

Cours de JAVA



Initiation à la programmation Objet en Java

Emmanuel ADAM

Institut des Sciences et Techniques de Valenciennes

Université de Valenciennes et du Hainaut-Cambrésis

source principale :
« Thinking in Java (B. Eckel) »

Introduction à JAVA

 Présentation

 Concepts

Présentation de JAVA

Petite histoire

- 🖱 1991, chez SUN est créé le langage OAK (pour interface entre appareil ménagers et ordinateurs)
- 🖱 1994, Oak se tourne vers Internet et devient Java
- 🖱 1995, Java est intégré à Netscape
- 🖱 1998, Version 2 de Java
- 🖱 2000, Version 1.3 de Java
- 🖱 2002, Version 1.4 de Java
- 🖱 2004, Version 1.5 de Java

Présentation (1/3)

- 🖱 Java est un langage récent qui s'est répandu de façon étonnamment rapide
- 🖱 Car : Java est portable, non dépendant de l'environnement d'implémentation (de la machine)
- 🖱 Car : la syntaxe de Java est très proche du langage C++ (et donc de C)...

Présentation (2/3)

- 🖱 Car : Java est robuste
 - 🖱 Gestion de la mémoire par un système de ramasse - miettes (Garbage Collector)
 - 🖱 Typage fort
- 🖱 Car : Il existe un grand nombre de classes sûres et commentées
- 🖱 Car Java est gratuit ...

Présentation (3/3)

- 🖱 Java est "vraiment" Orienté Objet :
tout est objet
- 🖱 Java fournit un code (ByteCode) interprété par une machine virtuelle
- 🖱 (+) Garantit la portabilité
- 🖱 (-) Ralentit l'exécution

Télécharger Java

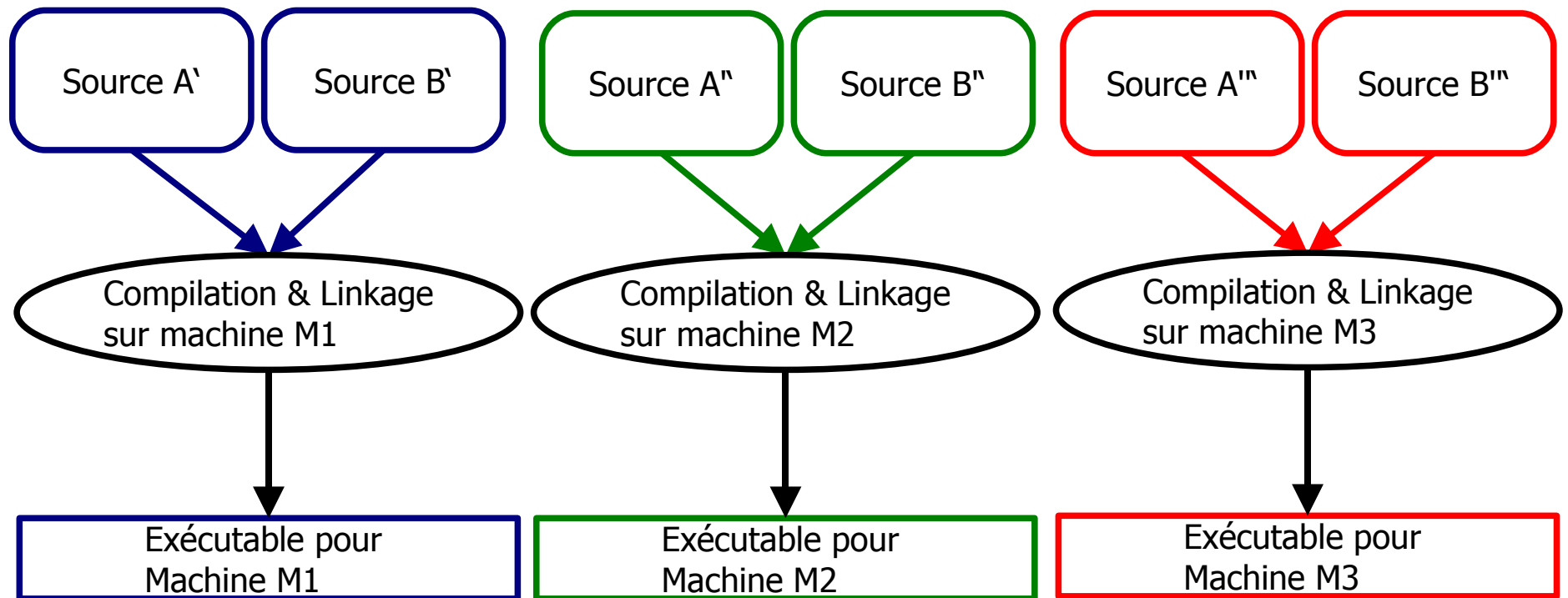
- 🖱 Sur son site (www.sun.com), SUN propose les différents Kits de Développement Java (JDK)
[JDK 1.0, JDK 1.2, JDK 1.3, JDK 1.4, JDK 1.5]
- 🖱 SUN propose également une documentation très complète
- 🖱 Sur Internet, grand nombre de sources, de projets open-sources et de documentations [Ex : "Thinking in Java" de B. Eckel]

