Programmation Orientée Objet avec JAVA

Plan

- Introduction
- Le langage JAVA
- La programmation objet
- Les bases du langage
- Les classes et les objets
- L'héritage et le polymorphisme
- La gestion des exceptions
- Le graphisme

Chapitre 1 – Le langage JAVA

JAVA

- Le terme JAVA est souvent employé pour désigner :
  - un langage de programmation
  - une machine virtuelle, la JVM
  - un ensemble d'**API** (*Application Programmer Interface*)
  - un ensemble d'outils

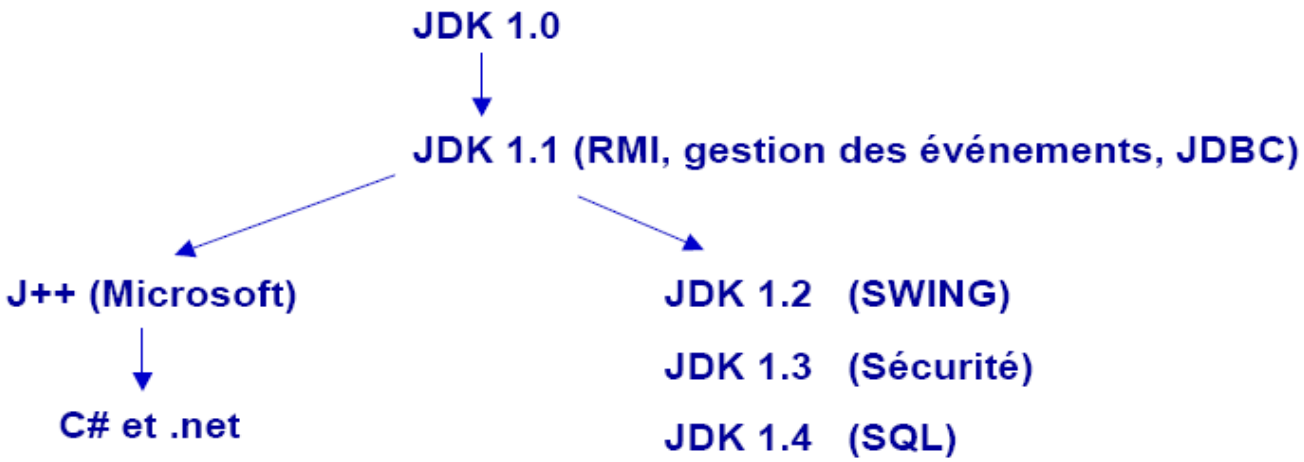
- JAVA est décrit par SUN (racheté par ORACLE en 2009) comme :
  - orienté objet – simple
  - portable – interprété
  - distribué – *multi-threads*
  - robuste – sécurisé

Historique

- Début des années 90
- Projet d'environnement indépendant du Hardware
- James Grosling (SUN) développe Oak ⟶ échec
- Montée en puissance de l'Internet
- Oak devient Java en 1995
- Succès énorme (Applet)

Evolution de JAVA

Le JDK : Société JavaSoft (branche Java de Sun)



Le JDK 1.2.1 a été officiellement renommé Java 2

Actuellement JDK 1.8 en Java 1.8
Voir la présentation sur Wikipedia
https://fr.wikipedia.org/wiki/Java_(langage)

Différentes éditions

Trois éditions différentes de JAVA :

- Java SE : Standard Edition
  - Environnement de base :
    - compilateur Java le JDK
    - API standard (java.lang, java.io, Swing, Thread, JDBC, RMI …
  - Environnement d'exécution
    - JVM
    - JRE Java Runtime Environnement

- Java EE : Enterprise Edition, destinée à l'écriture de programmes pour serveurs d'applications : Servlet, JSP, JSF, Javamail, EJB

- Java ME : Micro Edition, destinée à l'écriture d'applications embarquées.

Un langage simple

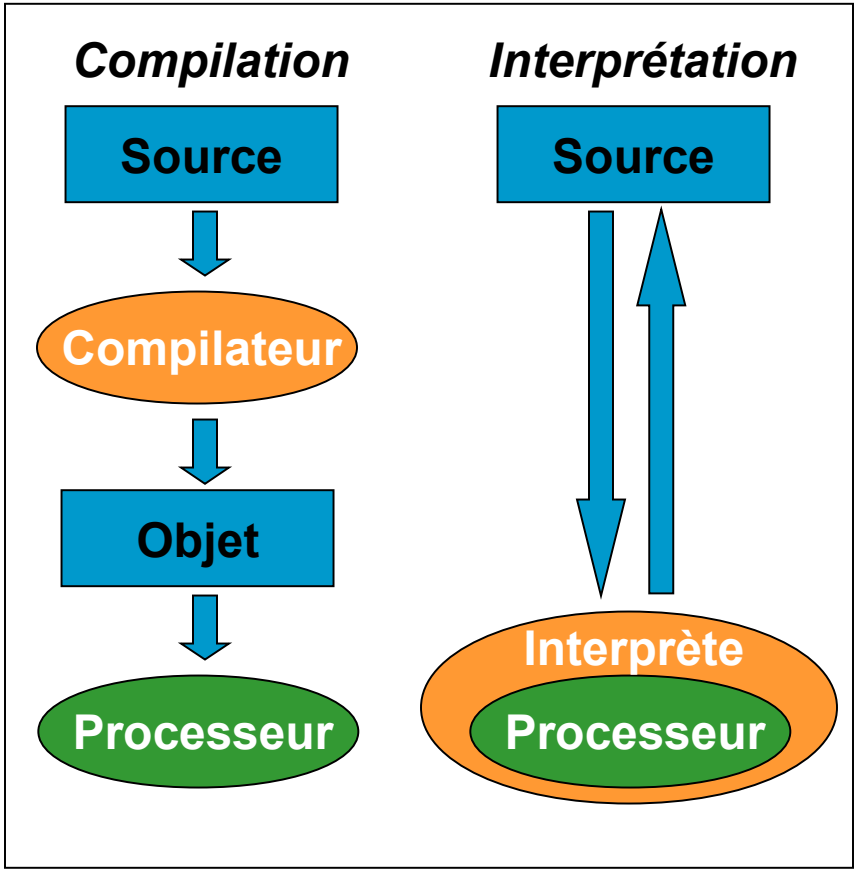
- **Simplicité**
  - syntaxe restreinte **proche** du C, du C++
  - pas de `struct` mais des `class`
  - **pas de pointeur**
  - pas de problème de gestion mémoire
  - API de plus de **2000 classes prédéfinies**

