CNAM – Aix en Provence

Valeur C - RSM

Conception d'Applications Multimedia

Programmation Multimedia

Session 2

Erwan TRANVOUEZ
Maître de Conférences

erwan.tranvouez@polytech.univ-mrs.fr
http://erwan.tranvouez.free.fr

Objectifs du cours:

- (Rappel sur) la conception orientée objet (avec Java)
- Programmation en Java

Plan de la session

1. Java en général

Quoi, Pourquoi, Comment, …

Pourquoi Java et le Multimédia ? 5

Langage indépendant de l'architecture logicielle et matérielle

- **1 code pour plusieurs cibles :**
  - **PC (Windows, Linux)**
  - **Unix**
  - **Mac …**
- **Permet d'appréhender les enjeux techniques généraux utilisable ailleurs…**

Le Java fluent :

- **J2SE : Java 2 Standard Edition**
- **J2EE : Java 2 Enterprise Edition** (J2SE + API pour professionnels)
- **J2ME : Java 2 Micro Edition** (applications embarqués pour « appareils mobiles »)
- **Site web : http://java.sun.com**

Qu'est ce que Java ? 6

Java est un langage :

- **Orienté Objet :** tout est objet.
- **Portable :** le code compilé peut être exécuté sur toute machine informatique possédant une machine virtuelle java.
- **Orienté Réseau :** bibliothèques de fonctionnalités réseau, applications dédiées (applets/servlets,...).
- **Multi-Tâches :** Threads.
- **Sûr :** Typage fort, pas de gestion de pointeur,…
- **Sécurisé :** protection réseau, ...
- **Avec une syntaxe proche du C/C++.**

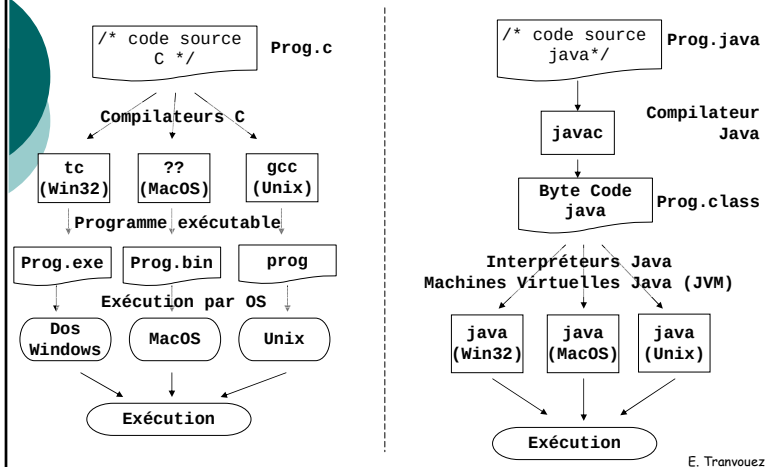
Qu'est ce que Java ? Historique 7

- **1991 :** Green project : électronique intelligente … Le langage Oak est lancé. Se propose de compenser les difficultés ou sources d'erreurs du C++.
- **1993 :** … renommé ensuite Java. Débouchés pas encore murs pour l'électronique mais boom de l'Internet (page dynamique, contenu multimédia, … => Applet).
- **1995 : vers. 1.0 :**.API de base avec AWT, un peu d'audio.
- **1997 : vers. 1.1 :** JFC (Swing, JDBC, JNI, …) => applications
- **1998-1999 : vers. 1.2, 1.3 :** JFC incluses, compilateur JIT, Java 2D, Java Sound Engine,
- **2002: vers. 1.4 :** Java 2D, Java Sound Engine
- **2005 : vers 1.5.** Optimisation (application et graphique), évolution langage …

Inconvénient 8

- **Requiert une certaine maîtrise technique :** assistance à la programmation et prototypage important mais non transparent.
- **Performance :** compilation puis interprétation du byte code => plus lent qu'exécution directe … même si ce critère s'est amélioré (optimisation machine virtuelle ex. Mobile)
- **Aspect visuel :** bibliothèques crées à partir de 0 : aspect visuel parfois moins « jolies » quoiqu'en évolution (ex. AWT, Swing, SWT -> Eclipse)

Principe du **"Compile Once, Run Everywhere"** Comparaison programmation en C et en Java



Premier Programme Java : Hello World

- Un **programme Java** c'est une **classe Java** contenant une **fonction main()**
- **Ex:**

HelloWorld.java

Déclaration de la classe HelloWorld

Fonction principale

Fonction d'affichage d'une chaîne de caractères

```
public class HelloWord {

  public static void main(String args[])
  {
    System.out.println("Hello World !!\n");
  }

}
```


Compiler et exécuter un programme Java

- Compilation / exécution : utilisation de javac / java
- **Ex:**

Sous DOS :

```
C:\>javac HelloWorld.java

C:\>java HelloWorld
Hello World
C:\>
```

Sous Unix/Linux

```
~/javac HelloWorld.java

~/ java HelloWorld
Hello World
~/
```

Le code généré sous l'un ou l'autre OS marchera sous les deux…

Dans les deux cas, le chemin d'accès des programmes javac et java (ex : `c:\dev\jdk1.2\bin`) doit être défini dans la variable d'environnement PATH (cf. transparents suivant).

Notion de paquetage

- **Traduction de *package* :** définit une librairie de classes. Correspond à un regroupement de plusieurs classes dans un même répertoire portant le même nom que celui du package.
- **Utilisation similaire au** #include **du C/C++ : avec le mot clé import :**

```
// Toto.java
import java.awt.*;
import java.util.Date;
public class Toto
{
 public static void main(String args[]) {
   Date d = new Date();
   System.out.println("Aujourd'hui :" +d);
 }
}
```

Toto peut faire appel à toutes les classes publiques de `java/awt/`

Toto peut faire appel à la classe Date accessible dans `java/util/`

Les nombreuses classes de base de java sont stockées dans un fichier compressé `".jar"` **(fichier "zippé").**

Quelque paquetage généraux importants

Nom Paquetage	Description	Exemple de classes
java.io	Gestion des entrées-sorties (clavier, fichier, flux, …)	System avec les membres publiques et statiques in et out, IOException, InputStream, File
java.lang	Classes de bases utilisée pour définir langage de programmation Java	String, classes Wrapper (Boolean, Integer, …), Exception, Thread, Object
java.util	Diverses classes utilitaires : tableau dynamique et autres, , modèle d'évenement, accès à l'horloge (date, heure,…), calcul aléatoire…	Hashtable, Vector, EventListener, Random, Date
java.net	Classes dédiées aux applications réseau.	Socket/ServerSocket, URL, DatagramPacket
java.applet	Création d'applets (i.e. application cliente dans un navigateur internet)	Applet
java.awt	Classes de bases pour créer des interfaces graphiques minimales.	Button, Checkbox, Dialog, Image, Menu
javax.swing	Composants d'interfaces graphiques (plus général et puissant que AWT mais … plus lourd)	Jbutton, JcheckBox, JfileChooser, Jmenu, LookAndFeel

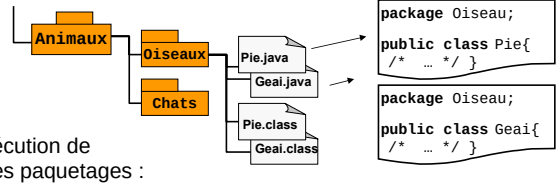
Quelque paquetage multimédia importants

- **Java 2D :** primitives de dessin en 2 dimension (image, animation). **Natif.**
- **Java 3D :** idem en 3D. **Module.**
- **Java Media Framework :** lecture de média synchronisé (Module).
- **Java Sound :** lecture de son et synthétiseur MIDI **(Natif)**
- **Java Speech :** reconnaissance vocale et synthèse vocale (avec IBM) (module).
- **Java Telephony : …Module**
- **Plus des projets de TV, …**

Gestion de paquetage

- Déclaration d'appartenance à un paquetage avec le **mot clé package**

Exemple : Moineau.java et Pigeon.java (ainsi que les fichiers binaires) doivent être dans un répertoire appelé Oiseau.




```
package Oiseau;

public class Pie{
 /*  … */ }
```

```
package Oiseau;

public class Geai{
 /*  … */ }
```

- Compilation/exécution de classes dans des paquetages :

```
C:\dev\Animaux>javac Oiseaux/Pie.java
C:\dev\Animaux>java Oiseaux.Geai
C:\dev\>java Animaux.Oiseaux.Geai
C:\>java –classpath c:\dev\Animaux Oiseaux.Geai
```

Les '.' sont interprétés comme des niveaux de répertoire (comme '\' sous Dos et '/' sous Unix)

On peut aussi définir une variable d'environnement CLASSPATH (ex : `set CLASSPATH=\dev`) contenant les chemins des paquetages que l'on veut rendre accessible ou passer cette information en paramètre de java ou javac (option –classpath).

Règles de programmation

- **1 fichier Java (Xxx.java) peut contenir**
  - **1 seule** classe **publique** (le fichier doit porter le même nom que la classe)
  - **N** classes '**package**' cad visible seulement par les classes du même paquetage (pas de déclaration de visibilité : ex. class MaClasse { … }
  - Donnera lieu a autant de fichiers .class qu'il y a de classes.

