été écrit dans la classe un constructeur ***par défaut*** est fourni sans arguments et son corps est vide.
  - Si une classe comporte au moins un constructeur elle perd son constructeur par défaut.

Accès aux attributs

- L'accès à un attribut d'une classe se fait toujours à l'intérieur d'une méthode.
- Si la méthode (sauf pour main) appartient à la même classe que l'attribut l'accès se fait en indiquant le nom de l'attribut directement.
- Si la méthode n'appartient pas à la même classe de l'attribut ou s'il s'agit de la méthode main il faut créer d'abord un objet à partir de la classe de l'attribut et y accéder comme suit:

nom_objet.nom_attribut;

Exemple

```
    class Point
    {
    double x;
    double y;
    void affiche()
    {
    System.out.println("p=("+x+", "+y+")");
//on appelle les attributs directement par leurs noms
    }

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

```
Point p = new Point();
System.out.println("abscisse de p="+p.x);
//accès à l'attribut après création de l'objet
p.affiche();
}
```

On obtient sur l'écran:

abscisse de p= 0.0

P=(0.0,0.0)

Accès à une méthode

- Si l'appel de la méthode se fait à l'intérieur de la classe (dans une autre méthode sauf pour main) on écrit directement:

nom_méthode(liste_paramètres);

- Si l'appel de la méthode est effectué à l'extérieur de la classe ou dans la méthode main, il faut créer un objet de cette classe et appeler la méthode comme suit:

nom_objet.nom_méthode(liste_paramètres);

Encapsulation

- Le regroupement des données et des méthodes dans un même objet ainsi que le masquage de données est appelé: ***encapsulation***.
  - Une classe fournit un certain nombre de services et les utilisateurs de cette classe n'ont pas à connaître la façon dont ces services sont rendus.
- Il faut donc distinguer dans la description de la classe deux parties :
  - La partie publique, accessible par les autres classes,
  - La partie privée, accessible uniquement par les méthodes de la classe.
- Il est vivement recommandé de mettre les attributs d'une classe dans la partie privée, ainsi protégés, les membres privés d'une classe ne sont pas directement modifiables.
- **Les niveaux d'accès**
- Java définit 3 niveaux d'accès pour les variables d'instances (données membres) et les méthodes :
- **public** : un élément *public* est accessible de partout et sans aucune restriction.
  - Certaines classes (comme la classe principale *main*) doivent obligatoirement être déclarées publiques (pour pouvoir exécuter l'application...)
- **protected** : un élément *protected* (*protégé*) est accessible uniquement aux classes d'un package et à ses classes filles
- **private** : un élément *private* (*privé*) est accessible uniquement au sein de la classe dans laquelle il est déclaré.

- Ces éléments ne peuvent être manipulés qu'à l'aide de méthodes spécifiques appelés ***accesseur*** et ***mutateur***

La notion d'accesseur

- Un accesseur est une méthode permettant de ***récupérer*** le contenu d'une donnée membre protégée.
- Un accesseur, pour accomplir sa fonction :
  - doit avoir comme type de retour le type de la variable à renvoyer
  - ne doit pas nécessairement posséder d'arguments
- Une convention de nommer veut que l'on fasse commencer de façon préférentielle le nom de l'accesseur par le préfixe ***get***.

Exemple

```
class Point
{
private double x;
private double y;
….
public double getX()
{
return x;
}
```

La notion de mutateur

- Un mutateur (modificateur) est une méthode permettant de ***modifier*** le contenu d'une donnée membre protégée.
- Un mutateur, pour accomplir sa fonction :
  - doit avoir comme paramètre la valeur à assigner à la donnée membre. Le paramètre doit donc être du type de la donnée membre
  - ne doit pas nécessairement renvoyer de valeur (il possède dans sa plus simple expression le type *void*)
- Une convention de nommer ve ut que l'on fasse commencer de façon préférentielle le nom du mutateur par le préfixe ***set***.

Exemple