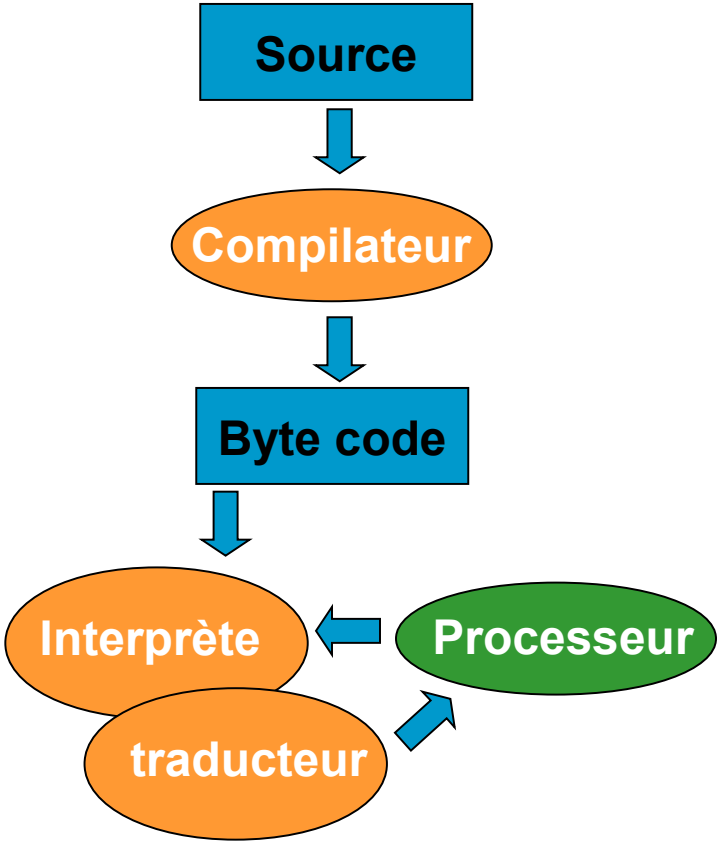
API	Fonctionnalités
java.lang, java.io	Types de données et entrées-sorties
java.awt	Bases du graphisme et du multi fenêtrage
java.swing	Graphisme 2D et IHM
Java 3D	Graphisme 3D s'appuyant sur les bibliothèques OpenGL et DirectX
JDBC	Accès aux bases de données relationnelles
Thread	Gestion du multi-threading
Servlet / JSP	Aspect Web, génération de pages HTML
RMI	Appels de fonctions à travers le réseau
EJB	Objets « métier » (nécessite un serveur d'application J2EE)
...	

Un langage portable

- **Portabilité**
  - source JAVA compilé en code intermédiaire (*byte code*)
  - *byte code* interprété par la machine virtuelle
  - *byte code* identique quelque soit l'architecture
  - taille des types primitifs indépendante de la plate forme
  - machine virtuelle présente sur UNIX, Microsoft Windows Macintosh, SUN, HP, ...

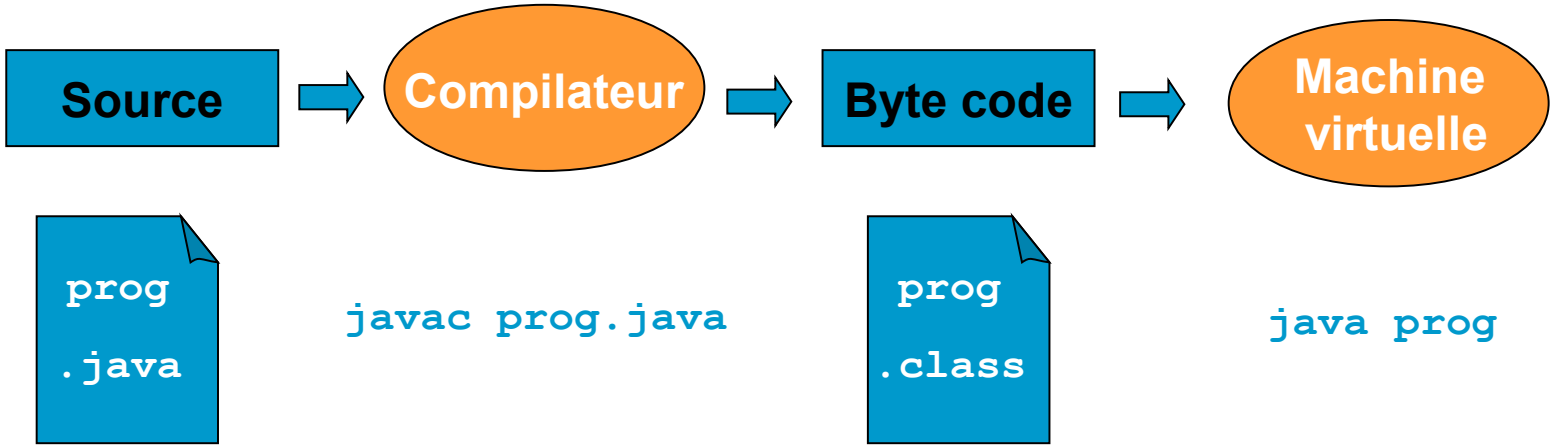
Principe du *byte-coding*





Compilation

- **Création du *byte-code***



Intérêts de JAVA

- **Multi-threads et distribué**
  - Objets locaux ou distants manipulés de la même manière
  - Multitâche et synchronisation incorporé au langage
  - Nombreuses classes prédéfinies.

Intérêts de JAVA

- **Robustesse**
  - pas d'accès direct à la mémoire
  - langage fortement typé
  - mécanisme d'exception pour les erreurs courantes
  - compilateur très contraignant.

Intérêts de JAVA

- **Sécurité**
  - prise en charge dans l'interpréteur
  - nombreux contrôles lors de l'exécution
    - ✓ ***class loader*** **vérification des classes**
    - ✓ ***verifier*** **vérification du *byte code* des méthodes**
    - ✓ ***security manager*** **vérification de l'accès aux ressources.**

Loader & verifier

- ***class loader***
  - **classes chargées dynamiquement en fonction des besoins**
  - **une classe est légale si :**
    - ✓ les formats de données sont corrects
    - ✓ les méthodes se comportent bien
- ***byte code verifier***
  - **méthodes recherchées dynamiquement en fonction des besoins**
  - ***byte code* contrôlé avant son exécution**
  - **un code est légal s'il :**
    - ✓ ne modifie pas les pointeurs
    - ✓ ne viole pas les droits d'accès
    - ✓ ne tente pas d'altérer les objets

Intérêts de JAVA

- **Performances :**
  - **AVANT :** Java sacrifie la performance au profit de la portabilité
    - ✓ **le *byte code* n'est pas conçu pour être efficace sur une plateforme particulière**
    - ✓ **la JVM est un interpréteur**
    - ✓ **elle est responsable de nombreuses taches à l'exécution.**
  - **MAINTENANT :** Elles sont améliorées par la production de code natif.
    - ✓ **les compilateurs *Just In Time* (compilation lors du premier appel)**
    - ✓ **Compilation statique : GNU compiler for Java**
    - ✓ **Compilation dynamique : compilation sélective du code nécessaire selon le contexte d'exécution**

Chapitre 2 – La programmation objet

Rappel

- **Rappel : la programmation structurée s'appuie sur :**

Programmes = Structures de données + Algorithmes

Equation de Wirth *(père des langages Modula2, Pascal, Oberon)*
« Les programmes ralentissent plus vite que le matériel accélère »
(corollaire de la loi de Moore)