- **Convention de nommage :**
  - Une classe commence par une majuscule et doit décrire son contenu : MaClassePourFaireQuelquechose
  - Les variables commencent par une minuscule
  - Nom de paquetage en minuscule organisé en niveau du plus général au plus particulier ex : appli.ihm.dialogues.

2. Principes généraux de la Conception et Programmation Orienté Objet

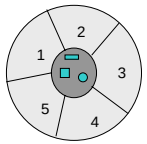
1. Présentation générale de la POO
   - **Qu'est ce que la POO**
   - **Qu'est ce qu'un objet**
   - **Avantage de la POO**
2. Programmer avec des Objets
   - **Représentation d'objets**
   - **Utilisation d'objets**

Qu'est ce que la POO ?

- **Paradigme de programmation** consistant à aborder la **modélisation** d'une **partie du monde réel** (ou **domaine**) à l'aide d'**objets.**
- **Chaque objet** est **représentatif** d'une ou plusieurs **entités réelles** : il caractérise leur **état** et leurs **comportements.**
- **Conception Orientée Objet:** consiste à analyser le **domaine** d'étude et à **identifier** les **objets** qui le composent c'est à dire à **regrouper** en objets des **services** et des **données homogènes**.

Qu'est ce qu'un Objet ?

- **Une entité informatique caractérisée par :**
  - un **état** : formé par l'ensemble des attributs d'un objet (i.e. la valeur de ses **variables**).
  - des **comportements** : définis au travers des **services** réalisable par cet objet (i.e. des fonctions ou méthodes).
- **Exemple:**



Objet Voiture

Exemple d'états et leurs valeurs possibles :

Etat Moteur	[Marche, Arrêt]
Vitesse	[0..130]
Direction	[Avant, Arrière, Gauche, Droite]
Nombre passagers	[0..5]

Exemple de comportements et leur conséquence sur l'état de l'objet Voiture :

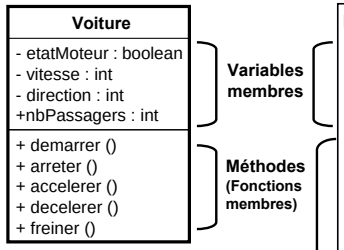
Démarrer	Changer Etat Moteur à Marche
Arrêter	Changer Etat Moteur à Arrêt
Accélérer	Si Etat Moteur Marche augmenter vitesse de 10
Décélérer	Si Etat Moteur Marche diminuer vitesse de 10
Freiner	Si Etat Moteur Marche mettre vitesse à 0
...	

Avantages de la POO

- **Propriété :**
  - **Modularité :** chaque objet est une partition du domaine.
  - **Abstraction, Simplification de la complexité :** chaque objet agrège un ensemble de propriétés du domaine et cache les différences entre entités réelles pour mettre en valeur leurs similitudes.
  - **Encapsulation des données :** on cache certains détails d 'implémentation. Du point de vue de l'utilisateur, l'objet est réduit aux services qu'il rend (boite noire).
  - **Extensibilité :** un objet peut être utilisé pour définir d'autres objets (agrégation, composition), ou être spécialisé (héritage), ...
- **Conséquences :**
  - **Réutilisabilité améliorée :** grâce aux objets déjà définis
  - **MaJ facilitée :** modifications limitées ou plus accessibles ...

Représentation d'objets

- **En UML(Unified Modelling Language)**



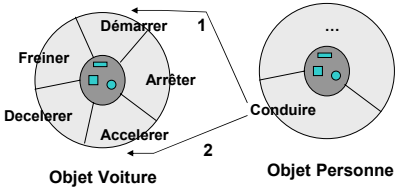
Les **méthodes publiques** forment **l'interface** de l 'objet c.à.d la **partie visible** à **l 'extérieur de l 'objet.**

- **En Java**

```
public class Voiture {
   private boolean etatMoteur=false;
   private int vitesse=0;
   private int direction=0;
   public int nbPassagers=1;
   public void demarrer() {
      etatMoteur=true;
   }
   public void accelerer() {
      if(etatMoteur)
         vitesse= vitesse+10;
   }
   /* définition des autres
   méthodes... */
}
```


Utilisation d'objets

- Les **comportements des objets** sont **activables** par le biais de **messages.**
- Un **message** est un **appel** à l'une des **méthodes** d'un **objet** par un **autre objet.**
- **Exemple :** l'objet Personne envoie des messages à l'objet Voiture. L'état de la voiture est ainsi modifié au niveau de l'Etat Moteur **(1)** et la vitesse du véhicule **(2)**.




```
/*Traduction en Java*/

public class Personne {
 /*..*/

 public void conduire(Voiture v)
 {
   v.demarrer(); /* (1) */
   v.accelerer(); /* (2) */
 }
 /* etc ... */
}
```


3. Base en Java

1. Compléments sur la programmation objet en Java
2. Variables membres
3. Méthodes
4. Constructeurs

Compléments sur la POO en Java
Concepts de classe et d'instance

- **Qu'est ce qu'une classe ?** C'est la **description générale commune** à un **ensemble d'objets**.
  Une Renault 19 ou une Golf GTI partagent un ensemble de **propriétés communes** permettant de les qualifier **toutes deux** de **voiture**. Leurs **différences** se situent au niveau des **valeurs des attributs**.

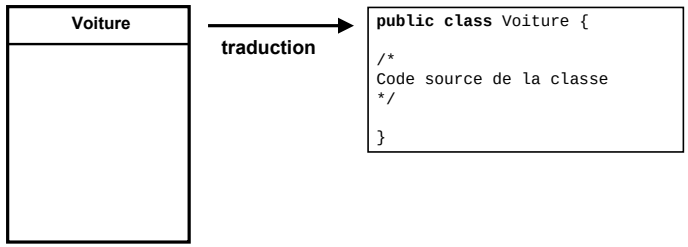
- **La différence entre une classe et un objet** est la même que celle qui **distingue** un **type de donnée** et une **variable** instanciant ce type. C'est la différence entre une **classe d'objet** et d'un **élément particulier** (appelé **instance**) de cette classe.

- **En java, rien ne peut exister en dehors d 'une classe** (à la différence du C++).

- Les **instances** d'une **même classe** partagent ainsi la même **interface** (càd qu'ils ont le même **comportement**) mais peuvent se différencier par leur **état** (càd la **valeur** de leurs **attributs**).

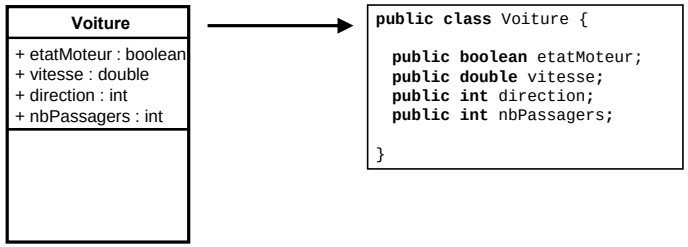
Compléments sur la PO : de UML à Java
Exemple 1/3

- **Définition de la classe voiture**
- **Corps de la classe … là ou tout se passe puisqu'en java tout est objet.**



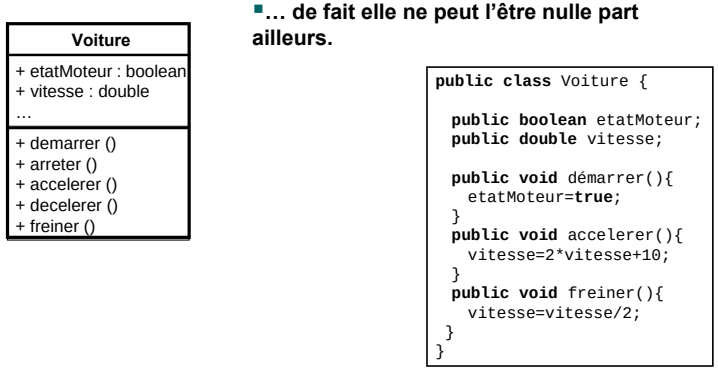
Compléments sur la PO : de UML à Java
Exemple 2/3

- **Définition des attributs/propriétés**
- **Déclaré(e)s dans le corps de la classe.**



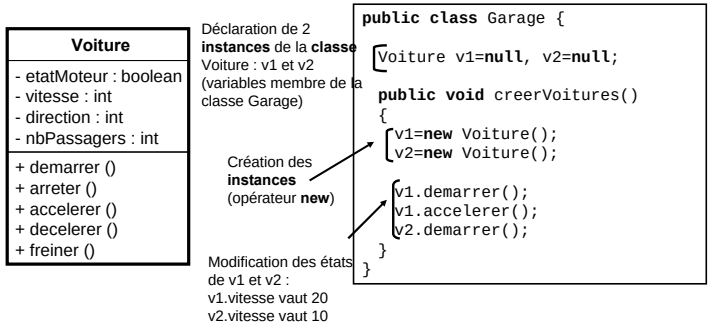
Compléments sur la PO : de UML à Java
Exemple 3/3

- **Définition des méthodes**
- **D'un point de vue programmation, une méthode est une fonction définie à l'intérieur d'une classe…**
- **… de fait elle ne peut l'être nulle part ailleurs.**



Compléments sur la PO en Java
Utilisation d'une classe

- **Définition de la classe voiture**
- **Utilisation de plusieurs instances de la classe Voiture**



Déclaration d'une classe

- **Visibilité peut valoir:**

  + public : dans ce cas : 1) la classe est accessible par toutes les autres classes et 2) le fichier doit s 'appeler `NomClasse.java`. Si un fichier .java peut contenir plusieurs classes il ne peut contenir qu'une seule classe publique.