```
class Point {
private double x;
private double y;
….
public void setX(double a)
{  x=a;  }
```

Notion de destructeur

- La destruction des objets se fait de manière *implicite*

- Le *ramasse-miettes* ou *Garbage Collector* se met en route
  - Automatiquement
  - Si plus aucune variable ne référence l'objet
  - Si le bloc dans lequel il est défini se termine
  - Si l'objet a été affecté à « null »
  - Manuellement :
    - Sur demande explicite par l'instruction « **System.gc()** »

```
class Point {
private double x;
private double y;
public Point(double x, double y) {
this.x = x;
this.y = y;
}
public void translater(double dx, double dy) {
x += dx;
y += dy;
}
public String toString() {
return "Point[x:" + x + ", y:" + y + "]";
}
public void finalize() {
System.out.println("finalisation de " + this);
}
}
```

```
Class TestPoint
```

```
{
Public static void main(String []args)
{
Point p1 = new Point(14,14);
Point p2 = new Point(10,10);
System.out.pritnln(p1);
System.out.pritnln(p2);
p1.translater(10,10);
p1 = null;
System.gc();//Appel explicite au garbage collector
System.out.pritnln(p1);
System.out.pritnln(p2);}}
```

```
Point[x:14.0, y:14.0]
Point[x:10.0, y:10.0]
finalisation de Point[x:24.0, y:24.0]
null
Point[x:10.0, y:10.0]
```

Chapitre III
L'Héritage

Définition et intérêts

- **Héritage**

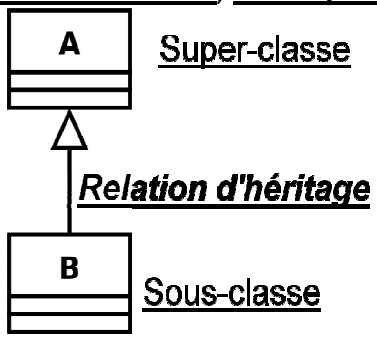
Technique offerte par les langages de programmation pour construire une classe à partir d'une (ou plusieurs) autre classe (déjà existante) en partageant ses attributs et opérations.

- **Intérêts**

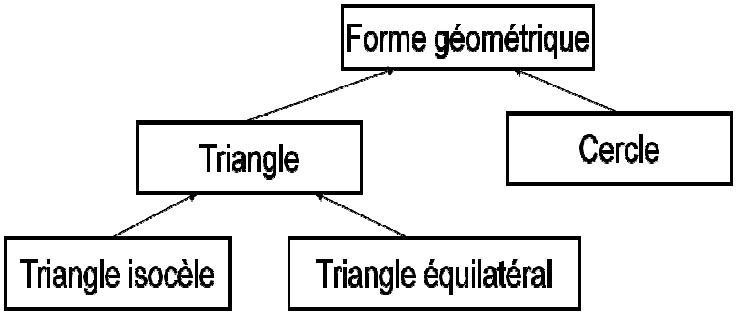
- **Spécialisation, enrichissement** :
  - une nouvelle classe réutilise les attributs et les opérations d'une classe en y ***ajoutant*** et/ou des opérations particulières à la nouvelle classe de la classe à hériter
- **Redéfinition** :
  - une nouvelle classe redéfinit les attributs et les opérations d'une classe de manière à en ***changer*** le sens et/ou le comportement pour le cas particulier défini par la nouvelle classe
- **Réutilisation** :
  - évite de réécrire du code existant et parfois on ne possède pas les sources

Vocabulaire

- La classe **B** qui hérite de la classe **A** s'appelle ***une classe fille*** ou ***sous-classe***
- La classe **A** s'appelle ***une classe mère***, ***classe parente*** ou ***super-classe***

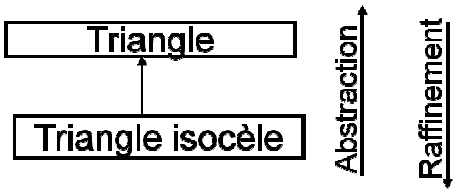


Exemple

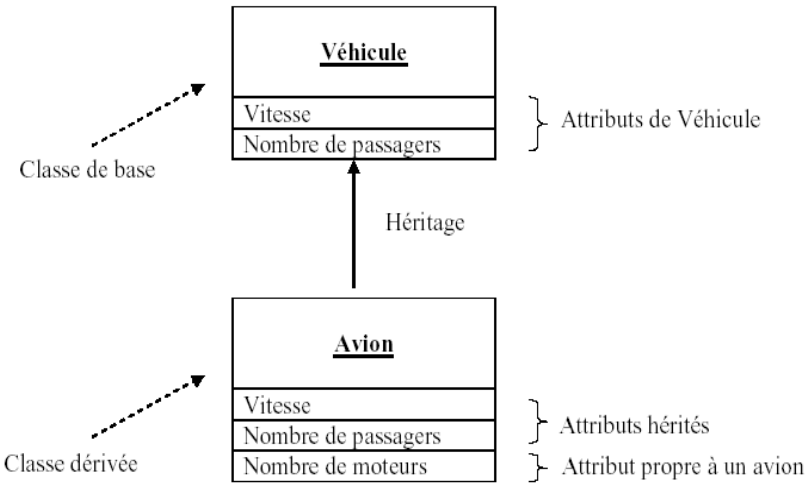


Généralisation et Spécialisation

- La généralisation exprime une relation "**est-un**" entre une classe et sa super-classe.
  - Le triangle isocèle **est un** triangle
- La sous classe est plus spécifique que la classe mère qui est plus générale que la classe fille
- L'héritage permet
  - de **généraliser** dans le sens *abstraction*
  - de **spécialiser** dans le sens *raffinement*



Exemple



Remarques

- Les attributs de la classe dérivée comprennent :
  - les attributs *hérités* de la classe de base
  - les attributs *propres* à la classe dérivée.
- La classe dérivée peut définir des attributs portant le même nom que ceux de sa classe mère.
- La classe dérivée peut changer l'implémentation d'une ou plusieurs méthodes héritées.
- Pour accéder aux méthodes ou aux attributs redéfinis de sa classe mère il faut utiliser le mot clé **super.**

Héritage et Java

- Héritage *simple*
  - Une classe ne peut hériter que **d'une seule** autre classe
    - Dans certains autres langages (ex : C++) possibilité d'héritage multiple
- Utilisation du mot-clé **extends** après le nom de la classe.
- Héritage à plusieurs niveaux

Syntaxe