Les concepts de JAVA

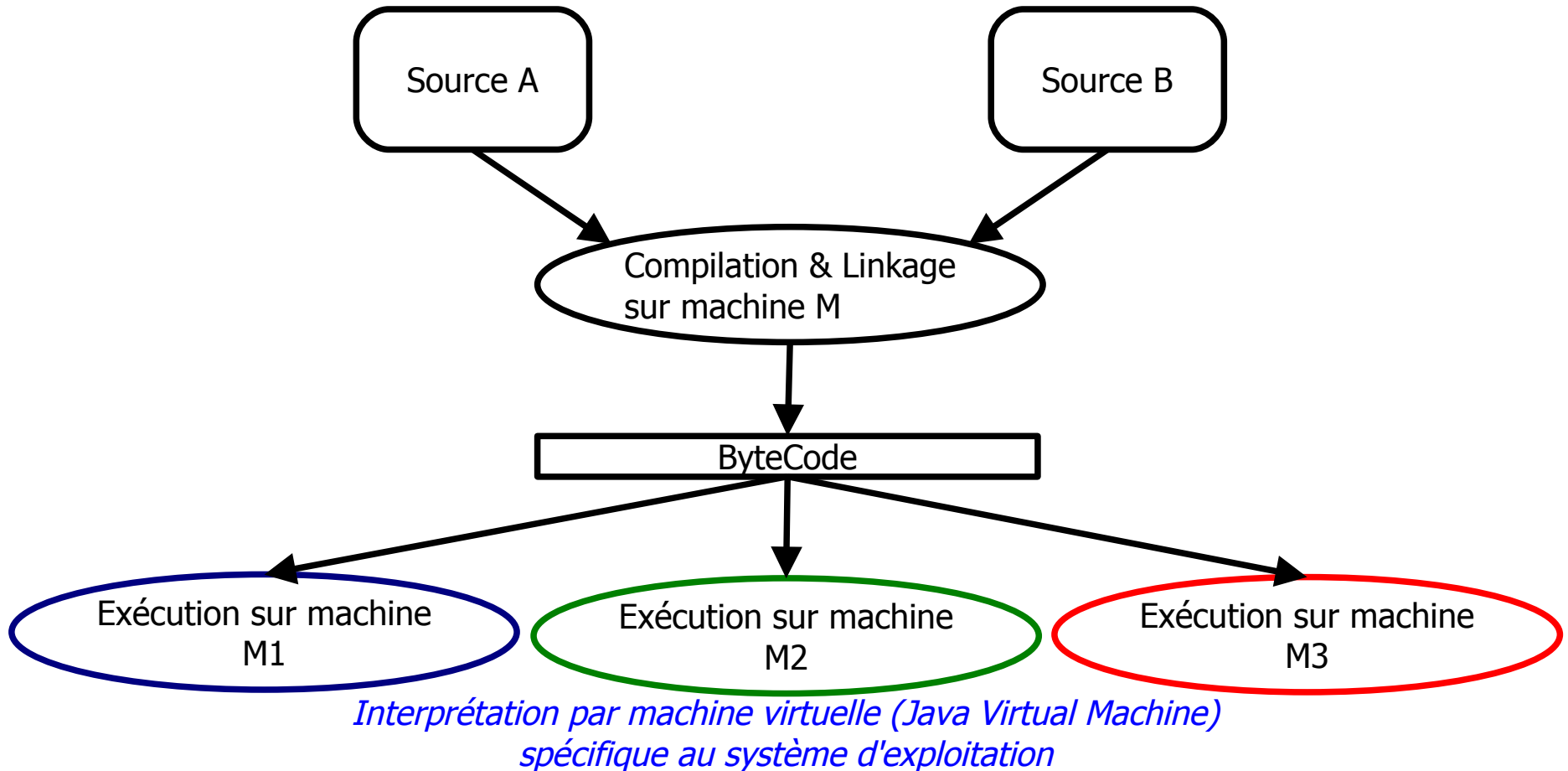
La Compilation & JAVA

- 🖱 Le principal atout de JAVA concerne sa portabilité.
- 🖱 Celle-ci est possible par la notion de ByteCode.
- 🖱 JAVA ne produit pas un code natif, directement exécutable.
- 🖱 JAVA produit un code intermédiaire, le ByteCode, interprétable par une machine virtuelle.

La Compilation "classique"



La Compilation Java



Code natif vs ByteCode

- 🖱️ (+) rapidité d'exécution
- 🖱️ (-) source pas toujours portable
- 🖱️ (-) recompiler pour changer de système

- 🖱️ (+) source portable
- 🖱️ (+) Bytecode portable
- 🖱️ (-) moins rapide car interprété

Source portable

- 🖱 Les types primitifs sont totallement spécifiés.
- 🖱 Ils ne dépendent pas de l'architecture de la machine :
 - 🖱 Les entiers, réels, etc... sont toujours définis sur le même nombre de bits.
- 🖱 Génération du ByteCode par le compilateur javac

Ramasse Miettes

- 🖱 En JAVA, tout est objet. Il faut donc créer explicitement chaque objet manipulé.
- 🖱 Cependant, il est inutile de les détruire. Le Garbage Collector (ramasse miettes) s'en charge.
- 🖱 Dès qu'un objet n'est plus utilisé (sortie de boucle, ...), il est détruit.