  + "": i.e. pas de qualificatif de visibilité. La classe n 'est visible que des classes faisant partie du **même** package (i.e. même répertoire).

`Fichier.java`

```
(Visibilité) [Usage] class NomClasse
 {
 /* Déclaration/Initialisation
  des variables membres ou champs*/
 /* Définition des fonctions
 membres ou méthodes */
}
```

- **Usage peut valoir:**

  + **final :** cette classe ne peut avoir de classe **fille**. Héritage interdit.

  + **abstract :** cette classe ne peut être **instanciée.** Elle définit un modèle ou patron de classe. Toute classe héritant de NomClasse doit redéfinir les méthodes définies dans NomClasse.

Instanciation d'une classe

- La **création** d'une **instance** s'effectue à l'aide de l'opérateur **new :**

```
MaClasse objet = new MaClasse();
```

- **Cette opération :**
  - **alloue** la **mémoire** pour contenir **une instance** de la classe MaClasse (opérateur **new**).
  - **appelle** le **constructeur** adéquat pour **initialiser** (en fonction des paramètres) cette instance.
  - **affecte** à la variable objet une **référence** sur l 'objet venant d'être crée (≈ une adresse en C++).
- Il est alors possible **d'envoyer** des **messages** à l'objet : `objet.setIntValue(10);`

Variable de classe et variable d'instance
Présentation

2 grands type de variables existent :

- Les ***variables de classes :***
  - **existent indépendamment** de toute **instance**. Elles sont donc utilisées directement avec le nom de la classe.
  - peuvent par exemple être utilisée pour compter le nombre d 'instances crées.
  - sont déclarées à l'aide du mot clé **static**.
- Les ***variables d'instances*** :
  - **ne peuvent exister** qu'au sein d'un **objet .** Elles ne sont donc accessible qu'après instanciation d'une classe.
  - **caractérisent l'état** d'un **objet** (utilisation similaire à celle d'un **struct** en C).
  - se déclare de la même manière qu'un type primitif.
- **Ex : la classe Integer du paquetage java.lang contient les variables suivantes :**

```
package java.lang;
public class Integer{
   public static final int MIN_VALUE;
   public static final int MAX_VALUE;
   private int value;
/*…*/    }
```

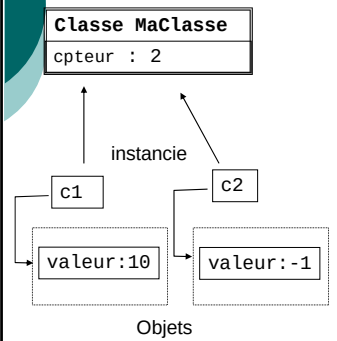
final définie la variable comme étant constante.

i.value ≠ k.value

```
import java.lang.Integer;
public class Test{
  public static void main(String args[]){
    int a = 20;
    Integer i = new Integer(10);
    Integer k = new Integer(a);
    a=Integer.MIN_VALUE;
  }
}
```


Variable de classe et variable d'instance
Illustration

Résultat des instructions de la méthode main de la classe Main




```
public class MaClasse {
   static int cpteur= 0;
   public int valeur=-1;
}
```

```
public class Main {
 public static main(String arg[])
 {
    MaClasse.cpteur++;
    MaClasse c1=new MaClasse();
    c1.valeur=10;
    MaClasse c2=new MaClasse();
    MaClasse.cpteur++;
 }
}
```

Les classes MaClasse et Main doivent se trouver dans le même répertoire, pour que Main puisse accéder à cpteur.

Les variables membres d'une classe : Déclaration, Utilisation (1/2)

- **La déclaration et l'utilisation sont similaires à celle du C/C++ :**
  `[Visibilité][Usage] type nomVar [= valInit]`
- **type :** est soit un **type primitif** (cf. liste ci-après), soit le **nom** d'une **classe** (nomVar est alors une référence).
- **nomVar** : est le **nom identifiant** la variable. Par convention, ce nom s'écrit en minuscules, les majuscules signalant des mots composés.
- **Visibilité :**
  - **+ public :** accessible par tout le monde (ie méthodes de toutes classes).
  - **+ « » ie rien :** niveau **paquetage** visible par les classes dans le même répertoire. Équivalent au **friendly** de C++.
  - **+ protected :** visibilité réduite aux classes de même **paquetage et** aux classes **filles**.
  - **+ private :** variables accessibles **uniquement** depuis la **même classe**.
- **Usage : optionnel,** il peut valoir l'une ou plusieurs de ces valeurs :
  - **final :** la variable est une **constante** (la **1ère valeur** donnée sera la **dernière** ex. dans le constructeur).
  - **static** : cf. variable de classe.
  - **volatile** : lié à une utilisation de threads ou processus léger.
  - **transient** : indicateur utilisé lors de la sérialisation.

../..

Les variables membres d'une classe : Déclaration, Utilisation (2/2)

• La valeur d'initialisation de la variable est utilisée soit au **chargement** de la classe (variable de classe) ou lors de son **instanciation** (variable d'instance);

•**Exemple : …** donne à l 'exécution

MaClasse.java

```
package test;
public class MaClasse
{
 int nb=5;
 float note=12.5;
 String monNom= "Toto!!";

 public static void main(String []args){
  MaClasse ma = new MaClasse();
  System.out.println("nb=" + (ma.nb*10));
  System.out.println("note= "+(ma.note+1));
  System.out.println("monNom= "+ma.monNom);
 }
}
```

⇒

```
C:\>java test.MaClasse
nb=50
note= 13.5
monNom= Toto !!
C:\>
```


Type Particulier : les Tableaux en Java Déclaration, Initialisation et allocation

• **La déclaration de tableaux** diffère des déclarations de variables classiques au niveau du **type** et de l'**initialisation**.

• **Déclaration** : (crochets avant ou apres le type) :

```
type [] nomVar [= {Initialisation}];
```

Exemple : `int [] tabInt; String tabS[];`

• **Initialisation** avec une **liste de valeur** : { val1, val2, …, valn }

Exemple :
```
int [] tabInt = { 10, 20, 30};
String tabS[] = { "aaa", "bbb"};
```

• **Initialisation/Allocation** avec l 'opérateur **new** : **new** type [taille];

Exemple :
```
int [] tabInt = new int[10];
String tabS[] = new String[5];
```

Les tableaux de types primitifs sont initialisés (contrairement au C).

Type Particulier : les Tableaux en Java Utilisation

• L'**utilisation** est **similaire** à celle en C/C++ :
`int tab[]=new int[10]; => tab[0] … tab[9];`

• Exemple dans une boucle **for** :

```
int tab[]=new int[10]; int i;
for(i=0; i<10; i++)
        tab[i]=100*i;
```

• **Taille** du tableau accessible au travers de la variable membre **length** :

```
for(i=0; i<tab.length; i++)
   tab[i]=100*i;
```

Un tableau est donc défini comme un objet (transparent pour l'utilisateur).

• Contrairement au C++, **vérifie** le **non dépassement** du **tableau** (lance une **exception ArrayIndexOutOfBoundException à l'exécution**).

Pour le vérifier il suffit d'essayer d 'accéder à tab[11]; ou de remplacer 10 par 11 dans la première boucle for.

=> Les erreurs de gestion de mémoire sont systématiquement détectées.

Les variables membres d'une classe : Types de données primitifs

- **Types du C++ augmentés du type booléen.**

Type Primitif	Taille (Bit)	Minimum	Maximum	Valeur par défaut (1)
boolean	1(2)	-	-	false
char	16	Unicode 0	Unicode 2^{16}-1	'\u0000' (caractère nul)
byte	8	-128	+127	(byte) 0
short	16	-2^{15}	$+2^{15}$-1	(short) 0
int	32	-2^{31}	$+2^{31}$ - 1	0
long	64	-2^{63}	$+2^{31}$-1	0L
float	32	IEEE754	IEEE754	0.0f
double	64	IEEE754	IEEE754	0.0d
void	-	-	-	-

(1) pour les variables membres d'une classe seulement. Les variables de fonctions non initialisées génère un erreur du compilateur.

(2) en théorie, en fait le compilateur convertit le boolean en int.

Chaînes de caractères : la classe String

- Une **chaîne de caractère** est définie comme une **instance** de la classe **String**.
  Ex: `String s1 = "Hello World";`
- Contrairement au C, la variable **s1** ne contient pas l'adresse d'un tableau de caractères mais une **référence** sur un **objet String** .
- L'opérateur le plus courant est celui de concaténation '**+**'. C 'est la seule **surcharge** d'opérateur autorisée (contrairement au C++).
  Ex: `String s2 = s1 + "\n"; // s2 vaut "Hello World\n"`
- La classe String contient un ensemble de méthodes permettant de manipuler une chaîne de caractères (longueur, transformation,...).
  Ex: `System.out.println("s1 majuscule "+s1.toUpperCase()).`
- La méthode d'affichage `System.out.println()` n'accepte qu'une chaîne de caractères comme paramètre. Aussi, elle convertit d'elle même toute valeur primitive ou objet en String (10.58 devient "10.58") . Ceci est également vrai pour les objets (cf. présentation de la méthode `toString()`).
  Ex: `System.out.println("s1="+s1+" et d'un nombre:"+ 10);`

Méthodes d'une classe Présentation

2 grands types de méthode existent :

- Les ***méthodes de classe :***
  - elles **existent indépendamment** de toute **instance**. Elles peuvent être appelées directement avec le nom de la classe.
  - elles peuvent par exemple être utilisées pour des fonctions non liée à une instance (bibliothèque de fonction comme Math).
  - Elles sont déclarées à l'aide du mot clé **static**.
  - S'utilisent directement avec le nom de la classe.
  - **Ex:** `Integer.parseInt("154");` cette méthode traduit la chaîne de caractères "154" en valeur entière et la retourne (ici 154).
- Les ***méthodes d'instance*** :
  - elles **ne peuvent être utilisée** qu'à travers une **référence d'une instance de classe .** Elles ne sont donc accessible qu'après instanciation d'une classe. Ex: `new Integer(10).toString();`
  - Elles **caractérisent** le **comportement** d'un **objet.**
  - Ne nécessitent pas de déclaration particulière.