```
class Derivee extends Classe_de_base
{
// définitions d'attributs
// définitions de méthodes
 }
```

Exemple d'héritage en Java

```
class Livre
{
        private int nbpages;
        private String titre;

   public Livre(int i)
        {  nbpages=i;}

        public Livre(int i, String S)
        {
        nbpages=i;
        titre=S;
    }
   public int getNbpages()
        {
                return nbpages;     }  }
```

```
class Dictionnaire extends Livre
{
private int nbDef=15000;
private String type="FR/A";
public int getNbdefparpage()
{
        return nbdef/geNbpages();
}

}
```

- Les attributs privés d'une classe ne sont pas accessibles même par la classe fille :
- Le modificateur ***protected*** autorise l'héritage

```
class Livre
{
   protected int nbpages;
   private String titre;

  public Livre(int i)
   {  nbpages=i;}

   public Livre(int i, String S)
   {
   nbpages=i;
   titre=S;
   }
  public int getNbpages()
   {
        return nbpages;
   }  }
```

- La méthode getNbdefparpages() de la classe Dictionnaire devient comme suit :

```
class Dictionnaire extends Livre
{
private int nbDef=15000;
private String type="FR/A";
public int getNbdefparpage()
{
        return nbdef/nbpages;
}}
```

Usage des constructeurs

- La classe dérivée doit prendre en charge la construction de la classe de base.
  - Pour construire un Dictionnaire, il faut construire d'abord un Livre;
- Le constructeur de la classe de base est donc appelé **avant** le constructeur de la classe dérivée
- Si un constructeur de la classe dérivée appelle explicitement un constructeur de la classe de base, cet appel doit être obligatoirement la première instruction de constructeur.
  Il doit utiliser pour cela, le mot clé ***super***.
  Exemple