Définition des variables d'environnement

🖱 Pour compiler en java, il faut définir quelques variables d'environnement :

🖱 Chemin d'accès aux exécutable. Par exemple :
`SET PATH=%PATH%;C:\jdk1.4.1\bin`

🖱 Chemin d'accès aux librairies. Par exemple :
`SET CLASSPATH=.;c:\jdk1.4.1\lib`

Exemple de compilation

Hello.java

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

Source

ByteCode

Hello.class

C:\Java>javac Hello.java

```
Ëp°¼ -
()V (Ljava/lang/String;)V ([Ljava/lang/String;)V <init>
Code Hello      Hello !!!
Hello.java LineNumberTable Ljava/io/PrintStream;
SourceFile java/io/PrintStream java/lang/Object
java/lang/System main out println !
      * . ±                %      ² ¶ ±
```

C:\Java> java Hello
Hello !!!

Éléments de base de JAVA

Tout est Objet (ou presque...)

- ☞ Tous les éléments en JAVA dérivent de la classe Object.

- ☞ Tous sauf les primitives (8 en tout) représentant des types simples.

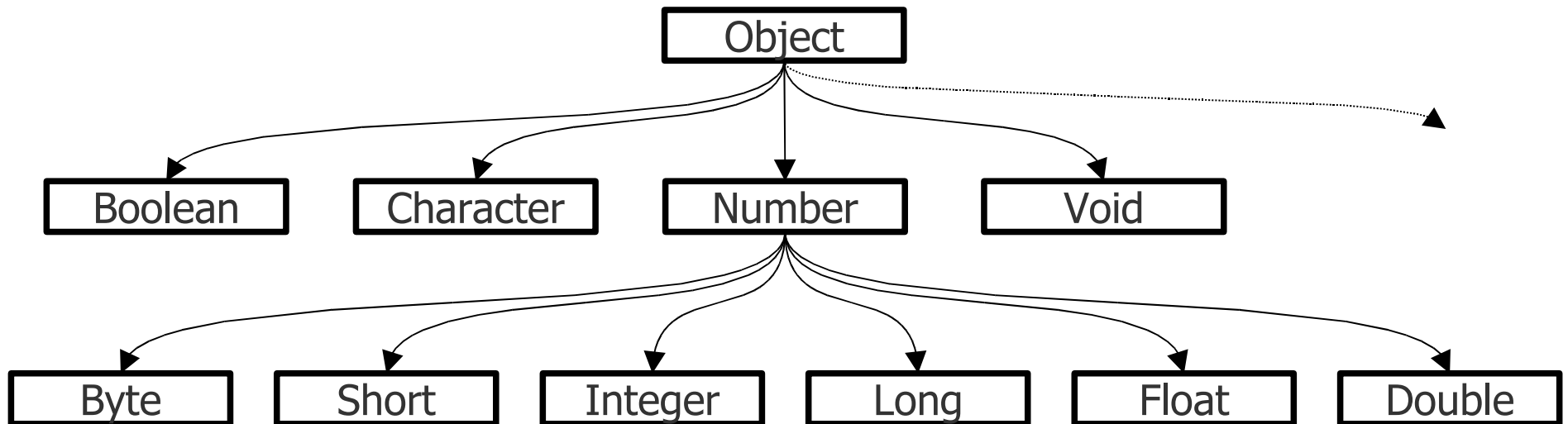
- ☞ A chaque type de primitive correspond une classe d'objet.

Les primitives : des types simples

<i>Nom</i>	<i>Taille</i>	<i>Minimum</i>	<i>Maximum</i>	<i>Classe</i>
boolean	1 bit	false	true	Boolean
char	16 bits	Unicode 0	Unicode $2^{16}-1$	Character
byte	8 bits	-128	127	Byte
short	16 bits	-2^{15}	$2^{15}-1$	Short
int	32 bits	-2^{31}	$2^{31}-1$	Integer
long	64 bits	-2^{63}	$2^{63}-1$	Long
float	32 bits	$(+/-)1.4 \cdot 10^{-45}$	$(+/-)3.4 \cdot 10^{38}$	Float
double	64 bits	$(+/-)4.9 \cdot 10^{-324}$	$(+/-)1.8 \cdot 10^{308}$	Double
void	-	-	-	Void

Les objets simples

🖱️ Tout objet Java hérite de la classe Object :



Primitives <-> Objets

- ☞ Passer d'une primitive à l'objet correspondant :

```
int i = 5;  
Integer obj = new Integer (i);
```


- ☞ Dans la classe Number sont définies les méthodes de conversion intValue, doubleValue, floatValue, ...

```
int j = obj.intValue();
```

- ☞ Dans la classe Object est définie la méthode **toString** convertissant un objet en chaîne. Cette fonction est *surchargée* dans les classes d'objets numériques.