Définition d'une méthode (1/2)

- **La déclaration et l'utilisation sont similaires à celle du C/C++ :**

```
(Visibilité)(Usage) Type nomMethode (Parametres){
 /* code */ }
```

- **Visibilité** définit l'**accessibilité** de la **méthode.** Elle peut valoir **private, protected, public** ou rien (cf. transparent 69)
- **Usage** peut valoir :
  - **final** : lié à l'héritage, empêche le redéfinition de la méthode
  - **static** : méthode de classe, équivalent à une méthode globale
  - **native** et **synchronized** seront abordé plus tard.
- La **valeur de retour** de la méthode indiquée par **Type** peut être un **type primitif** (**void** compris) ou le **nom** d'une **classe.**

Définition d'une méthode (2/2)

- **Parametres** peut valoir **void** ou une **suite de paramètres** séparés par une virgule. Un paramètre est défini par le **type de données** (primitif ou classe) et le **nom** d'une **variable.**
- Les **paramètres** sont passés ***uniquement* par valeur** pour les types primitifs et par **adresse** pour les objets (contrairement au C++) .
  Une fonction de permutation de valeur ne peut donc passer par des types primitifs mais par des objets!
- **Ex:**

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

```
public static void setValue(Etudiant e, int note){
  e.note = note;
}
```


Déclarations de variables locales et instructions dans méthode

- **La déclaration d 'une variable :**
  - est **similaire** à celle d 'une **variable d'instance**.
  - peut être effectuée **n'importe où** dans un **bloc** … mais **avant** sa **première utilisation**.
  - doit **initialiser** la variable (génère une **erreur** alors que C++ un warning!)
  - **Portée et durée de vie** limitée à celle du **bloc**.
- **Ex:**

```
public int truc(int x)
{
 int a=10*x;
 {
   int b=2;
   for(int i=0; i<10; i++)
     b+=i*10; // i détruit a la fin du for
   a=a+b;
 }   // b detruit apres
 return a;
} // a et x detruits a la fin de la fonction
```


Une méthode particulière : la méthode toString()

La méthode toString()

- retourne une **chaîne de caractère (String)**
- est utilisée pour **convertir un objet en chaîne de caractère** lorsque l'objet est passé **en paramètre d'une méthode** nécessitant un **String (≈cast implicite)**
- Par défaut renvoie le nom de la classe et la référence de l 'objet `(Titi@111f71)`

Ex: la méthode d'affichage System.out.println()

```
public class Test{
 int a=0; int b=0;
 public String toString() {
   return "(test <a="+a+",b="+b+">)";
 }
 public static void main(String arg[]) {
   Test t = new Test();
   t.a=10; t.b=12;
   System.out.println("Res :"+t);
 }
}
```

⇒

```
C:\>java Test
Res :(test <a=10,b=12>)
C:\>
```


Surcharge de méthodes :
Principe

- **Consiste** à utiliser des **nom identiques** pour des **fonctions** effectuant le **même rôle** mais **utilisant** des **types de données d'entrée différents.**
- **Ex:** l'opérateur '+' ne joue pas le même rôle suivant les opérandes :
  `2 + 6` , `2.54 + 3.14` et `"tot"+"o"`.
  Les calculs effectués **diffèrent** selon le **type** de donnée utilisé (int, float ou String). C'est la seule **surcharge d'opérateur** autorisée par java (contrairement au C++).

- **Intérêts :**
  - Meilleure **lisibilité** du programme (un seul nom pour un seul rôle ex: somme(.) en lieu de sommeInt(.), sommeFloat(.), sommeString(.).
  - **Masque** la **différence** d'implémentation pour **souligner** le **comportement analogue**.

Surcharge de méthodes :
Exemple

- **La signature** de la méthode permet d'**identifier** la méthode à utiliser : `nom_methode + nombre_param + type_param`
- La **valeur de retour** n'est pas **utilisée** dans la **signature !**
- **Exemple :**

```
public class MaClasse{
 String nom="a";
 void println(String a) {
 System.out.println("String<"+nom+">:"+a);}
 void println(int a){
  System.out.println("Int<"+nom+">:"+a); }
 void println(float a){
  System.out.println("Float<"+nom+">::"+a);
}
 public static void main(String args[]) {
        MaClasse ma=new MaClasse();
        ma.println(2);
        ma.println(2.3f);
        ma.println("Hello"); }
}
```

Rque : 2.3 est par défaut stocké en **double**. Sans ajouter 'f' le compilateur refuse le risque de perte d'erreur en interdisant l'appel à println(**float**) et en déclarant ne pas trouver de méthode println(**double**) [cannot resolve symbol]

L'autre solution consiste à utiliser le type le plus général et déclarer println(double).

Constructeur d'objets :
Principe

- **Méthode particulière dédiée à l'instanciation d'une classe**
- **Un constructeur :**
  - **initialise** l'état de l'objet.
  - **porte** le même nom que la classe.
  - est appelé via l'opérateur **new** (qui **alloue la mémoire** pour contenir l'objet).
  - comme toute méthode peut être plus ou moins visible depuis les autres classes (private, protected, "package" ou public)
- **Signification du this :**
  - comme en C++ c'est une **référence** sur l'**objet courant**. Peut être utilisé pour distinguer des variables d'instances de variables locales.
  - **this(...)** signifie appel du constructeur depuis un autre constructeur. => Première instruction du constructeur.

Constructeur d'objets :
Exemple


```
class MaClasse{
 int a;
 float b;
 String nom;
 Object o [];
 private MaClasse() { o = new Object[10];}
 public MaClasse(int a, float b) {
    this.a=a; this.b=b; nom=""; }
 public MaClasse(int a, float b, String nom) {
    this(a,b); this.nom=nom; }
}
public class Main{
 public static void main(String arg[]) {
    MaClasse maClasse = new MaClasse(10,20f,"toto");
    //MaClasse maClasse = new MaClasse();
 }
}
```

private empêche l'appel du constructeur par défaut à l'extérieur de la classe : on veut forcer l'utilisateur de la classe à donner des valeurs initiales

Réutilise le deuxième constructeur

Génère une erreur car le constructeur par défaut n'est pas public !

Le constructeur par copie

- **Principe :** On créé un nouvel objet à partir d'un autre. Peut s'avérer pour créer des copies de sauvegarde avant modification.

```
class MaClasse{
 int a;
 float b;
 String nom;
 public MaClasse(int a, float b, String nom) {
    this.a=a;   this.b=b;   this.nom=nom;  }
}
 public MaClasse(MaClasse mc) {
    a=mc.a; b=mc.b; nom=mc.nom;   }
}
 public static void main(String arg[]) {
    MaClasse mc1 = new MaClasse(10,20, "toto");
    MaClasse mc2 = new MaClasse(mc1);  }
}
```

On aurait pu écrire nom=new String(mc.nom); pour que this.nom et mc.nom contiennent la même valeur mais ne "pointent" pas sur la même référence. Or comme un String ne peut être modifié on peut se limiter à copier les références.

On construit mc2 d'après mc1. Donc mc1!=mc2 même s 'ils ont les mêmes valeurs d 'attributs.