```
class Dictionnaire extends Livre
public Dictionnaire (int i, String t, int D)
{
 super(i,t);
        nbDef=D;
}
```

Remarques

- Lors de la construction de la classe dérivée, on doit tenir compte de la présence du constructeur dans la classe de base et/ou dans la classe dérivée.
- La classe de base et la classe dérivée ont au moins un constructeur public, c'est le cas général. De ce fait, le constructeur de la classe dérivée doit appeler le constructeur de la classe de base disponible.
- Si la classe de base n'a aucun constructeur, la classe dérivée peut ne pas appeler explicitement le constructeur de la classe de base.
  - Si elle le fait, elle ne peut appeler que le constructeur par défaut, vu que c'est le seul qui est disponible dans ce cas dans la classe de base.
- Si la classe dérivée ne possède aucun constructeur, dans ce cas, la classe de base doit avoir un constructeur public sans argument (par défaut ou un explicite).

Surcharge et redéfinition

- Une sous-classe peut *ajouter* des nouveaux attributs et/ou méthodes à ceux qu'elle hérite
  - ***surcharge*** en fait partie
- Une sous-classe peut *redéfinir* (***redéfinition***) les méthodes à ceux dont elle hérite et fournir des implémentations spécifiques pour celles-ci.

- **Surcharge** : possibilité de définir des méthodes possédant le même nom mais dont les arguments (paramètres et valeur de retour) *différent*
- **Redéfinition** (overriding) : lorsque la sous-classe définit une méthode dont le nom, les paramètres et le type de retour sont *identiques*

- Ne pas confondre surcharge et redéfinition.
  - Dans le cas de la surcharge la sous-classe ajoute des méthodes
  - tandis que la redéfinition « spécialise » des méthodes existantes

Exemple de redéfinition