Exemples de conversions

```
public class TestNum
{
    public static void main(String arg[])
    {
        double f = 5.2;    // f est un réel double précision
        Double objetDouble = new Double(f); // création d'un objet de la classe Double
        int i = objetDouble.intValue(); // appel à la méthode intValue
        System.out.println("" + f + " , " + objetDouble + " , " + i);
        // pour affichage, appel automatique à la méthode toString()
        // de l'objet objetDouble, ligne identique à :
        System.out.println("" + f + " , " + objetDouble.toString() + " , " + i);
    }
}
```



5.2 , 5.2 , 5

Les Tableaux

- ☞ Un tableau en Java est un type d'objet simple (il ne contient que l'attribut length).
- ☞ Tableau simple : `int tab [] = new int [10];`
- ☞ Tableau multi-dimensionnel :
`double cube[][][] = new double[3][3][3];`
- ☞ Initialisation : `int tab[] = {1, 2, 3, 4, 5};`

Les opérateurs (cf. C++)

Arithmétiques :

pour tout nombre x , on peut écrire :

$x++$; $x--$; $++x$; $--x$; $x+=2$; $x-=2$; $x*=2$; $x/=2$;
 $x\%=2$ ($x = x$ modulo 2)

Logiques : pour tout booléen x et y ,

non $x \Leftrightarrow !x$

x et $y \Leftrightarrow x \& y$ ou $x \&\& y$ (*test économe*)

x ou $y \Leftrightarrow x | y$ ou $x || y$ (*test économe*)

x ou exclusif $y \Leftrightarrow x \wedge y$

Structures de contrôle

🖱 Les structures de contrôle utilisées en Java sont fortement inspirées de celles du langage C ou C++ :

🖱 if ; else ; while ; do ; for ; break ; goto

Branchement conditionnel (1/2)


 Si ... Alors ... Sinon

if (test)

```
{  
    action1_si_test_vrai;  
    action2_si_test_vrai;  
}
```

else

```
{  
    action1_si_test_faux;  
    action2_si_test_faux;  
}
```

 Le test est de la forme :


a ; !a ; a & b ; a & !b ;
x < y ; x == y ; x != y ; ...


Branchement conditionnel (2/2)

 Branchement selon un choix multiple :


```
switch (expression)
{
    case valeur_1 : actions; break;
    case valeur_2 : actions; break;
    case valeur_3 : actions; break;
    default : actions si valeur inconnue;
}
```

Boucle "Tant que"

 **while** (test)
{
 actions;
}

 **do**
{
 actions;
}
while (test);

Boucle "Pour"

```
 int i;  
  for(i=0; i<5; i++) // ou for(int i=0; i<5; i++)  
  {  
    actions;  
  }
```

Sorties de boucles

- 🖱 L'instruction **break** permet de sortir d'une boucle avant sa fin "normale".

- 🖱 L'instruction **continue** passe le reste de la boucle, mais n'en sort pas.

- 🖱 L'instruction **goto label** subsiste encore. Le label doit alors être défini (et suivi de **:**).

Exemple de programme

- 🖱 Affichage de *'Hello !!!'*
- 🖱 Ce programme contient une classe principale publique Hello.
- 🖱 Il doit donc se nommer Hello.java

```
🖱 public class Hello
{
    public static void main(String args[])
    {
        System.out.println("Hello !!! ");
    }
}
```


Programmation Objet en JAVA

Concept Objet : rappels

- 🖱️ Tout **objet** dérive d'une **classe** (ou en instancie une).
- 🖱️ Une **classe** est composée de **membres**
- 🖱️ Les **membres** sont :
 - 🖱️ **des attributs (champs)**, définissant l'état d'un objet,
 - 🖱️ **des méthodes** (fonctions, procédures) agissant sur ou en fonction des attributs,
 - 🖱️ un ou des **constructeurs**, initialisant l'objet.

Exemple de classe

La classe suivante représente une personne:

```
class Personne
```

Nom de la classe commençant par une majuscule

```
{
```

```
    String nom, prenom;  
    int age;
```

Attributs

```
    Personne()
```

constructeur

```
{
```

```
    nom = new String("");  
    prenom = new String("");  
    age = 18;
```

méthode

```
}
```

```
void affichePersonne()
```

```
{
```

```
    System.out.println(prenom + " " + nom + ", age : " + age);
```

```
}
```

```
}
```

Exemple d'utilisation d'une classe

```
class Personne
```

```
{...}
```

```
public class TestPersonne
```

```
{
```

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

```
    {
```

```
        Personne p1 = null;
```

```
        Personne p2 = new Personne();
```

```
        System.out.println(p1);
```

```
        System.out.println(p2);
```

```
        p2.affichePersonne();
```

```
        p1.affichePersonne();
```

```
    }
```

```
}
```

```
C:\> javac TestPersonne.java
```

```
C:\> dir *.class  
TestPersonne.class  
Personne.class
```

```
C:\> java TestPersonne
```

```
null
```

```
Personne@ea2dfe
```

```
, age : 18
```

```
Exception in thread "main" java.lang.NullPointerException  
at TestPersonne.main(TestPersonne.java:26)
```

Utilisation des Objets en Java

🖱 En Java, on manipule des **références** aux objets,

// construction d'un objet Personne, f1 reçoit son adresse en mémoire virtuelle

```
Personne f1 = new Personne();
```

```
f1.nom = "Simon";
```

// f2 reçoit la valeur de f1, et donc pointe vers le même objet

```
Personne f2 = f1;
```

```
System.out.println(f1.nom + "-" + f2.nom);
```

*// → **Simon – Simon***

```
f2.nom = "hello";
```

```
System.out.println(f1.nom + "-" + f2.nom);
```