Récapitulatif sur la visibilité des classes

- **Visibilité des membres d'une classe depuis d 'autres classes**

Canari.java

```
package Oiseaux;
public class Canari {
  public int a;
  protected int b;
  int c;
  private int d;

  public void A() {/*…*/ }
  protected void B(){/*…*/}
  void C()  {/*…*/}
  private void D() {/*…*/}
}
```

Mesange.java

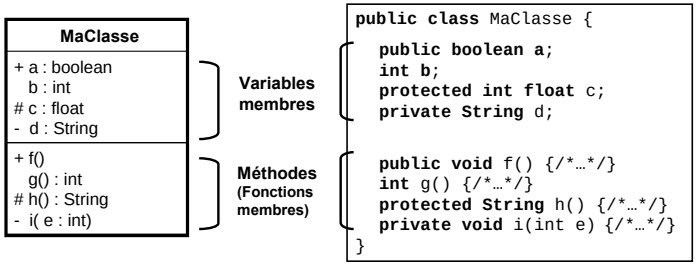
```
package Oiseaux;
public class Mesange{
 void F() {
  Canari c = new Canari()
    c.A(); // Ok
    c.B(); // oui car package
    c.C()  // idem
    c.D(); //erreur a la compil
 }
}
```

Persan.java

```
package Chats;
public class Persan{
 void G(Canari c) {
  c.A(); // Ok
  c.B(); //erreur a la compil
  c.C()  // idem
  c.D(); // idem
 }
}
```

Récapitulatif sur la notation UML

- **Visibilité des membres** d'une **classe** depuis **d'autres classes :**
  - **public** : '+'
  - "**package**" : ''
  - **protected** : '#'
  - **private** : '- '
- **Exemple :**



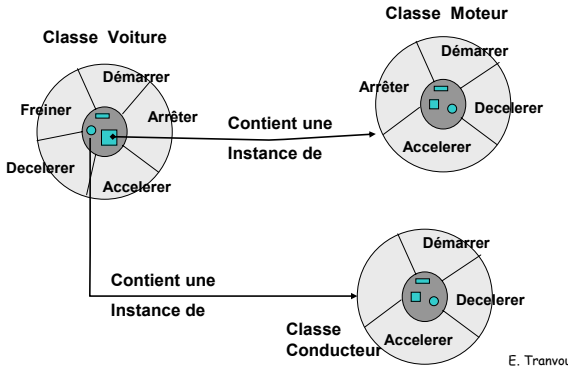
4. POO - Composition

- Principe et notation UML
- Composition par valeur
- Composition par référence

Relations d'associations entre classes :

Principe

- **L'attribut d'une classe est elle même une classe**
- **Exemple :**



Relations d'associations entre classes :
Intérêt de l'encapsulation

- **Gestion de la complexité de conception :**
  - Le problème est décomposé en sous problèmes de dimensions plus restreintes.
  - Masquer la complexité en limitant ce qu'a à savoir un développeur
    - Ex: Pour conduire une voiture il suffit de savoir qu'il faut appuyer sur un bouton pour la démarrer … à la limite il n'est pas utile de connaître la chaîne de commande entre l'appui de ce bouton et sa conséquence sur le moteur de la voiture (ou en tout cas pas en détail).
- **Gestion de la complexité d'implémentation**
  - Même idée que pour la compilation séparée en C !
  - Faciliter la mise à jour des codes (des classes) en séparant les missions de chaque classes tout en gardant des liens entre elles.
    - Je peux modifier le moteur sans modifier la manière de conduire la voiture.
  - Fournir au programmeur une interface càd des points de contrôle sur l'objet sans qu'il ait nécessairement besoin de savoir comment sa « commande » sera réalisée en interne dans la classe
  - Càd séparer le Quoi du Comment …

Association de type Composition

- **La relation d'association** entre deux objets de A vers B est qualifiée de **composition si :**
  - **B ne peut être définie sans A**
  - **La durée de vie de B est la même que celle de A (construction et destruction)**
- **Similaire à la notion d'inclusion** (B est inclus dans A)
- **Exemples :**
  - **Voiture / Moteur**
  - **Livre / Page**
  - **Ecole / Eleve**
- … La **classe englobante** est responsable de l'**initialisation** des **objets membres.**
  => appeler le **constructeur** des objets membres depuis le constructeur de la classe englobante**.**

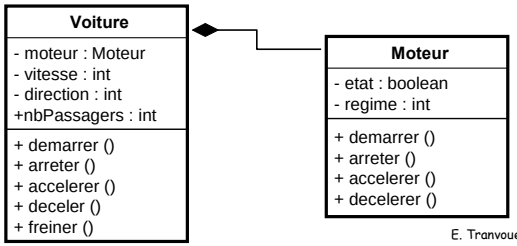
Compositions d'objets :
Notation UML

- **Symbole de composition :**

A ◆——— **B** se lit : **A est composé de B**

Le fond plein peut s'interpréter comme une association « bloquée » … B est lié à jamais à A.

- **Exemple :**



Compositions d'objets :
Code Java correspondant


```
// Moteur.java

public class Moteur {

 private boolean etat;
 private int regime;

 public Moteur() {
   etat=false;
 }

 public void demarrer() {
   etat=true;
}

 public void accelerer() {
   regime++;
 }
/* ...*/
}
```

```
// Voiture.java

public class Voiture {

 private Moteur moteur=null;

 public Voiture() {
   moteur=new Moteur();
// C'est Voiture qui créé le moteur
}

 public Voiture(int nbPassagers)  {
   this();  /* fait appel au
constructeur par défaut de la classe
Voiture cad « soit meme » */
   this.nbPassagers= nbPassagers;
 }

 public void demarrer() {
   moteur.demarrer();
 }
/* ... */
}
```

Composition par référence : l'agrégation

- Définition **complémentaire** de la composition, on parle aussi de composition par référence.
- **Symbole** **A** ◇——— **B**
- **Cas** où un objet **B** est simplement **référencé** dans **A sans** que **A ait pour autant aucune responsabilité envers l'objet B :**
  - B peut être défini sans A.
  - La durée de vie de B est indépendante de celle de A.
- La charge de construction ou de destruction ne lui est pas (forcément) dévolue
- **Exemples :**
  - Parent / Enfant
  - Livre / Auteur
  - Parking / Voiture

Agrégation d'objets :
Code Java correspondant


```
// Conducteur.java

public class Conducteur {

 private String nom;
 private boolean permisB;

 public Conducteur(String nom,
            boolean permisB) {
   this.nom=nom;
   this.permisB=permisB;
 }

 public boolean peutConduire(){
   return permisB;
}

/* ...*/
}
```

```
// Voiture.java

public class Voiture {

 private Moteur moteur = null;
 private Conducteur c=null;


 public Voiture() {
   // Voiture sans conducteur
   moteur=new Moteur();
}
 public Voiture(Conducteur c) {
   this();
   this.c=c; }

 public void changerConducteur(Conducteur c){
         this.c=c;
}

 public void demarrer() {
   if(c!=null && c.peutConduire() )
         moteur.demarrer();
 }
/* ... */
}
```

Exemple d'Initialisation d'objets agrégés


```
// Livre.java

public class Livre {

 String titre;
 Auteur auteur;

 public Livre(Auteur a, String t) {
 auteur=a;
 titre=t;}
 public Livre(String t) {
 auteur=new Auteur();
 titre=t;}
}
```

Pas de constructeur par défaut pour Livre. On ne peut pas créer de Livre sans Auteur.

Constructeur par défaut Auteur() utilisé pour les auteurs inconnus.

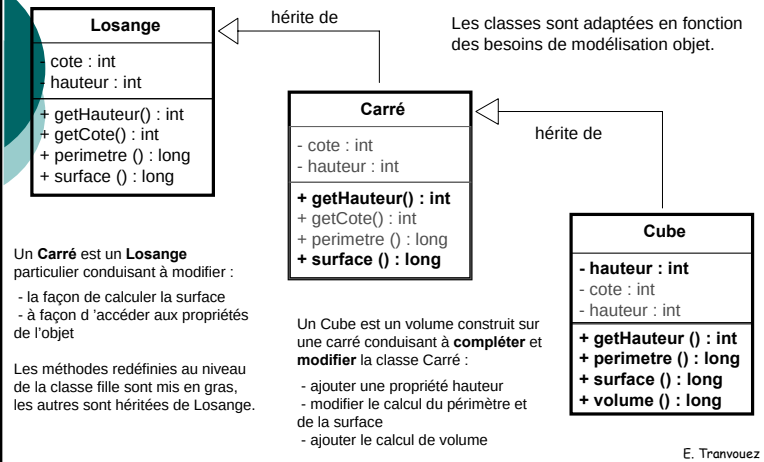
```
// Auteur.java

public class Auteur{
  static final String INC="Anonyme";
  String nom;
  String prenom;

  public Auteur() {
    nom=Auteur.INC;
    prenom=Auteur.INC;
}
  public Auteur(String n, String p)
  { nom=n;
    prenom=p;   }
}
```

```
// Main.java

public class Main {
 public static void main(String args[]) {
   Auteur a = new Auteur( "Hergé", "Remy");
   Livre l1 = new Livre( a, "Tintin");
   Livre l2 = new Livre( a, "Quick et Flupke");}
}
```


5. POO - Héritage Simple

- Principe et intérêt
- Redéfinition et surcharge
- Visibilité liée à l'héritage
- Polymorphisme

Première approche



Les classes sont adaptées en fonction des besoins de modélisation objet.

Un **Carré** est un **Losange** particulier conduisant à modifier :

- la façon de calculer la surface
- à façon d 'accéder aux propriétés de l'objet

Les méthodes redéfinies au niveau de la classe fille sont mis en gras, les autres sont héritées de Losange.