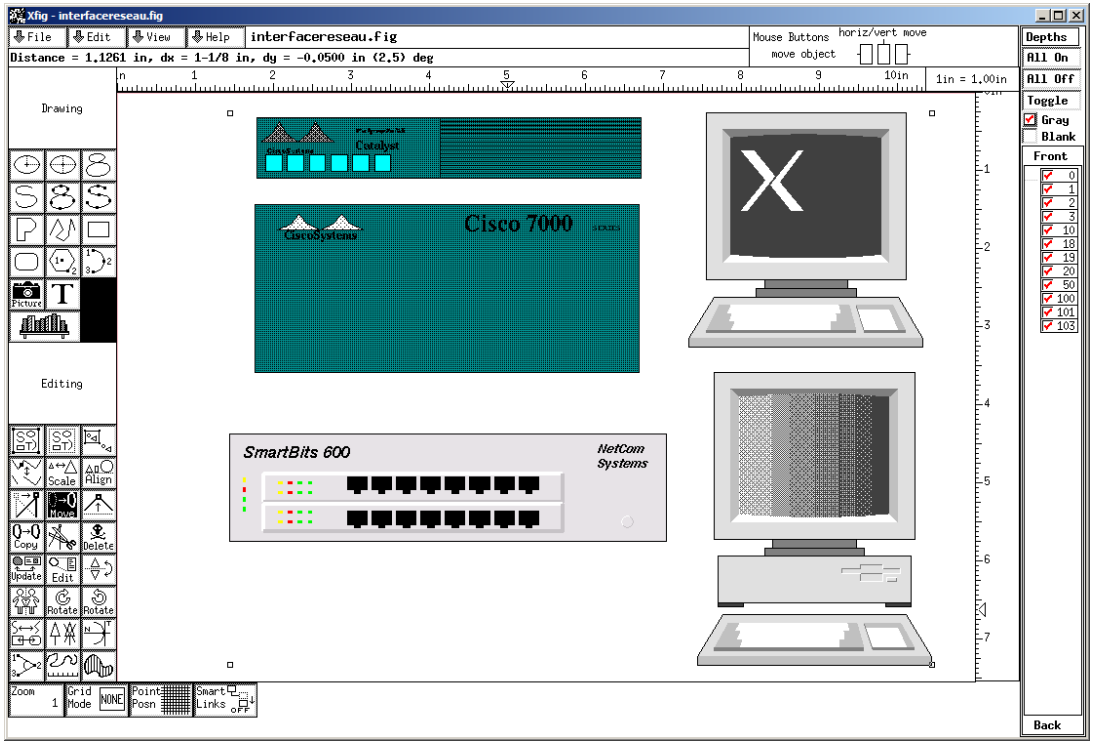
- **Elle est calquée sur l'analyse descendante : décompositions successives *(démarche top-down*)**
  cycle en V

- **A chaque étape (module) il faut se demander :**
  - **Comment représenter l'information (Structures de données) ?**
  - **Comment résoudre le problème sur ces données (Algorithme) ?**

Un exemple

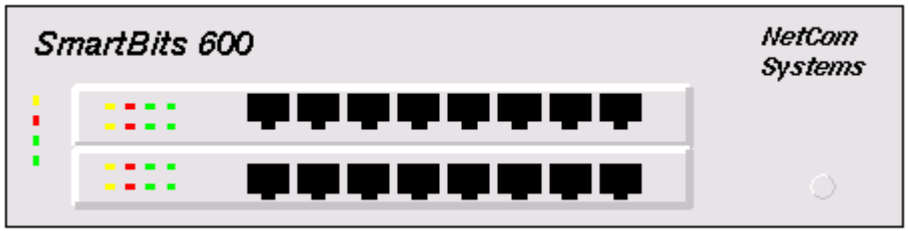
Exemple : un outil graphique de visualisation d'un réseau d'entreprise.

- **Etablir l'interface de visualisation**
  - Définir l'ensemble des matériels réseaux à visualiser



Dessiner un composant réseau

- ✓ Dessiner un routeur



  - ▪ Dessiner un rectangle ■
    - » Dessiner une droite ——

- ○ **Dessine un pixel**

Programmation structurée

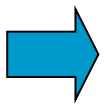
- **Programmation :**
  - **Structure de données pour représenter un pixel**
  - **Fonction DessinePixel(pixel)**
    - » Structure de données pour représenter une droite
    - » Fonction dessineDroite(droite)
    - ▪ Structure de données pour représenter un rectangle
    - ▪ Fonction dessineRectangle(rectangle)
    - ✓ Structure de données pour représenter un routeur
    - ✓ Fonction dessineRouteur(routeur)
  - **Structure de données pour représenter le réseau**
  - **Programme dessineReseau(reseau)**

Programmation objet

- **Programmation en C on aurait**

```
struct pixel {
  int x,y;
  int rgb[3];
};

struct droite {
  struct pixel p1, p2;
};

...

struct routeur{
  struct rectangle boitier, port1, port2, ...;
  char constructeur[256];
  int numero_inventaire;
  int tabrefport[MAXPORT];
  struct appareil * tabappareil[MAXPORT];
  ...
};
```

```
void dessinePixel(struct pixel p){
  ...
}

void dessineDroite(struct droite d){
  ...
}

...

void dessineRouteur(struct routeur r){
  ...
}
```



Idée de la programmation objet :
associer données et fonctions de manipulation

Notions d'héritage et de polymorphisme

- **Pour chaque matériel on aurait**

```
struct routeur{
...
void dessine() {…}
};

struct switch {
...
void dessine() {…}
};

struct hub {
...
void dessine() {…}
};

struct serveur {
...
void dessine() {…}
};
```

```
struct appareilreseau {
...
void dessine() {…}
};

void dessineRouteur(struct routeur r){
  ...
}

void dessineSwitch(struct sxitch s){
  ...
}

void dessineHub(struct hub h){
  ...
}

void dessineServeur(struct serveur s){
  ...
}
```

Tableau d'`appareilreseau`

Idées de la programmation objet : Héritage et polymorphisme.