*// → **hello – hello***

Objets anonymes



Il n'est pas nécessaire de donner un nom à un objet:

```
public class TestPersonne  
{
```

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

```
    {
```

```
        Personne [] tableau = new Personne[5];
```

```
        for (int i=0; i<5; i++)
```

```
        {
```

```
            tableau[i] = new Personne ();
```

```
            tableau[i].age += i;
```

```
        }
```

```
        for (int i=0; i<5; i++)        tableau[i].affichePersonne();
```

```
    }
```

```
}
```

```
, age : 18  
, age : 19  
, age : 20  
, age : 21  
, age : 22
```

Héritage

🖱 En java,
Si une classe A hérite d'une classe B,
elle possède tous les attributs et méthodes de
B **non privés**

🖱 Notation :
`class Animal {...}`
`class Hamster extends Animal {...}`

🖱 **P a s d ' h é r i t a g e m u l t i p l e**

Héritage : **généricité et surcharge** (1/4)

🖱 Une classe fille peut surcharger une méthode d'une classe mère

🖱 L'objectif de l'héritage est la **généricité** :
appliquer le même schéma d'actions à des
objets de type différents

Héritage : généricité et surcharge (2/4)

```
class Animal
{
    String nom;
    int nb_pattes;
    Animal()
    {
        nom = new String("");
        nb_pattes = 0;
    }
    void affiche()
    {
        System.out.println("Animal : " + nom + ",
                           nb de pattes = " + nb_pattes);
    }
}
```

```
class Hamster extends Animal
{
    String couleur_pelage;
    Hamster()
    {
        nb_pattes = 4;
        couleur_pelage = "gris";
    }
    void affiche()
    {
        System.out.println("Hamster : " + nom + ",
                           nb de pattes = " + nb_pattes + ",
                           couleur = " + couleur_pelage);
    }
}
```

```
public class TestAnimaux
{
    public static void main(String args[])
    {
        Animal a = new Animal();
        Hamster h = new Hamster();
        a.nom = "ani";
        h.nom = "ham";
        a.affiche();
        h.affiche();

        Animal tab[] = {a, h};
        tab[0].affiche();
        tab[1].affiche();
    }
}
```

```
Animal : ani, nb de pattes = 0
Hamster : ham, nb de pattes = 4, couleur = gris
Animal : ani, nb de pattes = 0
Hamster : ham, nb de pattes = 4, couleur = gris
```

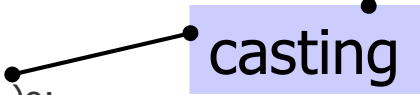
Héritage : **généricité et surcharge** (3/4)

- 🖱 Généricité = utiliser le même code sur tous types d'objet
- 🖱 Les objets Java héritent tous de la classe Object
- 🖱 => utiliser le type Object et le "**casting**"
- 🖱 Rq. : **instanceof** permet de retrouver la classe d'origine d'un objet

Héritage : généricité et surcharge (4/4)

Exemple : un tableau de 4 objets de 2 types différents

```
Integer i1 = new Integer(11); Integer i2 = new Integer(12);
Personne p1 = new Personne(); Personne p2 = new Personne();
p1.nom = "Nicolas"; p2.nom = "Jose";
Object tab [] = {i1, p1, i2, p2};
for(int i=0; i<4; i++)
{
    Object o = tab[i];
    if (o instanceof Integer)
        System.out.println("Integer, val = " + (Integer)o );
    else if (o instanceof Personne)
    {
        Personne f = (Personne )o;
        System.out.println(" Personne, nom = " + f.nom );
    }
    else System.out.println("Objet indéterminé" );
}
```



Comparer les objets en Java (1/2)

🖱 Le test **==** ne compare que les valeurs des références

```
Personne p1 = new Personne(); Personne p2 = new Personne();  
p1.nom = "Samia" ; p2.nom = "Samia" ;  
System.out.println(p1 == p2);           // false  
System.out.println(p1 == p1);           // true
```

🖱 Solution : surcharger la méthode **equals** de la super class **Object**

Comparer les objets en Java (2/2)

Il faut comparer chaque champs de la classe

```
class Personne // extends Object par défaut
```

```
{....
```

```
    public boolean equals(Personne f)
```

```
    {
```

```
        boolean rep;
```

```
        rep = nom.equals(f.nom);
```

```
        return rep;
```

```
    }
```

```
}
```

```
public class TestEgalite
```

```
{
```

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

```
    {
```

```
        Personne p1 = new Personne(); Personne p2 = new Personne();
```

```
        p1.nom = "Samia" ; p2.nom = "Samia" ;
```

```
        System.out.println(p1 == p2);
```

```
        System.out.println(p1.equals(p2));
```

```
    }
```

```
}
```



false
true

Protection des attributs et méthodes

- 🖱 Un attribut ou une méthode précédé de
 - 🖱 **private** n'est accessible que par sa classe,
 - 🖱 **package** est accessible aux classes voisines,
 - 🖱 **protected** est accessible aux sous-classes et aux classes voisines,
 - 🖱 **public** est accessible à toutes classes.
 - 🖱 *private protected* n'est plus utilisé,
- 🖱 Par défaut : package.