Un Cube est un volume construit sur une carré conduisant à **compléter** et **modifier** la classe Carré :

- ajouter une propriété hauteur
- modifier le calcul du périmètre et de la surface
- ajouter le calcul de volume

Principe de l'héritage

- L 'héritage consiste à **réutiliser** la définition d'une classe (appelée **classe** mère) pour obtenir une nouvelle classe (appelée classe **fille**).
- La classe **fille :**
  - est une **spécialisation** de la classe **mère.**
  - **hérite** des **propriétés** et **comportements** de la classe mère…
  - … qui peuvent être **modifiés** ou **complétés**
  - partage donc la même interface : une classe **fille** peut être utilisée comme une classe mère.
  - est une classe **mère plus** autre chose.
- **Tout classe java hérite, directement ou non, de la classe Object.**

  => toutes les classes partagent une interface commune minimale (toString() par exemple)**.**

Intérêts de l'héritage

- **Réutilisation :**
  - ne pas **tout refaire** quand de **nouveaux besoins** apparaissent
  - **maintenir** une **cohérence** dans le code déjà développé

⇒ **pas de remise en cause** du code existant : on étends les propriétés et comportements en rajoutant ou modifiant les classes déjà définies

Syntaxe Java pour l'héritage
le mot clé **extends**

- La classe fille étends les propriétés et comportements de la classe mère

```
class Mere {
  public int a=1;
  public int getA() { return a;}
}
class Fille extends Mere {
  public int a=2;
  public int getA() { return a;}
  public int getOldA() {return super.a; }
}
public class Main{
  public static void main(String args[]) {
    Mere m = new Mere();
    Fille f= new Fille();
    System.out.println("m.a="+m.getA());
    System.out.println("f.a="+f.getA());
    System.out.println("f.a hérité ="
              +f.getOldA());}
}
```

super équivalent au **this** de la classe mère.
On peut ainsi directement accéder au champs et méthodes de la classe mère.

Remarque : toute classe java hérite de la classe **Object** et partagent donc une interface minimum

⇒

```
C:\>java Main
m.a=1
f.a=2
f.a hérité=1
C:\>
```

Redéfinition des méthodes héritées

- Tout en gardant la même **interface**, la classe fille peut modifier son comportement.

```
class Mere {
  public int a=1;
  public String toString() {
    return "Mere : a="+a;}
}

class Fille extends Mere {
  public int b=2;
  public String toString() {
    return "Fille : a="+a+" b="+b;}
}

public class Main{
  public static void main(String args[]){
    Mere m = new Mere();
    Fille f= new Fille();
    System.out.println(m);
    System.out.println(f);
  }
}
```

Ce qu'on obtient si Fille.toString() est commentée.

⇒

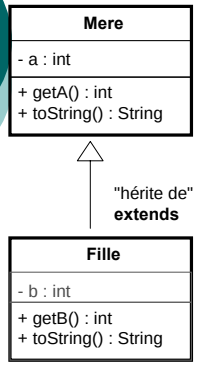
```
C:\>java Main
Mere : a=1
Mere : a=1
C:\>
```

Ce qu'on obtient si Fille.toString() redéfinie Mere.toString().

⇒

```
C:\>java Main
Mere : a=1
Fille : a=1 b=2
C:\>
```


Appel à une méthode d'une classe fille



1. La méthode n'existe que dans la classe mère :

Fille f=new Fille();
f.getA();

=> Le message est "transmis" à la partie héritée, et c'est la méthode getA() de la classe mère qui est exécutée.

2. La méthode a été surchargée dans la classe fille :

Fille f=new Fille();
f.toString();

=> Alors le message est intercepté par la classe fille.

3. La méthode n'existe que dans la classe Fille :

Fille f=new Fille();
f.getB();

=> Fonctionnement habituel d'envoi d'un message à un objet.

Héritage et instanciation : conséquence sur les constructeurs

- L'**instanciation** d'une classe **fille** impose celle de la classe **mère**.

- Java fait **automatiquement** appel, **depuis** le constructeur de **la classe fille**, au **constructeur par défaut de la classe mère**. Si ce constructeur n'existe pas, le compilateur le signale par une erreur.

- On peut **explicitement** faire appel au constructeur de la classe mère avec le mot clé **super()** (fonctionne comme **this()** mais il n 'est pas possible de faire les deux en même temps. this() ou super() doivent en effet être impérativement la première instruction du constructeur).

Héritage et instanciation : Exemple

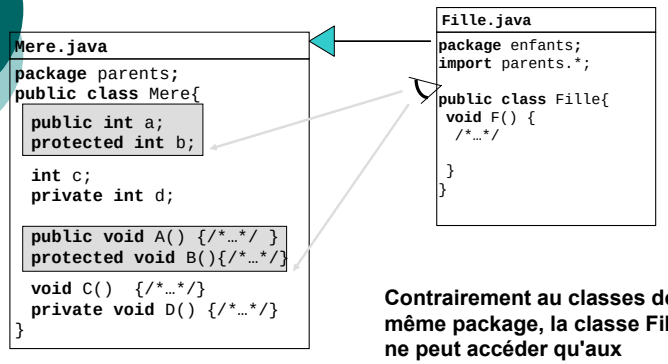

```
class Mere {
  public Mere(){
    S.o.pln("Constructeur Mere"); }
}

class Fille extends Mere{
  public Fille(){
    S.o.pln("Constructeur Fille");
}
  public Fille(String nom){
    S.o.pln("Constructeur Fille <"+nom+">");}
}

public class Main {
  public static void main(String arg[]) {
    Mere m  = new Mere();
    Fille f = new Fille("f");
  }
}
```

⇒

```
C:\>java Main
Constructeur Mere
Constructeur Mere
Constructeur Fille <f>
C:\>
```

Retour sur la visibilité des classes

- **Visibilité des membres d'une classe Mere depuis une classe Fille**



Contrairement au classes de même package, la classe Fille ne peut accéder qu'aux membres public et protected

Exercice

- **Modéliser les informations suivantes:**
  - Un **compte en banque** est possédé par une personne (identifiée par son nom), est identifié par un numéro de compte (défini une fois pour toute), dispose d'un solde et d'un niveau de découvert standard (défini par la politique de la banque), un niveau de découvert propre au compte (modifiable n'importe quand). Il doit être possible de déposer et de retirer de l'argent sur le compte tout en respectant les autorisations de découvert.
  - Un **compte épargne** est un **compte** sur lequel on ne peut retirer d'argent mais par contre qui peut être soldé… bloquant ainsi toute nouvelle opération et renvoyant le solde du compte.
  - Un **compte associatif** peut avoir une liste de personnes habilitées à gérer le compte.
  - Une **banque** gère une liste de compte. Elle peut créer un compte, le solder, et le bloquer temporairement.
  - Un organisme peut prélever des sommes sur un compte pour peu que le compte le mentionne et en tout les cas doit passer par une banque.
  - **Question**: l'une de ces propositions met à mal l'héritage. Pourquoi et comment y remédier…

Exercice 2

- Créer une classe Personne contenant les propriétés suivantes : nom, prénom, age, adresse; ainsi que les constructeurs adéquats (pas de constructeur par défaut) et les méthodes d'accès habituelles (ex : getNom()) sans oublier la méthode toString().
- Créer une classe Etudiant héritant de la classe Personne, possédant en plus les attributs numCE (numéro de Carte Etudiant) et le nom de l'établissement dans lequel s'inscrit un étudiant. Vous définirez ou redéfinirez les méthodes qui vous paraissent nécessaires.
- Créer une classe EtudiantSerieux héritant de la classe Etudiant, possédant en plus l'attributs nbJoursSerieux. Vous définirez ou redéfinirez les méthodes qui vous paraissent nécessaires.
- Créer une classe Film possédant les attributs titre, age minimum, prix et réduction étudiant. Vous définirez 2 méthodes facturer(Personne p) et facturer(Etudiant e) qui retournent le prix pour assister au film en distinguant la facturation d'une personne et d'un étudiant (prise en compte de la réduction étudiante). Vous définirez également 3 méthodes admettre(X x) pouvant prendre en paramètre respectivement une personne, un étudiant et un étudiant Polytech et retournant un booléen indiquant si l'objet passé en paramètre peut visionner le film. Pour une personne cela consiste à vérifier son âge.
- La direction du Cinéma a décidé d'interdire l'accès aux personnes se déclarant étudiant de Polytech'Marseille. Vous prendrez en compte cette contrainte tout d'abord au niveau de la méthode admettre(EtudiantSerieux e) en retournant false et en affichant le texte suivant "Non M/Mme DUPONT ! Allez plutôt travailler le TP Java". Pour éviter qu'un tel étudiant essaie de passer outre l'interdiction en se faisant passer pour un étudiant quelconque[1] (classe Etudiant) vous vérifierez, au niveau de la méthode admettre(Etudiant e), avec l'opérateur instanceof, que e est bien est un EtudiantSerieux. Le cas échéant vous démasquerez l'usurpateur en lui réitérant l'ordre d'aller travailler sérieusement.
- L'opérateur instanceof renvoie vrai ou faux selon qu'un objet instancie une classe ou non. Ex : "Hello" instanceof String renvoie vrai.
- [1] En effectuant un cast c'est-à-dire une conversion forcée de type.

Polymorphisme
Définition

- Exemple :
  - soit une classe B héritant d'une classe A
  - soit b une instance de B
  - Alors b peut être également perçu comme une instance de la classe A.
- Explication :

  *b peut revêtir toutes les identités de sa classe et de celles dont elle hérite : elle peut prendre **plusieurs formes***
- Conséquence : Généricité des passages de paramètres

  Cas particulier la classe Object.