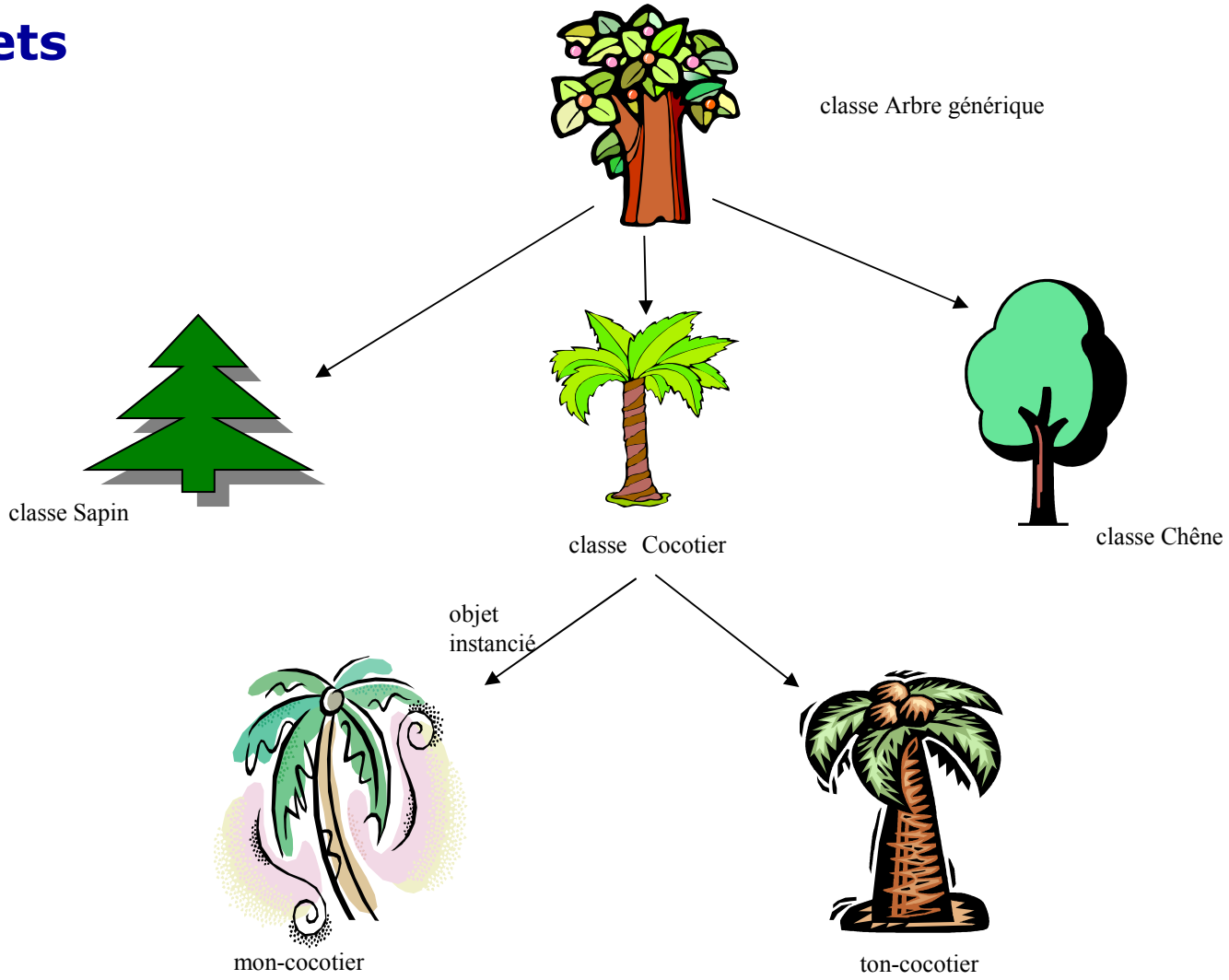
Programmation Orientée Objet

- **La POO (Programmation Orientée Objet)**
  - Fondée sur le concept d'**objet**
  - Par analogie avec l'équation de **Wirth** :

    Méthodes + Données = Objet
  - où méthodes = fonctions associées à un objet.
- **Un programme contient :**
  - types de données basiques : nombres et caractères
  - la possibilité de **définir** les entités d'intérêt, c'est-à-dire de nouveaux types d'objets (**classe**) :
    - ✓ Si vous êtes concerné par le service des ressources humaines, vous créerez une classe d'objet **employe**.
    - ✓ Si vous travaillez aux eaux et forêts, vous créerez une classe d'objets **arbre**.

Conception objet

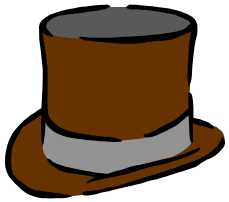
- **Objets**



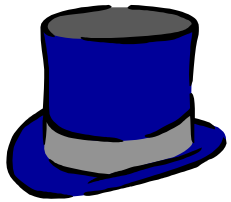
Classes et instances

- **Classes d'objets**

```
public class Chapeau
        {
           private String proprio;
           private String type;
           private String couleur;
           private int taille;
          private boolean mis;
        }
```



`proprio` : Marc
`type` : haut-de-forme
`couleur` : marron
`taille` : 6

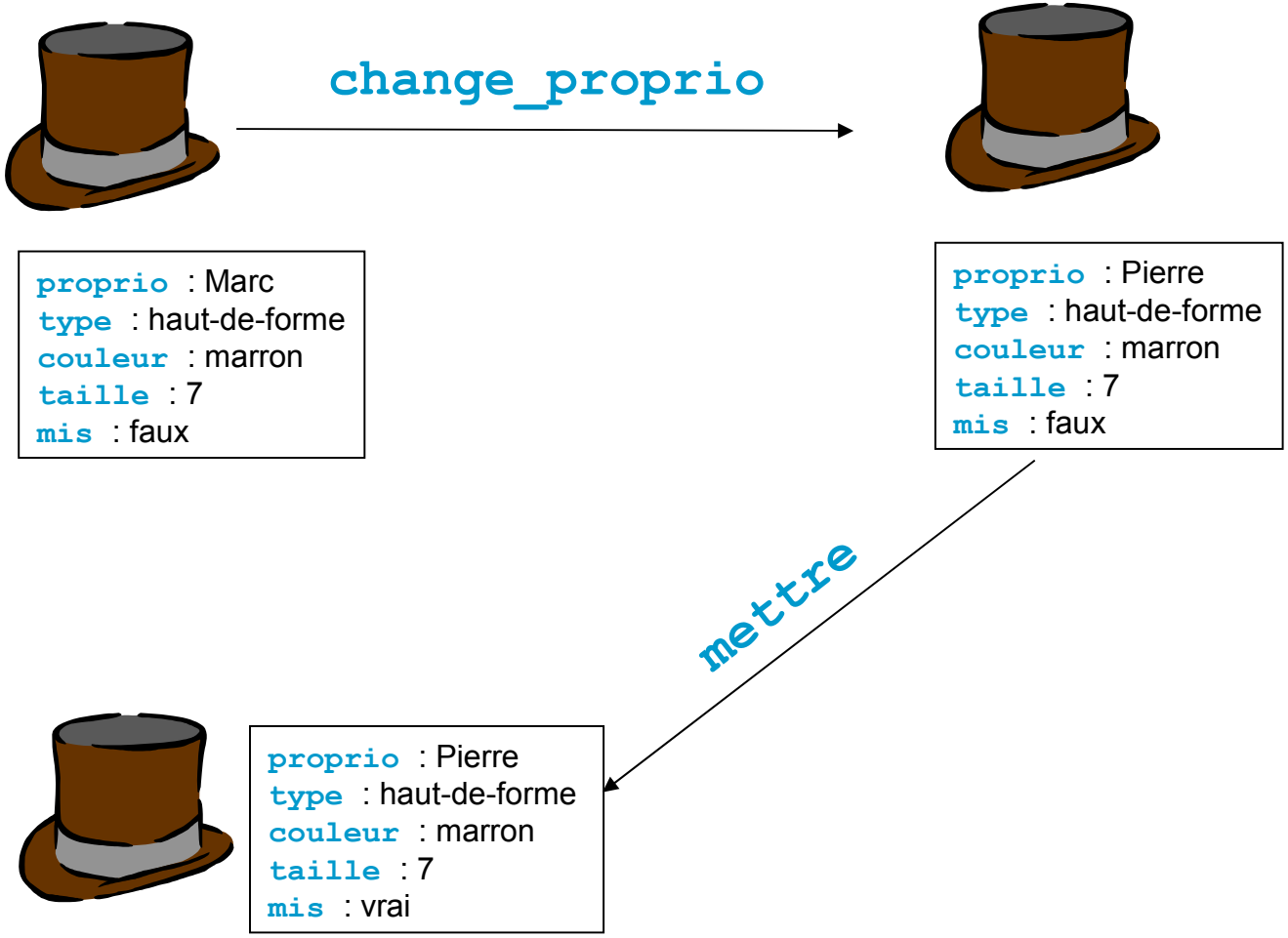


`proprio` : Pierre
`type` : haut-de-forme
`couleur` : bleu
`taille` : 6

`class` est un mot clé.