```
class FormeGeometrique
{
        private String nom;
        private String couleur;
        public void afficherSurface()
        {
                System.out.println("je ne paux pas calculer la surface");
        }
        public FormeGeometrique(String n, String c)
        {
                nom=n;
                couleur=c;
        } }
class Rectangle extends FormeGeometrique
{
        private int largeur;
        private int longueur;
        public Rectangle (String n, string c, int la, int lo )()
        {
                super(n,c);
                largeur=la;
                longueur=lo;
        }
        // Redéfinition de la méthode afficherSurface
        public void afficherSurface(()
                {
                System.out.println("la surface est " + longueur*largeur) ;
                } }
```

- L'appel de la méthode de la classe mère se fait par :
  - super.afficherSurface()
- Remarques

- Une classe peut protéger une méthode afin d'éviter qu'elle ne soit redéfinie dans ses sous-classes
    - On ajoute tout simplement le mot clé **final**
- **final** class FormeGeometrique protège les droits d'accès à cette classe

La classe Object

- La classe **Object** est la classe de plus haut niveau dans la hiérarchie d'héritage
    - Toute classe autre que **Object** possède une super-classe
    - Toute classe hérite directement ou indirectement de la classe **Object**
    - Une classe qui ne définit pas de clause **extends** hérite de la classe
- Étant donné que toute les classes héritent de la classe *Object*, tout objet a accès aux méthode de la classe Object.
    - **Méthode toString():** public String toString() permet d'obtenir une présentation de l'objet sous forme de chaîne de caractères.
    - **Méthode equals():**public boolean equals (Object a) permet de comparer deux objets, retourne true si identiques et false sinon.
    - **Méthode clone() :** protected Object clone() permet d'obtenir une copie de l'objet.

Chapitre IV
Les classes abstraites et les interfaces

Introduction

- En général, l'héritage suit la règle suivante :
  - Les sous-classes sont **plus concrètes** et **spécifiques** que les classes dont elles dérivent qui sont **plus générales** et **abstraites**.
- En POO, on conçoit ainsi souvent des classes dont la seule raison d'être est d'agir comme un **ensemble commun de méthodes et de variables** pour des sous-classes.
  - Par exemple, une classe *Vehicule* ne servirait que comme dénominateur commun entre les classes *Voiture* et *Velo*.

Les classes abstraites en JAVA

- Il existe en Java un modificateur spécial pour ce concept : **abstract**.
- Une classe déclarée ***abstraite*** **(abstract)** est une classe normale, mais il est interdit d'en créer des instances (des objets).
  - Elles servent donc de **modèle** à des sous classes.
- Une méthode abstraite ne nécessite pas d'implémentation.
- Ce sont les sous-classes qui doivent fournir l'implémentation en redéfinissant cette méthode.

➢ C'est une obligation : une sous-classe d'une classe abstraite **doit redéfinir** toutes les **méthodes abstraites** de la classe de base.

Définition d'une classe abstraite

- Une classe abstraite est une classe incomplète.
- Elle regroupe un ensemble de variables et de méthodes mais certaines de ses méthodes ne contiennent pas d'instructions,
  - Elles devront être définies dans une classe héritant de cette classe abstraite.

Utilité des classes abstraites

- Une classe abstraite sert à définir les grandes lignes du comportement d'une classe d'objets sans forcer l'implémentation des détails de l'algorithme.
- Prenons l'exemple d'une chaîne de montage automobile, de laquelle sort un modèle de voiture particulier.

- On peut choisir de faire une nouvelle chaîne de montage pour le modèle à pare-choc métallisé, une pour le modèle à pare-choc en plastique, etc.
- Ou on peut décider de faire une chaîne de montage générique, de laquelle sortiront des véhicules non finis, que l'on terminera sur d'autres petites chaînes.

Règles d'utilisation des classes abstraites

- ✓ ***Règle 1*** **:** Une classe est automatiquement abstraite si une de ses méthodes est abstraite.
- ✓ ***Règle 2*** **:** On déclare qu'une classe est abstraite par le mot clef **abstract**.

abstract NomClasse {Attributs et Méthodes}

- ✓ ***Règle 3*** **:** Une classe abstraite n'est pas instanciable
  - On ne peut pas utiliser les constructeurs d'une classe abstraite et donc on ne peut pas créer d'objet de cette classe.
- ✓ ***Règle 4*** **:** Une classe qui hérite d'une classe abstraite ne devient concrète que si elle implémente toutes les méthodes abstraites de la classe dont elle hérite.
- ✓ ***Règle 5*** **:** Une méthode abstraite ne contient pas de corps, mais doit être implémentée dans les sous-classes non abstraites:

abstract nomDeMéthode (arguments)**;**

- ✓ ***Règle 6*** **:** Une classe abstraite peut contenir des méthodes non abstraites et des déclarations de variables ordinaires.
- ✓ ***Règle 7*** **:** Une classe abstraite peut être dérivée en une sous-classe :
  - Abstraite si une ou plusieurs méthodes ne sont pas implémentées par la classe fille,
  - Non abstraite si toutes les méthodes abstraites sont implémentées dans la classe fille.

Activité

1. Il s'agit de définir une classe nommée **Forme** comportant

- deux méthodes abstraites :
  - **float perimetre()** et **float surface()**
- Et une méthode non abstraite **void contenantCarre(float surf)** qui affiche
  - "Cette forme peut contenir un carré de surface surf "
  - "Cette forme peut contenir un carré de surface surf"

2. Écrire ensuite les deux classes **Rectangle** et Cercle héritant de cette classe Forme et définissant les deux méthodes abstraites de celle-ci.

- La classe Rectangle a 2 attributs entiers privés
  - longueur et largeur
- La classe Cercle a un attribut privé de type float nommé rayon

Application

- Tester la méthode **contenantCarre** de la classe **Forme** avec un **surf = 20** (dans la méthode main) en l'appliquant pour un rectangle de **longueur = 2** et de **largeur =** 1
- Puis la tester avec **surf = 1** pour un cercle de **rayon=1**.

Les interfaces

- Une interface en Java est comme une classe abstraite ***sans aucune définition de méthode.***
- L'interface décrit les méthodes et les types de leurs arguments, sans rien dire sur leur implémentation
- les attributs sont public static final
  - Constantes
- les méthodes sont toutes abstraites

public abstract

A quoi sert une interface ?

- Elles sont utiles pour des équipes qui travaillent sur un même projet
  - Chacune doit pouvoir connaître les méthodes qui seront implémentées par les autres et leur spécification pour pouvoir inclure les appels dans leurs propres méthodes.

➢ Les interfaces sont des groupes de méthodes que l'on peut réutiliser où bon nous semble.

- Une seule interface par fichier
- Une interface est différente d'une classe,
  - On ne peut pas instancier une interface;
  - On ***implémente*** une interface.
- Une interface ne comporte que des constantes et des méthodes abstraites.
  - Elle n'est pas "classe abstraite"
- Une classe peut implémenter une ou plusieurs interfaces, c'est à dire définir le corps de toutes les méthodes abstraites.
  - **C'est l'héritage multiple de Java.**

Syntaxe

Déclaration