Polymorphisme
Exemple


```
package Chiens;
public class Caniche extends Chien {
 public Caniche(String nom) {super(nom);}
 public String aboyer() {
         return "Caniche-"+nom+" glapit.";}
}
```

```
package Chiens;
public class Labrador extends Chien {
 public Labrador(String nom) {super(nom);}
 public String aboyer() {
         return "Labrador-"+nom+" mugit.";}
}
```

```
package Chiens;
public class Chien {
 protected String nom;
 public Chien(String nom) { this.nom=nom;}
 public String aboyer() {
         return "Chien-"+nom+" aboie.";}
}
```

```
package Chiens;
public class Main {
 public static void entendre(Chien c) {
  System.out.println(c.aboyer());
 }
 public static void main(String arg[])
 {
    entendre(new Chien("Medor") );
    entendre(new Caniche("Tommy") );
    entendre(new Labrador("Blacky"));
  }
}
```

⇒

```
C:\>java Main
Chien-Medor aboie.
Caniche-Tommy glapit.
Labrador-Blacky mugit.
C:\>
```


Contrôle de l'héritage : le mot clé **final**

- **Associé à une classe : empêche toute possibilité d'héritage.**
- **Associé à une méthode : empêche la redéfinition de la méthode par les classe filles.**
- **Intérêt :**
  - Protège toute modification des comportement de l'objet.
  - On contrôle l'avenir de l'objet en protégeant certaines parties du code.
  - Remarque : une méthode finale apporte un gain de rapidité.

Identification du type de classe : l'opérateur **instanceof**

- Permet de connaître le type de classe d'un objet lors de l'exécution du programme !
- Utilisation :
  - `objet instanceof Classe`
  - `a instanceof String`
  - renvoi **true** ou **false** selon que le lien d'instance est avéré ou non.
- Conséquence : Généricité des passages de paramètres
  Cas particulier la classe Object.

Exemple d'utilisation de l'opérateur **instanceof**


```
public class Main {
  public static void placer(Chien c) {
    if( c instanceof Caniche)
      System.out.println("A la maison "+c);
    else
      if( c instanceof Labrador)
        System.out.println("A la niche "+c);
      else
        System.out.println("Je ne sais pas pour "+c);
  }

  public static void main(String arg[]) {s
    placer(new Chien("Medor") );
    placer(new Caniche("Tommy") );
    placer(new Labrador("Blacky"));
  }
}
```

⇒

```
C:\>java Main
Je ne sais pas pour Chien-Medor
A la maison Caniche-Tommy
A la niche Labrador-Blacky
C:\>
```

6. POO - Héritage Multiple

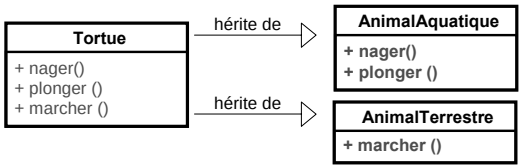
- Principe et intérêt
- Définition d'une interface
- Retour sur le polymorphisme
- Autre utilisation des interfaces

Principe de l'Héritage Multiple

- Il s'agit de donner la possibilité à un objet d'**implémenter** plusieurs **interfaces** c'est-à-dire de comprendre plus de **messages** que lui ou ses ascendants **directs** (classe Mere et ainsi de suite) ne le pourraient.

Exemples :	**Mais**
▪ Un Labrador est un Chien	▪ Une Tortue est un Animal Marin
▪ Une tortue est un Animal	▪ Une Tortue est un Animal Terrestre

• Une tortue **hérite** donc des comportements (nager, marcher) et propriétés des animaux terrestres et aquatiques. Elle possède **plusieurs ascendant directs**.



L'héritage Multiple en Java

- En **Java**, la relation d'**héritage simple** ne permet de définir qu'**1 ascendant contrairement** à d'autres langages objets comme le **C++**.
- Il est cependant possible d'effectuer un héritage multiple au travers des **interfaces**. Cet héritage est cependant limité à la description de l'interface héritée, soit concrètement au nom et paramètres des méthodes héritées et pas à leur **définition**.
- Toute méthode définie dans une interface est considérée comme **public**.
- L'héritage au travers d'une interface ne permet pas de transmettre des variables comme dans le cas de l'héritage simple. Toute **variable définie** dans une **interface** est **automatiquement** convertie en **static final**.

Définition d'une interface

- Se déclare comme une classe(1) : **class** est remplacé par **interface**(2) et **extends** par **implements.**

```
package heritage;
public interface Aquatique {
  boolean FLOTTE = true;
  public boolean nager() ;
}
```

La variable flotte sera considérée comme public static final. C'est une constante globale.

```
package heritage;
public interface Terrestre {
  public void marcher() ;
}
```

Pas de corps de méthode. Equivalent d'une déclaration en C.

```
C:\>java Test
Je nage !!
Je marche!!
Je flotte ? : true
```

```
package test;
import Heritage.*;
class Tortue implements Aquatique , Terrestre {
 public void marcher() {
    S.o.p ("Je marche!!");
  }
  public boolean nager() {
    S.o.p ("Je nage !!");
    return true; }
}
public class Test {
  public static void main(String args[]) {
    Tortue t = new Tortue();
    t.nager();
    t.marcher();
    S.o.p ("Je flotte ? : " + Tortue.FLOTTE );
   // t.FLOTTE=false;
  }
}
```

Les **interfaces** implémentées sont séparées par une virgule.

La classe « héritière » doit implémenter les méthodes qu'elle hérite. Sinon génère une erreur de compilation.

La modification n'est pas possible : cela provoquerait une erreur de compilation

(1) Cela implique l'obligation de nommage du fichier java si l'interface est publique.

(2) Avec JBuilder, utiliser le menu `Fichier>Nouveau>Interface`.

Polymorphisme avec les interfaces

- Une classe "héritant" via une interface, peut se réclamer de cette identité tout comme avec l'héritage simple. => **instanceof** fonctionne avec les interface

```
package test;

import heritage.*;

public class Test2 {
  public static void transporter(Aquatique a) {
    S.o.P(" Bac a Eau > "+a);  }
  public static void transporter(Terrestre a) {
    S.o.P(" Boite pour "+a);  }
  public static void transporter(Object o) {
    S.o.P (" Que faire de "+o);  }
  public static void main(String args[]) {

  public static void main(String args[]) {
    Tortue t = new Tortue();
    transporter( (Aquatique) t);
    transporter( (Terrestre) t);
    transporter( "Truc");
    boolean estAqua = t instanceof Aquatique;
  }
}
```

⇒

```
C:\>java test.Test2
Bac a Eau > heritage.Tortue@720eeb
Boite > heritage.Tortue@720eeb
Que faire de Truc
C:\>
```

Le **cast** est obligatoire car sinon il y a une ambiguïté avec Tortue puis qu'elle est à la fois Aquatique et Terrestre.

Vaut **true**

Autres utilisations des interfaces

- Il est possible d'étendre une interface.

```
public interface SeDeplace {
  public void marcher();
}

public interface SeDeplaceVite extends SeDeplace {
  public void courir();
}
```

- Il est également possible d'utiliser une interface pour **regrouper** des **constantes** (équivalent d'un `enum` en C).

```
public interface Eval {
  int JANVIER = 1, FEVRIER=2, … , DECEMBRE = 12;
}
```


7. Annexes

- Configurer son ordinateur pour programmer avec Java
- Programmer avec JBuilder

Configurer Windows 9x

- **Une variable d'environnement est une variable globale utilisée par certains programmes pour récupérer des informations système.**
- **Ex de variables systèmes :**
  - **PATH** : liste des chemins séparés par des ';' où windows/dos cherche les programmes exécutés sans indication de chemin (outre le répertoire courant). ex de valeur : c:\windows;c:\windows\command). Utilisé également par Unix/Linux.
  - **TEMP** : chemin du répertoire où Windows doit stocker les fichiers temporaires.
  - **CLASSPATH** : utilisé par java et javac pour indiquer les chemins d'accès au classes utilisées dans vos programmes (peuvent être des répertoires ou des fichiers .jar).

Configuration dans Windows 9x (95/98/Me)

- Il faut modifier le fichier `c:\autoexec.bat` en ajoutant en fin du fichier deux lignes ressemblant aux suivantes :
  - **SET PATH=%PATH%;C:\jdk1.4.1\bin**
    Si vous avez installé le kit de développement java (SDK) dans le répertoire C:\jdk1.4.1\bin. **%PATH%** renvoie la valeur de la variable PATH. Ainsi, on ajoute des chemins à cette variable sans écraser les anciennes valeurs (ex: C:\Windows).
  - **SET CLASSPATH=c:\dev\classes;d:\apis\monapi.jar**
    Par exemple pour pouvoir appeler la classe Main du paquetage test situé dans le répertoire c:\dev\classes\test.
- Selon cette configuration vous pourrez exécuter `java test.Main` depuis n'importe quel répertoire.

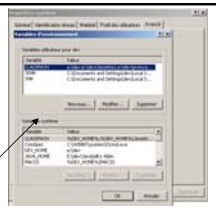
Configurer Windows NT/2000/XP

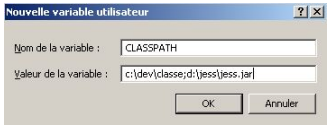
Même principe qu'avec Windows 9x (rôle et valeur). Seule change la façon de modifier les valeurs des variables d'environnement.

Il faut :

1. **A**ller dans : Démarrer> Paramètres> Panneau de configuration>Système.
2. **C**liquer sur l'onglet Avancé.
3. **R**echercher et cliquer le bouton "**Variables environnement**".
4. **D**ans le cadre "**variables utilisateur pour xxx**" (ou xxx correspond à votre login), ajouter (si elle n'existe pas déjà) une variable **CLASSPATH** contenant les chemins d'accès à vos packages (répertoire, fichier .jar).
5. **Ajouter votre propre variable PATH**.

Des exemples de valeurs sont donnés au transparent précédent.





Configurer Linux

Même principe que sous Windows. Seule change la façon de définir les valeurs des variables d'environnement.