Une classe contient donc des **données** et spécifie également les **opérations** possibles sur les objets sous la forme de **méthodes**.
Exemple avec les objets `chapeau` : le mettre, l'enlever.
On peut aussi vouloir changer de propriétaire.

Méthodes de classe



Application en JAVA

- **Ce qui donnerait en JAVA :**

```
public class Chapeau {
/* Les « attributs » (champs) du chapeau : */
  private String proprio;
  private String type;
  private String couleur;
  private int taille;
  private boolean mis;

/* Les « méthodes » (fonctions) du chapeau : */
  public Chapeau(String p, String t, String c) {
    proprio = p; type = t; couleur = c;
    mis = false;
  }
  public void change_proprio(String nouveau) {
    proprio = nouveau;
  }
  public void mettre() {
    mis = true;
  }
}
```

Chapitre 3 – Les bases du langage

Conventions

Un programme Java consiste en un ensemble de classes.

- **Il y a au minimum 1 classe, mais il peut y en avoir plusieurs.**
- **Il peut y avoir plusieurs classes de démarrage (chacune ayant une fonction main).**
- **On range le code de chaque classe dans un fichier séparé. On DOIT donner le même nom au fichier et à la classe qui y est déclaré.**
- **Un fichier source Java doit avoir le suffixe `.java`.**
- **Des conventions de codage sont à respecter.**

Premiers mots clés

- **Un premier programme :**

```
/*  L'application Bienvenu */
public class PremProg
{
  /*
   * La fonction main() est toujours la première exécutée au
   * démarrage d'un programme.
   * - Le mot public indique que main est accessible globalement
   * - Le mot static assure que main est accessible même si aucun
   *   objet de la classe n'existe.
   * - Le mot void indique que la méthode ne retourne rien.
   */
  public static void main(String[] args)
    {
      System.out.printf( "Bienvenu dans le monde de Java!" );
    }
}
```

Commentaires

- **Commentaires JAVA**
  - `/* à la mode C */`
  - `// à la mode C++`
  - `/** à la mode javadoc */`
    - ✓ Les commentaires javadoc sont des balises HTML permettant de générer automatiquement de la documentation.
    - ✓ « `javadoc` » ignore tous les espaces au début d'une ligne de même que le premier astérisque.
    - ✓ Il ne peut y avoir d'espace entre le commentaire de classe et la classe.
    - ✓ L'arobas (`@`) permet l'introduction d'informations spécifiques :

`@see class`	Permet de spécifier les classes liées.
`@author Nom`	Permet de citer un ou plusieurs auteurs.
`@param Nom Descript`	Permet de donner de l'information sur un ou plusieurs paramètres.
`@throws Except Desc.`	Permet de donner de l'information sur les exceptions.

Types

- **Les types primitifs**

Type	Contenu	Valeur par défaut	Taille (bits)	Valeur min.	Valeur max.
boolean	true ou false	false	1		
char	Caractères Unicode	\u0000	16	\u0000	\uFFFF
byte	Entier signé	0	8	-128	+127
short	Entier signé	0	16	-32768	+32767
int	Entier signé	0	32	-2147483648	+2147483647
long	Entier signé	0	64	-9223372036854775808	+9223372036854775807
float	Flottant IEEE 754	0.0	32	1.40239846 E -45	3.40282347 E +38
double	Flottant IEEE 754	0.0	64	4.94…E -324	1.797… E +308

Variables et constantes

- **Déclarations de variables**

```
int i = 5, j, k ;
double x = 2.5, y, z ;
boolean b = true, bb;
```

  - Une variable peut-être déclarée n'importe où (pas forcément en début de bloc comme en langage C).
  - La portée d'une variable est limité au bloc dans lequel elle a été déclarée.
  - Une variable doit être déclarée avant toute utilisation.

- **Déclarations de constantes**

```
final double TVA = 19.6 ;
```

  - Une constante se déclare comme une variable en la précédent du mot-clé `final`.
  - D'une manière générale une variable est considérée comme constante si sa déclaration est définie avec le mot-clé `final`.

Conversions

- **Conversions**

```
i = (int) x; // i récupère la partie entière de x

i = 2;

j = 3;

k = i / j; // k vaut 0 car ici '/' est la division entière

z = ((double) i) / ((double) j); // k vaut 0.6666666
```

Opérateurs

- **Les principaux opérateurs :**

Opérateurs	Rôle
+	Addition
-	Soustraction / Opposé
*	Multiplication
/	Division
=	Affectation
>	Supérieur
<	Inférieur
>=	Supérieur ou égal
<=	Inférieur ou égal
==	Test d'égalité
!	Négation logique
!=	Différence

Opérateurs

- **Les principaux opérateurs :**

Opérateurs	Rôle
++	Incrémentation (pré ou post : ++ i ou i ++)
--	Décrémentation(pré ou post : -- i ou i --)
%	Modulo
&&	« Et » logique
||	« Ou » logique
+=	i += j équivaut à i = i + j
-=	i -= j équivaut à i = i - j
*=	i *= j équivaut à i = i * j
/=	i /= j équivaut à i = i / j
( )	Conversion de type
instanceof	Test d'appartenance à une classe

Les tests

- **Les instructions de contrôle pour l'alternative**
  - **Test simples**

```
if ( a > 0 )
    b = a ;
else
    b = -a ;
```

```
if ( a == b ) {
    // plusieurs instructions
}
else {
    // plusieurs instructions
}
```

Choix multiple

- **Les instructions de contrôle pour l'alternative**
  - **Test multiples**

```
switch (variable) {
  case 0: // cas 0
        break;
  case 1: // cas 0
        break;
  case 2: // cas 0
        break;
  …
  default: // cas par défaut
}
```

Les boucles

- **Les instructions de contrôle de répétition**

```
for ( int i = 0 ; i < 10 ; i++ ) {
    tableau [ i ] = i ;
}
```

```
int i = 100 ;
while ( i > 10 ) {
    i -- ;
    // plusieurs instructions
}
```

```
do {   // faire au moins 1 fois
    // instructions
} while ( i != 0) ;
```

Affichage et saisie

- **Pour aller plus loin dans les premiers programmes :**
  - **Affichage :**

```
System.out.printf("Chaine de format" [, arg1, arg2,…]);
```

pour écrire dans la fenêtre de console,
s'utilise comme le `printf` du langage C.
Il existe également `System.out.print` et `System.out.println`

  - **Saisie au clavier :**

La saisie au clavier n'était pas standardisée avant JAVA5, il fallait écrire sa propre classe. JAVA 5 a introduit la classe `Scanner` :

```
/** Classe permettant d'utiliser la fonction Scanner **/
import java.util.Scanner;
public class SaisieClavier{
  public static void main (String [] args){
/** Récupération et affichage d'une chaine de caracteres **/
/** Declaration d'un nouvel objet Scanner **/
    System.out.printf("Entrez votre nom :");
    Scanner clavier = new Scanner(System.in);
/** Recuperation de la chaine de caractère **/
    String nom = clavier.nextLine();
    System.out.printf("Bonjour %s\n", nom);
/** Récupération et affichage d'un entier **/
    int entier = clavier.nextInt();
        System.out.printf("La valeur saisie est %d\n", entier);
     }
```

Tableaux

- **Pour aller plus loin dans les premiers programmes :**
  - **Les tableaux sont des objets particuliers**

✓ Dans un premier temps vous pourrez les utiliser comme en C en remplaçant :

```
int tableau [20] ;
```

par : faisant .

```
int tableau [] ;
tableau = new int[20];
```

✓ Attention comme en C++ les tableaux sont manipulés par référence, mais ils peuvent être définis dynamiquement.