Exemple de protection

La classe suivante représente une personne:

class **PersonneP**

```
{
    public String nom, prenom; // accessible par toutes les classes
    private int noSecu;        // accessible par cette classe uniquement
    int noTel; //accessible par cette classe et classes voisines
    int age; //accessible par cette classe et classes voisines
    Personne()
    {
        nom = new String("");
        prenom = new String("");
        age = 18; noSecu = 0; noTel = 0;
    }
    public void affichePersonne() // toutes les classes peuvent appeler cette méthode
    {
        System.out.println(prenom + " " + nom + ", age : " + age);
    }
}
```

Exemple d'utilisation d'attributs protégés

```
class PersonneP
```

```
{...}
```

```
public class TestPersonneP
```

```
{
```

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

```
    {
```

```
        PersonneP p = new PersonneP();
```

```
        System.out.println(p);
```

```
        p.affichePersonne();
```

```
        p.noSecu = 5;
```

```
    }
```

```
}
```



```
>javac TestPersonne.java
```

```
TestPersonne.java:26: noSecu has private access in Personne
```

```
    p.noSecu = 5;
```

```
        ^
```

```
1 error
```


Membres particuliers : Constructeurs

- 🖱 Un constructeur est appelé automatiquement à la création de l'objet
- 🖱 Le constructeur permet d'initialiser un objet
- 🖱 Le constructeur d'une classe porte le nom de cette classe. Exemple :

```
class Fenetre
{
    String nom;
    Fenetre () {nom = "";}           // constructeur par défaut
    //Fenetre (String nom) {this.nom = nom;}
    Fenetre (String _nom) {nom = _nom;}
}
```

Membres particuliers : This

☞ Le champ **this** d'une classe se réfère à l'instance de l'objet en cours de traitement.

☞ Il peut être utilisé :

- Dans un constructeur,
- Pour passer une référence à l'objet

```
- class Fenetre
{
    ObjetGraphique og = null; ...
    public void affiche()
    {
        og.afficheGraphique( this );
    }...
}
```

- Pour retourner une référence à l'objet

```
- class Fenetre
{
    ....
    Fenetre changeNom(String nom2)
    {
        nom = nom2;
        return this;
    }
}
```

Héritage et constructeurs

☞ Si la classe A hérite de la classe B, le constructeur de A fait automatiquement appel au constructeur par défaut de B.

☞ **super** permet de choisir le constructeur de la classe mère :

```
class B
{
    ...
    B(){...}
    B(int i){...}
}
```

```
class A extends B
{
    ...
    A() {...}
    A(int i, int j) {super(i); ...}
    A(int i, int j, int k) {this(i,j); ...}
}
```

Destructeurs ?

☞ Java gère lui même les objets

☞ Mais, au besoin, la méthode `finalize()` peut être surchargée

☞ Et le ramasse miettes peut être lancé `System.gc()`
uniquement à partir d'une méthode statique !!!

Autres mots-clés : final

 **final** définit :

 un attribut invariable,
dont la valeur ne peut être modifiée que par le constructeur
final int noSecu;

 une méthode qui ne peut être surchargée,
final void affichePersonne(){...}

 une classe qui ne peut être héritée.
final class Personne {...}

Autres mots-clés : **static** (1/2)

- 🖱 **static** définit un membre commun à toutes les instances de la classe
- 🖱 On parle de membre de classe par opposition à membre d'instance
- 🖱 Une variable de classe est indépendante de l'objet, elle existe dès le premier appel à la classe.

Autres mots-clés : static (2/2)

- ❏ Seule une méthode de classe peut utiliser des variables de classe

- ❏ Une méthode de classe ne peut être surchargée dans les sous-classes

- ❏ Utilisation des membres de classe :
 - ❏ compteur d'objet,
 - ❏ optimisation (définition de constante),
 - ❏ définition de librairies, ...

Utilisation de static (1/4)

```
class Personne
{
    String nom, prenom;
    int age;
    static int nbPersonnes = 0;           // définition de la variable statique
    final int noPersonne;
    private int noSecu;

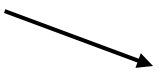
    Personne()                          // constructeur par défaut
    {
        nom = new String("");;         prenom = new String("");;
        age = 18;                       noSecu = 0;      nbPersonnes ++;      noPersonne = nbPersonnes;
    }

    Personne(String n, String prenom)   // constructeur avec paramètres
    {
        nom = n;
        this.prenom = prenom;
        age = 18;                       noSecu = 0;      nbPersonnes ++;      noPersonne = nbPersonnes;
    }

    public String toString()             // surcharge de la méthode héritée de Object
    {
        return (prenom + " " + nom + ", age : " + age +
                ", noSecu = " + noSecu + ", no personne = " + nbPersonnes );
    }
}
```


Utilisation de static (2/4)

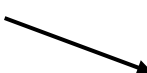
```
public class TestPersonne
{
    public static void main(String args[])
    {
        Personne p1 = new Personne("aa", "a"); // Personne.nb_personne ← 1
        System.out.println(p1);
        Personne p2 = new Personne("bb", "b"); // Personne.nb_personne ← 2
        System.out.println(p2);
        Personne p3 = new Personne("cc", "c"); // Personne.nb_personne ← 3
        System.out.println(p3);
        Personne p4 = new Personne("dd", "d"); // Personne.nb_personne ← 4
        System.out.println(p4);
        Personne p5 = new Personne("ee", "e"); // Personne.nb_personne ← 5
        System.out.println(p5);
    }
}
```



```
a aa, age : 18, no_secu = 0, nb personne = 1
b bb, age : 18, no_secu = 0, nb personne = 2
c cc, age : 18, no_secu = 0, nb personne = 3
d dd, age : 18, no_secu = 0, nb personne = 4
e ee, age : 18, no_secu = 0, nb personne = 5
```