Les séparateur de chemin sont des '/' (à la place de '\') et pour séparer plusieurs chemins utiliser ':' (à la place de ';').

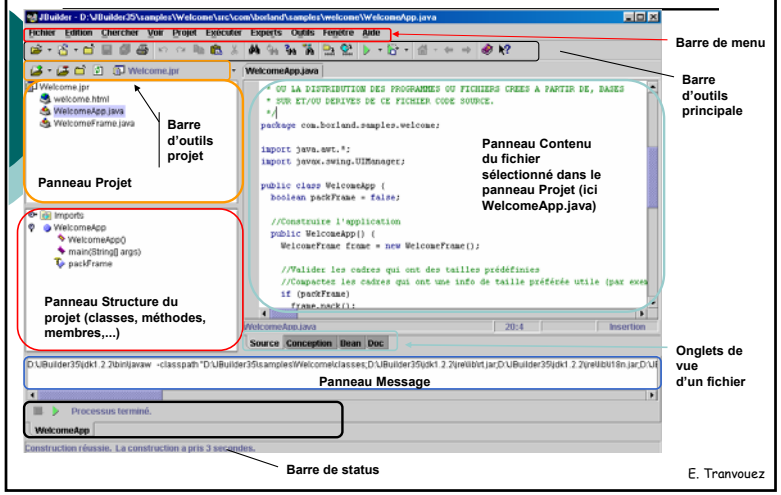
Avec un shell bash :

1. **Editer le fichier ~/.bash_profile**
2. **Définir une variable CLASSPATH :**
   - CLASSPATH=/home/moi/classes:/home/lui/pub/classes
3. **Redéfinir une variable PATH :**
   - PATH=$PATH:/usr/bin

Vous pouvez également définir dans ce fichier vos propres alias.

Ex : alias dircol="ll –l --color"

Présentation de JBuilder



Qu'est ce que JBuilder ?

JBuilder® est un **environnement de Développement Rapide d'Applications** Java (RAD).

Il propose :

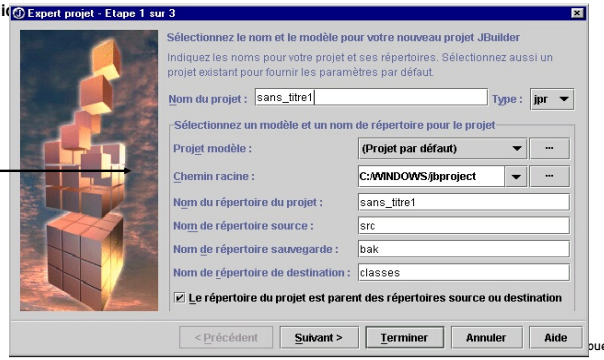
- Un **éditeur de programmes** Java très complet (aide à la saisie avec complétion des mots utilisés dans le programme, vérificateur de syntaxe java, …)
- **Un concepteur d'interface graphique** (bibliothèques AWT, Swing, …)
- **Un débuggueur** évolué
- Des **bibliothèques** de fonctions Java **riches** (accès bases de donnée, …)
- Suivant les éditions une **gestion des versions** etc…
- JBuilder est gratuit pour un usage personnel (Personal Edition)

Premier Programme Java avec JBuilder (1/4)

1. Pour écrire un programme avec JBuilder, il faut d'abord créer un projet :

   1.1. Menu Fichier>Nouveau Projet pour faire apparaître la **boîte de dialogue de l'Expert projet**

   1.2. Indiquer l'endroit où créer le projet (*) (ex. `C:\Projets\P1\HelloWorld.jpr`) puis **cli**



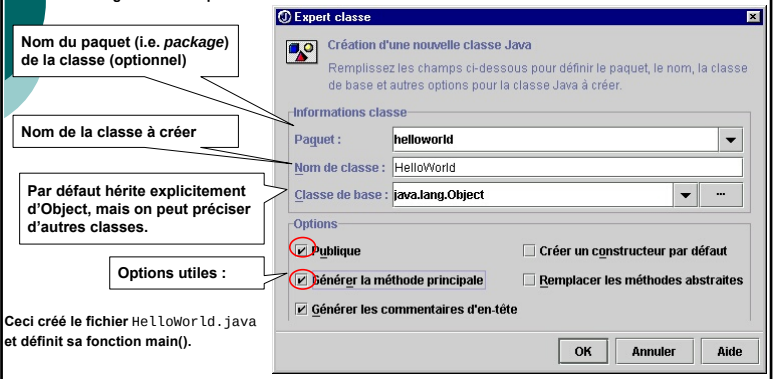
*

Premier Programme Java avec JBuilder (2/4)

2. Ajouter un fichier :

   2.1. Menu Fichier>Nouvelle Classe pour faire afficher la boîte de dialogue de l'**Expert classe**

   2.2. Renseigner les champs <Information classe>



Ceci créé le fichier `HelloWorld.java` **et définit sa fonction main().**

Premier Programme Java avec JBuilder (3/4)

3. Construire le projet :

   Dans la barre d'outils principale, cliquer sur l'icône  **ou dans le menu** Projet **cliquer sur `Construire le projet "nom_du_projet"` ou utiliser le raccourci `Ctrl+F9`.**

4. Exécuter le projet :

   4.1a. Spécifier la classe principale : **c 'est la classe dans laquelle JBuilder doit appeler la fonction** main**. Pour cela:**

   4.1.1. Accéder aux propriétés du projet : soit par le biais du menu **Projet> Propriété du Projet…** soit en cliquant avec le bouton droit de la souris sur le fichier projet (ex: HelloWorld.jpr) et cliquer sur **Propriétés…**

   4.1.2. Cliquer sur l'onglet **Exécution** et le sous-onglet **Application.**

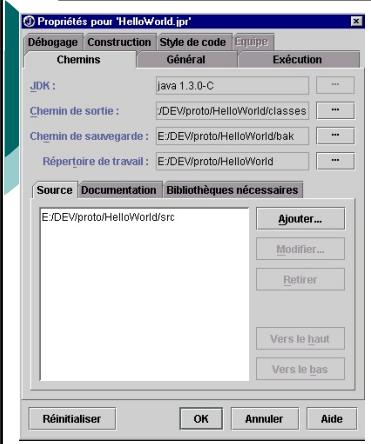
   4.1.3. Vérifier que le champ **Classe Principale** est bien renseigné. Le cas échéant modifier l 'information en cliquant sur **Définir** et cliquer sur la classe désirée (ex: HelloWorld)

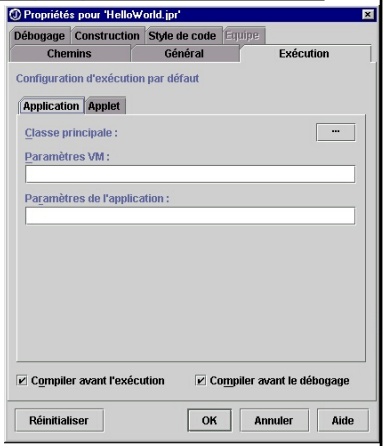
   4.1b. ou **Cliquer-Droit** sur la classe à **exécuter** et **cliquer** sur **l'icône Exécuter.**

   4.2 Dans la barre d'outils principale, cliquer sur l'icône ou dans le menu **Exécuter** cliquer sur Exécuter le projet ou utiliser le raccourci F9.

5. Le résultat s'affiche dans le panneau Message.

Premier Programme Java avec JBuilder (4/4)





Ressources Multimedia

- ***Technique de base pour le multimédia*, Gérad Weidenfeld et al., Ed. Masson 1997.**
- ***Java et le multimédia*, Jean-Marc Farinone, Edition 01 Informatique, DUNOD, 2003.**
- ***Java : Comment programmer.* Deitel & Deitel. Ed. Reynald Goulet. 2002.**
- **JavaWorld. (généraliste) : http://www.javaworld.com**

Ressources Java

- ***Thinking in Java*. B. Eckel.** `http://www.BruceEckel.com`
  ***Java de Base*, Richard Grin,** `http://http://deptinfo.unice.fr/~grin/`
  ***Site officiel de Java* :** `http://java.sun.com`
  ***Tutoriel Java* de Sun** (en anglais) **:**
  - `http://java.sun.com/docs/books/tutorial/index.html`
  - `http://java.sun.com/docs/books/tutorial/getStarted/cupojava/index.html`
- ***Spécification du langage Java*** (en anglais) **:**
  - `http://java.sun.com/docs/books/jls/index.html`
- ***Configurer java* : attention** configuration linux considère le shell sh.
  - `http://java.sun.com/j2se/1.4.1/docs/tooldocs/tools.html`
  - `http://vbeaud.free.fr/Informatique/PersoDebian/HTML/node4.html`
  - `http://cern91.tuxfamily.org/linux/indexconf.php4?page=env#ss1`
- ***E-zine Java World* :** `http://www.javaworld.com`
- ***IDE Java gratuits :***
  - ***JBuilder– Personal Edition (Borland)* :** `http://www.borland.com/jbuilder`
  - ***Sun ONE Studio - Community Edition (Sun Systems) :***
    `http://java.sun.com/j2se/1.4/download.html`
  - ***Jcreator (Light version)* :** `http://www.jcreator.com`
  - ***Eclipse* :** `http://www.eclipse.com`

* ancien Forte (ancien NetBeans)