✓ Comme en C les tableaux commencent toujours à `0`.

```
tableau [0] = 10 ;
```

✓ Les tableaux possèdent un attribut `length` qui correspond à leur taille :

```
System.out.printf( "taille = %d", tableau.length);
```

Mathématiques

- **Pour aller plus loin dans les premiers programmes :**
  - **Class Math**

méthode	type d'argument	résultat
sin(arg)	**double (en radians)**	**double**
cos(arg)	**double (en radians)**	**double**
tan(arg)	**double (en radians)**	**double**
asin(arg)	**double**	**double (en radians)**
acos(arg)	**double**	**double (en radians)**
atan(arg)	**double**	**double (en radians)**
Atan2 (arg1, arg2)	**deux double**	**double (en radians)**

Mathématiques

- **Pour aller plus loin dans les premiers programmes :**
  - **Class Math**

méthode	type d'argument	résultat
abs(arg)	**Int, long, float ou double**	**de même type que l'argument**
max(arg1,arg2)	**int, long, float ou double**	**de même type que les arguments**
min(arg1, arg2)	**int, long, float ou double**	**de même type que les arguments**
ceil(arg) **vérifier**	**double**	**double**
floor(arg) **vérifier**	**double**	**double**
round(arg) **vérifier**	**float ou double**	**de type int pour un argument float, de type long pour un argument double**
rint(arg) **vérifier**	**double**	**double**
IEEEremainder **(arg1, arg2)**	**deux type double**	**de type double**

Chaînes de caractères

- **Pour aller plus loin dans les premiers programmes :**
  - **Class String**

✓ Pour représenter les chaînes de caractères :

```
String ch = "bonjour";
```

✓ Pour concaténer :

```
String ch1 = "Un bien beau ";
String ch2 = "langage ";
String ch3 = ch1 + ch2 + 1 + " fois !";
```

✓ Pour convertir vers String :

```
int i = Integer.parseInt(chaine);
float f = Float.parseFloat(chaine);
double d = Double.parseDouble(chaine);
```

✓ Pour convertir en String :

```
int n = 123;
String ch = String.valueOf(n);   // ch = "123"
```

Chapitre 4 – Les classes et les objets

Conception Orientée Objet

- **La conception par objets**
  - **Principe**
    - ✓ Les objets du monde réel ou abstrait constituent le point de départ
    - ✓ Construction à partir des données et non des fonctions
    - ✓ Les traitements se situent auprès des objets sur lesquels ils opèrent
    - ✓ Chaque module possède une interface pour communiquer avec les autres

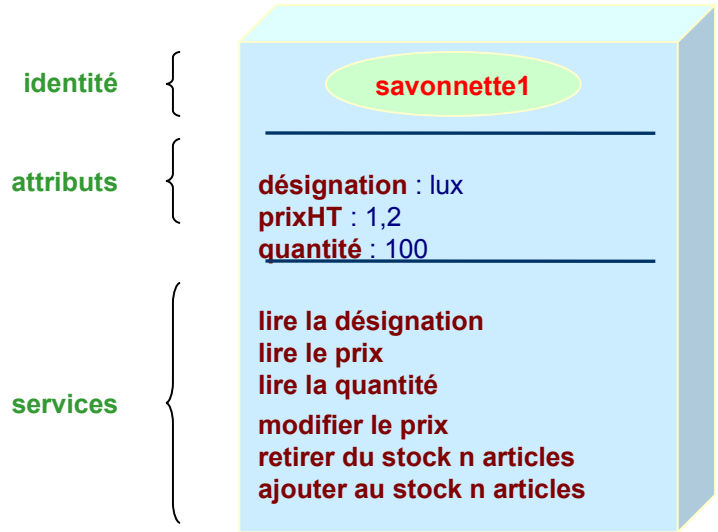
Classes d'objets

- **Étapes de conception**
  - **Identifier les objets**
  - **Décrire les familles d'objets défini par**
    - ✓ leur comportement
    - ✓ les services qu'ils offrent
- **Notion de classe**
  - **Représentation abstraite d'un ensemble d'objets**
  - **Description qui porte autant sur les données que sur les fonctions qui leurs sont appliquées**
  - **Analyse orientée objet regroupe en fait des descriptions de classes**

Objet

- **La notion d'objet**
  - **Un objet possède :**
    - ✓ une **identité** (pour le différencier des autres)
    - ✓ un **état** caractérisé par la valeur de ses **attributs**
    - ✓ Un **comportement** relatif aux services qu'il offre (**méthodes**)

Représentation d'un objet

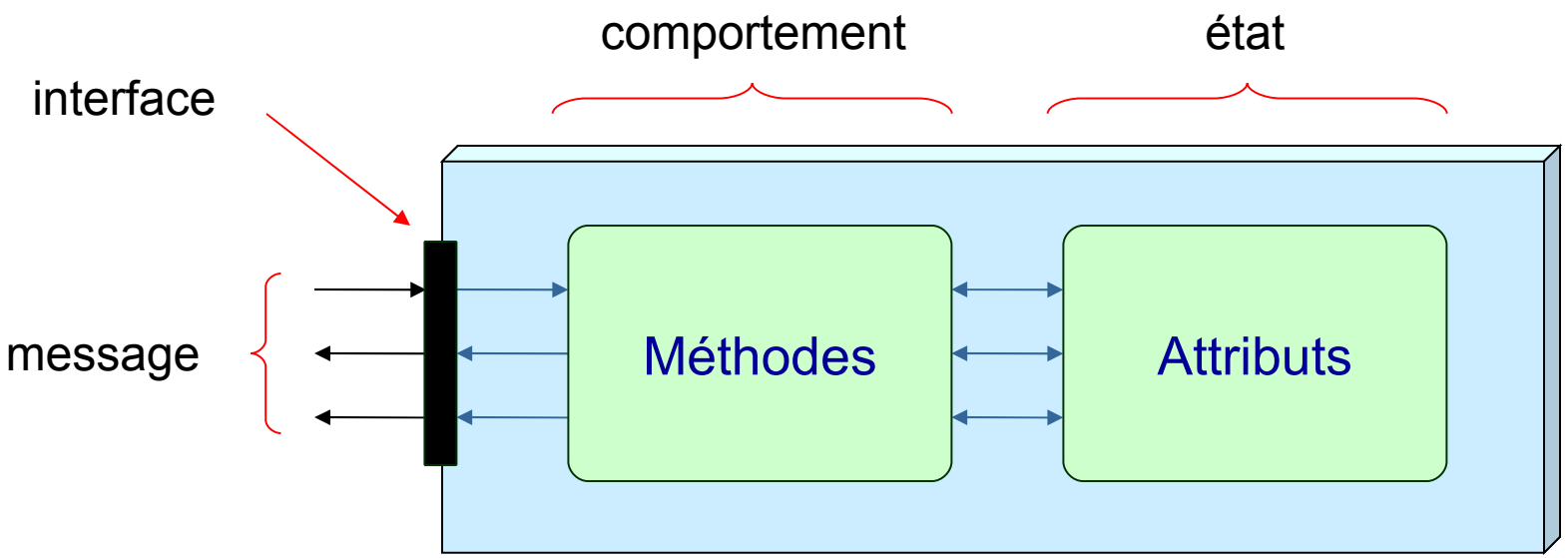


Encapsulation

- **Principe d'encapsulation**
  - **Seul l'objet peut modifier ses propre attributs**
  - **Masquer les détails de l'implémentation**
    - ✓ toutes les méthodes définies du point de vue utilisateur sont publics
    - ✓ les attributs et les autres méthodes sont secrets.
  - **Un contrat (méthodes publiques) définit ce que fait la classe tandis que son implémentation précise comment elle fait.**
  - **Les contrats définis dans la classe sont hérités par les sous-classes**