```
interface NomInterface
{
// attributs (par défaut public, static, final)
// méthodes (par défaut abstract, public)
}
```

Utilisation

```
class LaClasse implements NomInterface1, ...
{
// Définition de toutes les méthodes abstraites
}
```

Activité

- Ecrire une *interface* nommée **Homme** possédant une seule méthode void identite() dont le rôle est d'afficher les informations concernant un homme.
- Ecrire une classe **Personne** implémentant cette interface et possédant deux attributs privés nom et prenom de type String et un constructeur paramétré avec le nom et le prénom d'une personne.
- Ecrire une classe **Client** héritant de la classe Personne et implémentant l'interface Homme avec un attribut privé supplémentaire nommé numero de type entier et un constructeur paramétré en conséquence.

Chapitre V
Le transtypage & le polymorphisme

Définition du transtypage

- Le transtypage
  - conversion de type
  - *cast* en anglais
- ➢ consiste à modifier le type d'une variable ou d'une expression.
- Par exemple, il est possible transtyper un *int* en *double*.
  - **Transtypage des types primitifs**
- Il existe deux types de transtypages:
  - le **transtypage implicite**
  - le **transtypage explicite**.

Transtypage implicite

- **Le transtypage implicite est traité automatiquement par le compilateur**
  - lors d'une affectation
  - Ou du passage d'un paramètre effectif.
- Un transtypage peut être implicite **si le type cible a un plus grand domaine que le type d'origine (gain de précision)**
- Par exemple, il est possible d'affecter directement un *int* à un *double*, ou un *byte* en *short*, sans expliciter de transtypage
- Le compilateur effectue automatiquement la conversion.

Transtypage explicite

- Pour réaliser un cast explicite, ***on met entre parenthèses le nom du type dans lequel on veut convertir suivi du nom de la variable*** (ou de l'expression entre parenthèses) qu'on veut transtyper.

Exemple

```
double d;
 int i;
i = (int) d;
//transtypage d'un double en int pour affectation
```

- En revanche lorsqu'on veut **convertir une variable ou une expression dans un type qui lui fait perdre de la précision (domaine de valeurs plus restreint)**, il faut réaliser un **transtypage explicite**
  - sinon erreur du compilateur.
- Par exemple :
  - convertir un *float* en *int* fait perdre les chiffres après la virgule,

- convertir un long en short donne des résultats incohérents pour les grands entiers (qui ne peuvent pas être représentés avec un short)
- Dans ces cas, il faut donc réaliser un ***cast explicite***.

Le transtypage de références d'objets

- Il est possible de convertir un objet d'une classe en un objet d'une autre classe ***si les classes ont un lien d'héritage***
- Reamrque :ce sont les références aux objets et non les objets eux-mêmes qui sont transtypés).
- Le transtypage d'un objet dans le sens **fille-mère** est implicite.
- En revanche, le transtypage dans le sens **mère-fille** doit être explicite et n'est pas toujours possible.

Exemple

```
class Ville
{
private String nom; /*le nom ne sera accessible que par la classe Ville, et pas par la classe Capitale*/
protected int nbHab; /*le nombre d'habitants sera accessible par la classe Capitale*/
public Ville (String leNom, int leNbHab)
{
nom = leNom.toUpperCase( );
if (leNbHab< 0)
{
System.out.println("Un nombre d'habitant doit être positif.");
nbHab = -1;
}
else
nbHab = leNbHab;
}
public String getNom()
{return nom;}
public int getNbHab( )
{return nbHab;}
public void setNbHab(int nvNbHab)
{
if (nvnbHab < 0)
System.out.println("Un nombre d'habitant doit être positif. La modification n'a pas été prise en compte");
else
```

```
nbHab = nvNbHab; }
public String presenteToi()
{
String presente = "Ville "+ nom +" nombre d'habitants ";
if (nbHab = = -1)
presente = presente + "inconnu";
else
presente = presente + " = " + nbHab;
return presente;
} }
```

Classe *Capiltale*