Utilisation de static (3/4)

```
public class TestPersonne
{
    public static void main(String args[])
    {
        Personne p1 = new Personne("aa", "a"); // Personne.nb_personne ← 1
        Personne p2 = new Personne("bb", "b"); // Personne.nb_personne ← 2
        Personne p3 = new Personne("cc", "c"); // Personne.nb_personne ← 3
        Personne p4 = new Personne("dd", "d"); // Personne.nb_personne ← 4
        Personne p5 = new Personne("ee", "e"); // Personne.nb_personne ← 5
        System.out.println(p1);
        System.out.println(p2);
        System.out.println(p3);
        System.out.println(p4);
        System.out.println(p5);
    }
}
```



```
a aa, age : 18, no_secu = 0, nb personne = 5
b bb, age : 18, no_secu = 0, nb personne = 5
c cc, age : 18, no_secu = 0, nb personne = 5
d dd, age : 18, no_secu = 0, nb personne = 5
e ee, age : 18, no_secu = 0, nb personne = 5
```

Utilisation de static (4/4)

```
class Personne
{
    ...
    static int retourneNbPersonnes()
    {
        return (no_personne);
    }
}
```

nombre d'objets Personne crees = 2

```
public class TestPersonne
{
    public static void main(String args[])
    {
        Personne p1 = new Personne("aa", "a"); // nb_personne ← 1
        Personne p2 = new Personne("bb", "b"); // nb_personne ← 2

        int nb_personnes = Personne.retourneNbPersonnes();

        System.out.println("nombre d'objets Personne crees = " + nb_personnes);
    }
}
```

Autres mots-clés : **abstract** (1/2)

- 🖱 **abstract** associé à une méthode permet de ne pas lui donner de corps.
- 🖱 Si une méthode est abstraite, sa classe doit être définie comme abstraite.
- 🖱 Une classe abstraite ne peut être instanciée.
- 🖱 L'abstraction : définir un moule général pour les classes dérivées.

Autres mots-clés : abstract (2/2)

```
abstract class Animal
```

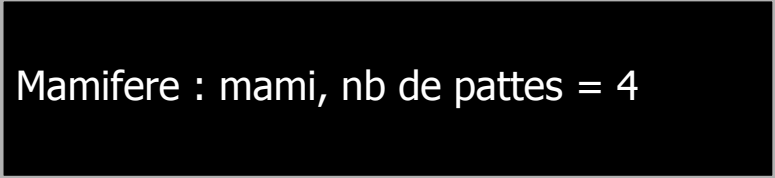
```
{  
    String nom;  
    int nb_pattes;  
    Animal()  
    {  
        nom = new String("");  
        nb_pattes = 0;  
    }  
    abstract void affiche();  
}
```

```
class Mammifere extends Animal
```

```
{  
    Mammifere()  
    {  
        nb_pattes = 4;  
    }  
    void affiche()  
    {  
        System.out.println("Mammifere : " + nom +  
                             ", nb de pattes = " + nb_pattes);  
    }  
}
```

```
public class TestAnimaux
```

```
{  
    public static void main(String args[])  
    {  
        Mammifere m = new Mammifere();  
        m.nom = "mami";  
        m.affiche();  
    }  
      
    // Animal a = new Animal(); n'est pas possible !!!  
}
```



Mammifere : mami, nb de pattes = 4

Interface : une "*classe vide*"

☞ Une **interface** est une classe :

☞ ne contenant que les entêtes de méthodes (*public* par défaut)

☞ ne contenant que des attributs constants (de type *static* et *final*).

☞ Notation :

interface EtreVivant

```
{  
    String type = "etre vivant"; // obligation d'initialiser l'attribut  
    void naitre();  
}
```

class Animal **implements** EtreVivant

```
{  
    ...  
}
```

Implémentations multiples

- ☞ Pas d'héritage multiple en java, mais possibilité d'implémentation multiple:
- ☞ Notation :

```
interface EtreVivant    { ... }
```

```
interface EtreMortel extends EtreVivant  
{  
    void mourir();  
}
```

```
interface EtreSpirituel  
{  
    void philosopher();  
}
```

```
class Animal implements Etre_vivant, EtreMortel    { ... }
```

```
class EtreHumain extends Animal implements EtreSpirituel { ... }
```

Interface pour regrouper les constantes

- 🖱 Les attributs des interfaces sont statiques et finaux par défaut
 - 🖱 ➔ possibilité de créer des groupes de constantes

```
interface Couleur
{
    int    BLANC = 999,    NOIR   = 000,    GRIS   = 555,
          BLEU   = 900,    ROUGE  = 090,    VERT   = 009;
}

public class TestCouleur
{
    public static void main(String arg[])
    {
        int couleur_a = Couleur.BLEU;
        System.out.println(couleur_a);
    }
}
```