Encapsulation

- **Représentation**



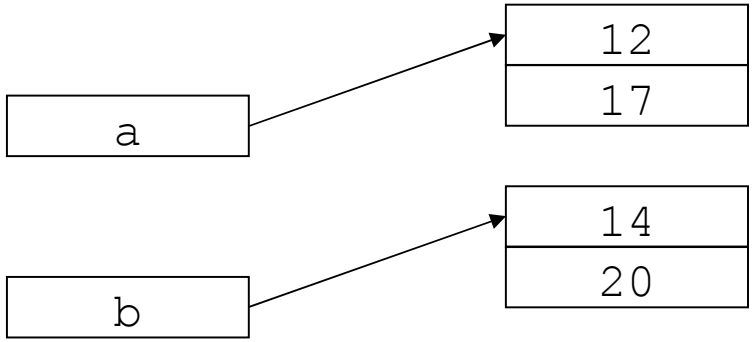
Exemple (avec un constructeur par défaut)

```
public class Point {
    /**
    * Positions x et y courante du point
    */
    private int x;
    private int y;

    /** Initialise une instance de Point */
    public void initialise(int abs, int ord) {
        x = abs;
        y = ord;
    }

    public void affiche() {
        System.out.printf("Je suis un point de
          coordonnées (%d,%d)", x,y) ;
    }

    public void deplace(int dx, int dy) {
        x += dx;
        y += dy;
    }

}
```

Mémoire

- **Affectation et comparaison d'objets**

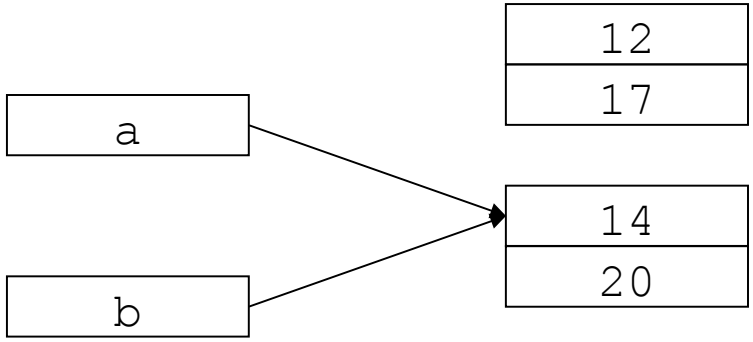
```
Point a,b;
a = new Point();
b = new Point();
a.initialise(12,17);
b.initialise(14,20);
```




```
a.getclass().equals(b.getclass()); // true
```

```
a = b;
```

`a` et `b` désignent le même objet.



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

Question : que devient le Point(12,17) ?

Géré par le **Garbage Collector.**

Méthodes

- **Le modèle représenté par la classes**
  - **Regrouper les données (variables, constantes et agrégation d'autres objets) qui caractérisent chaque instance qui sera générée.**
  - **Définir les services qui permettront de :**
    - ✓ ***construire les objets*** (les « ***constructeurs*** »)
    - ✓ ***modifier les données*** (les « ***modifieurs*** » ou méthodes de transformation)
    - ✓ ***lire les données*** (les « ***accesseurs*** » ou méthodes d'accès).

Exemple

```
public class Point {
    /**
    * Positions x et y courante du point
    */
    private int x;
    private int y;

  /** Constructor of an instance of Point */
 public Point(int abs, int ord) {
        x = abs;
        y = ord;
  }
  /** Methode de type accesseurs */
 public void affiche() {
      System.out.println("Je suis un point "+
        "de coordonnées (" + x +","+ y + ").");
    }

 /** Methode de type modifieurs */
 public void deplace(int dx, int dy) {
       x += dx;
       y += dy;
    }
}
```

Attributs : types simples *vs* objets

- **Implantation des attributs**
  - **attributs primitifs :**
    - ✓ `int entier1 = 10;`
    - ✓ `int entier2 = entier1;`

déclaration de **`entier1`** de type int de valeur 10
déclaration de entier2 de type int de même valeur que **`entier1`**

  - **attributs objets**
    - ✓ `Point pointA = new Point(12,17);`
    - ✓ `Point pointB = pointA;`

déclaration d'une référence pointA accrochée à un objet de la classe Point initialisé aux coordonnées (*12*,*17*).
déclaration d'une nouvelle référence pointB qui accroche le même objet que pointA

Exemple

```
public class Point {
    int x,y;
    String name;

    public Point(String unnom) {
        x=0;
        y=0;
        name = unnom;
    }
    public void affiche()
    {
      System.out.printf
       ("x=%d, y=%d, Name=%s\n",x,y,name);
    }
    public void incrementeX(){
        x++;
    }
    public void changeName(String nouvNom){
        name=nouvNom;
    }
}
```

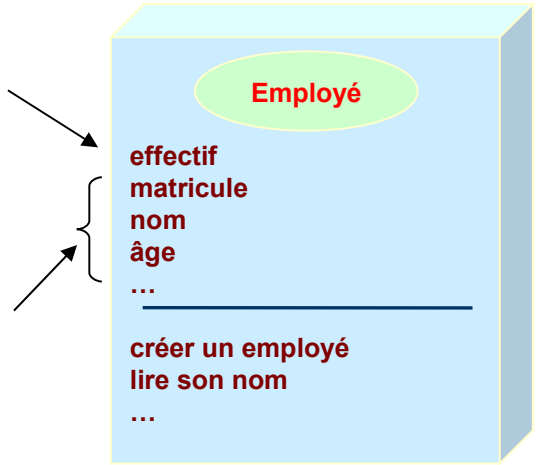
```
public class Testaffectationtypeprimitifvsobjet {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Point p1 = new Point("Origine");
        Point p2 = p1;

        p2.changeName("A");
        p2.incrementeX();

        System.out.printf("p1: "); p1.affiche();
        System.out.printf("p2: "); p2.affiche();

        int c = 10;
        int d = c;

        d++;

        System.out.println("  c="+c+"  d="+d);
    }
}
```

Attributs

- **Implantation des attributs**
  - **Variable de classe**
    - ✓ donnée commune à l'ensemble des objets
  - **Variable d'instance**
    - ✓ donnée propre à l'objet
  - **Exemple :**
    - ✓ chaque employé d'une entreprise possède un numéro de matricule;
    - ✓ le nombre total des employés constitue l'effectif de l'entreprise.