```
class Capitale extends Ville
{
private String pays;
//constructeur
public Capitale (String leNom, String lePays, int leNbHab)
{
super(leNom, leNbHab);
pays = lePays;
}
//accesseurs supplémentaires
public String getPays( )
{
return pays;
}
public void setPays(String nomPays)
{
pays = nomPays;
}
//méthode presenteToi( ) redéfinie
public String presenteToi( )
{
String presente = super.presenteToi( );
presente = presente + " Capitale de "+ pays;
return presente;
}  }
```

Transtypage implicite (sens fille→mère)

- Il est toujours possible d'utiliser une référence de la classe mère pour désigner un objet d'une classe dérivée (fille, petite-fille et toute la descendance).
- Il y a alors transtypage implicite de la classe fille vers la classe mère.
- Cela est logique si on se dit qu'un objet dérivé **EST** un objet de base.

Exemple

- Une Capitale **EST** une Ville, donc on peut utiliser une référence de type Ville pour désigner une Capitale
- Il y a transtypage implicite d'une Capitale en Ville.
- Par contre, une Ville n'est pas forcément un Capitale : donc on ne peut pas transtyper une Ville en Capitale, autrement utiliser une référence de Capitale pour désigner une Ville.
- Remarquez que contrairement au transtypage des types primitifs, le transtypage implicite de références entraîne une perte de précision
  - le type Ville étant moins précis que le type Capitale
- *Exemple d'utilisation du transtypage implicite :*

```
Ville v;
Capitale c = new Capitale("Paris", "France",10000000);
v = c;
//transtypage implicite d'une Ville en Capitale
```

- L'instruction d'affectation **v = c** implique un transtypage implicite.
  - La référence v du type Ville est alors utilisée pour désigner un objet de type Capitale.
  - L'objet désigné par c, qui est du type Capitale est donc transtypé en Ville.
- Donc on ne pourra pas utiliser v pour appeler une méthode spécifique de la classe Capitale (car v est une référence sur une Ville, qui ne peut pas appeler des méthodes de la classe Capitale) .
- Il faudrait pour cela effectuer un transtypage explicite de v en Capitale.

Transtypage explicite (sens mère→fille)

- Le transtypage explicite des références est utilisé pour convertir le type d'une référence dans un type dérivé.
- C'est logique si l'on se dit qu'un objet d'une classe mère n'est pas un objet de ses classes filles
  - Une Ville n'est pas une Capitale
- Par exemple, si l'on voulait utiliser la référence v (qui est du type Ville mais qui désigne une Capitale) pour invoquer une méthode spécifique de la classe Capitale, il faudrait réaliser un transtypage de v en Capitale

- Sinon, le compilateur refuserait : le transtypage dans le sens mère fille n'étant pas implicite.

Exemple