Classes internes

- 🖱 Il est possible d'imbriquer
 - 🖱 des classes et interfaces dans des classes
 - 🖱 des interfaces dans des interfaces
- 🖱 Pour grouper les classes logiquement rattachées

Classes internes : exemple 1/3

```
class Formation
{
    String nom;
    String type;
    Etudiant [] tabEtu = new Etudiant[10];
    private int index = 0;

    Formation() { nom = ""; type = ""; }
    Formation (String nom, String type) { this.nom = nom; this.type = type; }

    int nbEtudiants() { return index; }

    void ajouteEtu(String nom, String prenom)
    {
        Etudiant e = new Etudiant(nom, prenom);
        e.ajoute();
    }

    public String toString()
    {
        String chaine = "";
        for(int i=0; i<index; i++)
            chaine = chaine + tabEtu[i] + "\n";
        return chaine;
    }
    ...
}
```

Classes internes : exemple 2/3

```
....
class Etudiant
{
    String nom, prenom;

    Etudiant() {nom = ""; prenom = "";}
    Etudiant(String nom, String prenom) {this.nom = nom; this.prenom = prenom;}

    void ajoute()
    {
        if (index<10)
        {
            tabEtu[index] = this; // la classe interne a accès à tous les éléments
            index++;              // de la classe conteneur, même privés
        }
    }

    public String toString()
    {
        return (nom + " " + prenom);
    }
}
}
```

Classes internes : exemple 3/3


```
public class TestFormation
{
    public static void main(String args[])
    {
        Formation f = new Formation("mater2 ichm", "bac+5");

        f.ajouteEtu("alain", "zouave");
        f.ajouteEtu("ben", "volta");

        System.out.println(f);

        Formation.Etudiant e = f.new Etudiant("cloe", "xerty");
        System.out.println(e);
    }
}
```

```
%javac TestFormation.java
%ls *.class
Formation.class Formation$Etudiant.class
TestFormation.class
```



```
alain zouave
ben volta
cloe xerty
```

Classes anonymes

- 🖱 Il est possible de créer des classes anonymes
 - 🖱 héritant d'une classe ou implémentant une interface
 - 🖱 Pour un usage limité à 1 objet, ...

```
interface Professeur                                // soit une interface Professeur
{
    String getName ();
}

...
Professeur prof = new Professeur()                // création dynamique d'une classe
{                                                       // implémentant Professeur
    private String nom = "Adam";                       // et création de l'objet prof de
    public String getName () { return nom; }             // cette classe anonyme.
};

System.out.println(prof.getName());                  // utilisation classique de l'objet
```

Classes anonymes : Exemple

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

class MaFenetre extends Frame
{
    MaFenetre()
    {
        this.setTitle("exemple de classe anonyme");
        this.setBounds(10,10,400,50);
    }
    public void ajoutGestionFenetre()
    {
```

MaGestionFenetre mgf = new MaGestionFenetre();

```
class MaGestionFenetre extends WindowAdapter
{
    public void windowClosing(WindowEvent evt)
    {
        System.out.println("Fermeture de la fenetre");
        System.exit(0);
    }
}
}
}
```

```
public class TestFenetre
{
    public static void main(String arg[])
    {
        MaFenetre mf = new MaFenetre();
        mf.ajoutGestionFenetre();
        mf.show();
    }
}
```



Création dynamique d'instance

- 🖱 Il est possible de créer une nouvelle instance d'une classe existante hors du projet (ni dans le fichier, package courant, ni dans les import)
- 🖱 La classe **Class** représente une classe java existante.
- 🖱 Elle permet de créer dynamiquement des nouvelles instances

```
Class c = Class.forName("NomDeLaClasse");
```

Création dynamique d'instance: exemple

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

public class TestClass
{
    public static void main(String args[])
    {
        try
        {
            Class c = Class.forName("Personne");           //recherche de la classe Personne
            Class typeParamCons[] = {String.class, String.class}; // recherche du constructeur
            avec
            Constructor ctor = c.getDeclaredConstructor(typeparamCons); // 2 paramètres String
            Object paramCons[] = {"benoit", "zed"};           // appel du constructeur avec
            Object personne = ctor.newInstance(paramCons);     // 2 paramètres
            Method m = c.getMethod("affiche", null);          // recherche de la méthode
            affiche
            m.invoke(personne, null);                          // appel de cette méthode
        }
        catch(Exception e)
        {System.out.println("Erreur de recherche de classe : " + e);}
    }
}
```


De la pratique !!!

- ☞ Une fenêtre est caractérisée par :
 - ☞ son nom, ses coordonnées, sa taille
 - ☞ Deux méthodes :
 - s'afficher(), se déplacer(dir, longueur)
- ☞ Toutes les fenêtres possèdent la représentation du déplacement :
 - 1 -> haut, 2 -> bas, 3 -> gauche, 4 -> droite
- ☞ Ecrire la classe fenêtre
- ☞ Ecrire la classe permettant de tester la classe fenêtre

De la pratique (suite)!!!

- 🖱️ Proposer 4 constructeurs :
 - 🖱️ Sans argument,
 - 🖱️ Avec 1 argument (le nom),
 - 🖱️ Avec 4 arguments (nom, position et taille),
 - 🖱️ Avec 2 arguments (une fenêtre et un nom),
la nouvelle fenêtre prend les caractéristiques de la
fenêtre passée en paramètre.