```
String lePays;
lePays = (Capitale)v.getPays( );
```

- Si on avait écrit **lePays = v.getPays( )** , le compilateur aurait généré une erreur car *getPays( )* n'est pas une méthode de la classe Ville (donc faire partie v).
- Un transtypage explicite n'est pas toujours possible, même si les classes ont un lien de parenté.
- C'est au programmeur de vérifier que son transtypage n'engendrera pas d'erreur, c'est-à-dire que l'objet est bien effectivement du type dans lequel on transtype la référence.
- v peut être transtypé en Capitale car v désigne effectivement un objet de type Capitale.
- Exemple de transtypage explicite impossible :
  - Créer un objet Ville à partir d'une référence de type Capitale
  - Dans ce cas, c ne peut pas désigner une capitale (il n'y a pas de pays)
- ~~Capitale c = new Ville("Bruxelles",15000000);~~

Le polymorphisme

- Une méthode ***polymorphe*** est une méthode déclarée dans une super-classe et redéfinie par une sous classe.
- Tout objet peut être vu comme instance de sa classe et aussi comme instance de toutes les classes dérivées de sa classe.
- Et si on voulait invoquer la méthode *presenteToi( )* avec la référence v ?
  - System.out.println(v.presenteToi( ));
- Quelle implémentation de *presenteToi( )* s'exécuterait alors?
- Celle de Ville … qui afficherait :
  - Ville Paris nombre d'habitants 10000000
- ou celle de Capitale ? qui afficherait
  - Ville Paris nombre d'habitants 10000000 Capitale de France

Chapitre VI
Les packages & les règles de visibilité

Les packages

- Un **package** ou un **paquetage** est un ensemble de classes et d'interfaces travaillant sur un même domaine.
- Un package est donc un groupe de classes associées à une fonctionnalité
- Exemples de packages
  - java.lang : rassemble les classes de base Java (Object, String, …)
  - java.util : rassemble les classes utilitaires (Collections, Date, …)
  - java.io : lecture et écriture
- Un ensemble de packages sont fournis avec Java, les packages spécifiques sont à la charge des programmeurs.
- Les packages sont organisés en hiérarchie
- On peut définir des sous packages, des sous sous packages, ….
- D'où : un package peut contenir
  - Des classes ou des interfaces
  - Un autre package (sous-package)
- Le nom d'un package décrit le nom du répertoire dans lequel se trouvent les classes de ce package.
- Les classes de java.lang se trouvent dans le sous répertoire ../java/lang
- Si aucun package n'est déclaré, la classe appartient à un paquetage appelé **paquetage par défaut**.
- Il est constitué de l'ensemble des classes appartenant au répertoire courant.
- Pour utiliser une classe appartenant à un package il y a deux possibilités:
  - On utilise le nom du package suivi du nom de la classe (les deux noms séparés par un point)
    - geo.Geometrie
  - On utilise l'instruction import (en début du fichier)
    - import geo.Geometrie;
- On peut généralement utiliser les 2 méthodes sauf dans le cas où deux classes de même nom sont définies dans deux packages différents.
  - On utilise la première méthode.
- Java importe automatiquement le package java.lang qui permet d'utiliser des classes comme System.

- Les directives import sont placées après la directive package (s'il y en a une) et avant les définitions de classes. On peut utiliser le caractère * pour dire qu'on veut importer toutes les classes et interfaces d'un package :
  - import geo.*;
  - Structure des classes et des packages
- Seules les classes publics sont accessibles à partir d'un autre package.
- Il ne peut y avoir qu'une classe publique par fichier qui doit porter le même nom que le fichier avec une extension .java (en respectant les majuscules et les minuscules).
- Les fichiers des classes qui font partie d'un même package doivent :
  - Etre placées dans une hiérarchie de répertoires correspondant à celle des packages.
  - Commencer par une déclaration précisant le nom de leur package précédé du mot clé package.
- Pour localiser les packages et donc les classes présentes sur disque, le compilateur et la java machine utilisent la variable CLASSPATH.
- CLASSPATH : permet de référencer tous les répertoires servant de racine à une arborescence de packages.
- Les classes se trouvant dans un même fichier appartiennent au même package.

Intérêts des packages

- Faciliter la recherche de l'emplacement physique des classes pour la compilation et l'exécution.
- Eliminer la confusion entre les classes de même nom.
- Structurer l'ensemble des classes selon une arborescence.
- Permettre de nuancer des niveaux de visibilité entre les classes selon qu'elles appartiennent ou non au même package.

Règles de visibilité des classes

- Une classe est soit publique, soit possède la visibilité par défaut.
- Une classe publique définie par le mot clé public est visible en dehors du package dans laquelle elle est définie.
- Par défaut, une classe n'est pas visible en dehors de son package.
- Dans un fichier Java, il ne peut y avoir qu'une seule classe publique.
- Si deux classes publiques sont définies dans un seul fichier il y a une erreur de compilation.

Accessibles aux	Méthodes et attributs de la même classe	Même package		Autres packages	
		Classes	Classes dérivées	Classes	Classes dérivées
Public	•	•	•	•	•
Protected	•	•	•	•	
Par défaut	•	•	•		
private	•				

Exercice d'application

- On dispose de deux fichiers sources C1.java et C2.java présentés ci-dessous.
- Indiquer quel doit être leur emplacement physique et trouver les instructions erronées dans ces deux fichiers tout en justifiant la réponse.