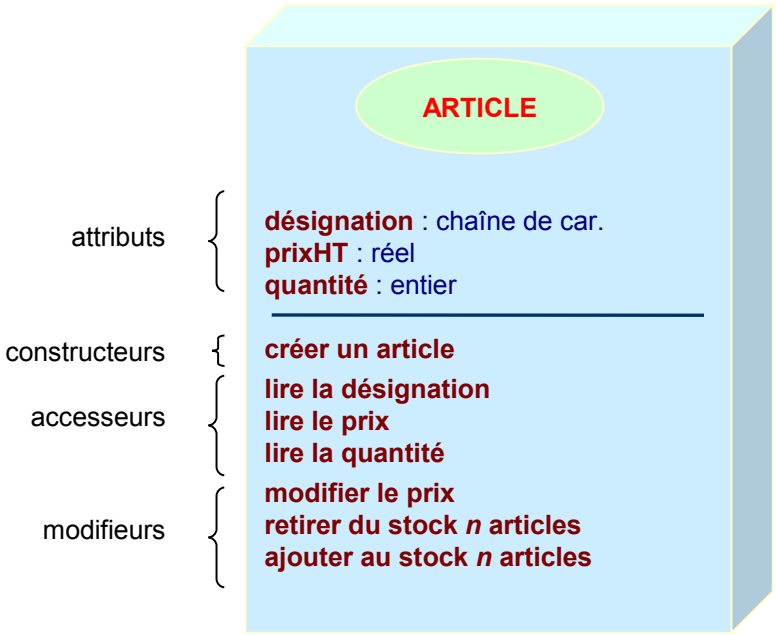
Variable propre à l'ensemble des employés

Variable propre l'employé



Attributs

- **Le modèle représenté par la classes**



Exemple

```
public class Article {
     private String désignation;
     private float prixHT;
     private static long quantité = 0;

     public Article(String nom, float prix) {
          désignation = nom;
          prixHT = prix;
          quantité++;
     }
/* Les accesseurs */
     public String lireDésignation() {
          return désignation;
     }
     public float lirePrix() {
          return prixHT;
     }
     public long lireQuantite(){
          return quantité;
     }
/* Les modifieurs */
     public void modifierPrix(float nouveauprix) {
          prixHT = nouveauprix;
     }
     public void retirerduStock(long n) {
          if (n>=0)
               if (quantité >= n)
                    quantité -= n;
               else
                    quantité = 0;
     }
     public void ajouterauStock(long n) {
          if (n<0)
               retirerduStock(-n);
          else
                    quantité += n;
     }
}
```

Attributs : résumé

- **Implantation des attributs**
  - Ils peuvent être un objet ou une primitive
  - Ils peuvent être une variable de classe ou d'objet
  - Ils sont désignés par des identificateurs
  - Ils doivent être initialisés
  - Ils sont consultés grâce aux méthodes d'accès
  - Ils sont modifiés par les méthodes de transformation

Méthodes

- **Implantation des méthodes**
  - **Une méthode représente un service rendu par les objets de la classe**
  - **Plusieurs méthodes peuvent avoir le même identificateur si elles n'ont pas la même signature (polymorphisme par surcharge)**
  - **La portée des identificateurs des méthodes s'étend à la totalité du bloc qui les contient**

Méthodes

- **Implantation des méthodes**
  - **Une méthode peut :**
    - ✓ Appartenir à la classe (`static`)
    - ✓ Ne pas avoir d'implémentation, celle-ci est laissée au bon soin des classes hérités (`abstract`).
    - ✓ Bloquer toute surcharge (`final`)

Méthodes

- **Surcharge de méthodes**
  - **Plusieurs méthodes peuvent porter le même nom.**
  - **On les distinguent par le nombre et le type de leurs paramètres.**

```
public void deplace(int dx, int dy) {
        x += dx;
        y += dy;
    }

public void deplace(int dx) {
        x += dx;
    }

public void deplace(short dx) {
        x += dx;
    }
```

Méthodes

- **Echange d'information avec les méthodes**
  - **Dans JAVA la transmission d'information primitives se fait toujours par valeur.**
  - **Tandis que la transmission d'objets se fait toujours par référence.**

Autoréférence

- **Autoréférence**
  - **Le mot-clé `this` désigne l'objet lui-même.**

```
public Point(int x, int y){
    this.x = x;
    this.y = y;
}
```

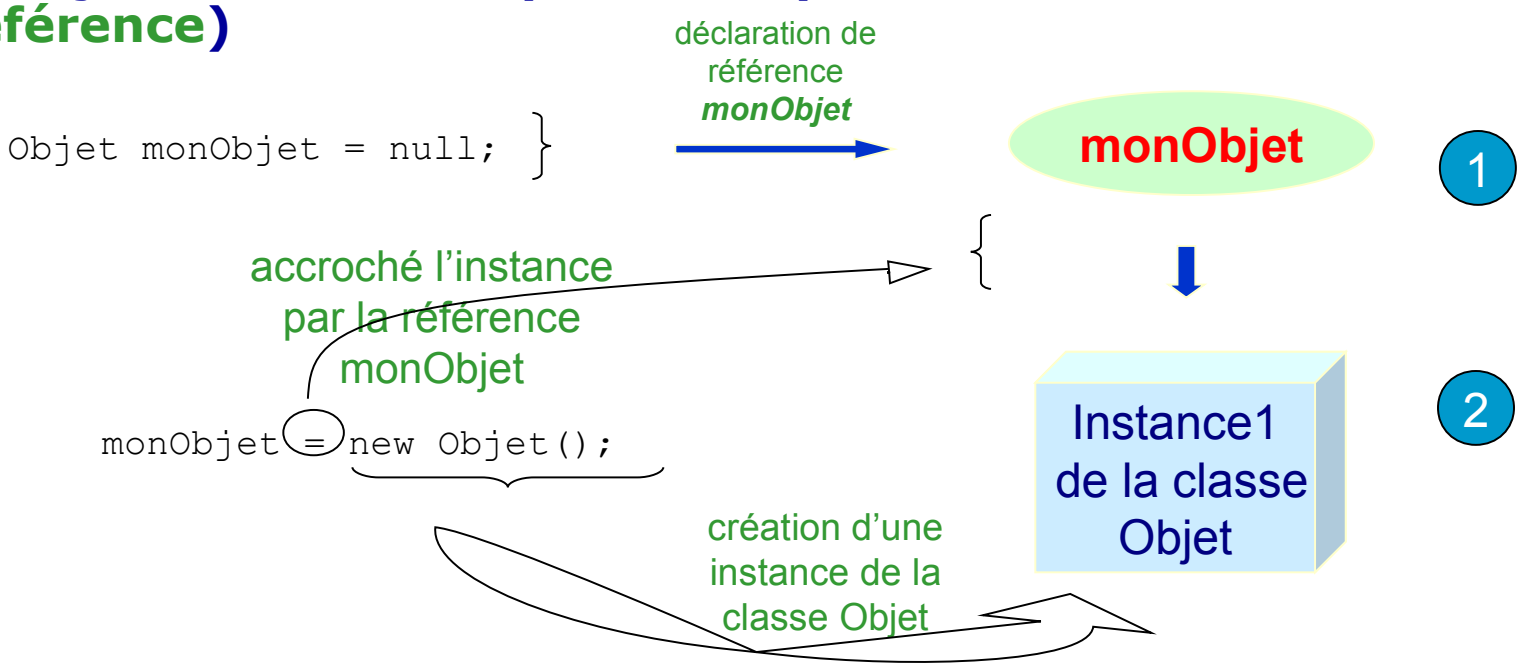
  - `this(…)` **correspond à l'appel du constructeur.**

```
public Point(){
    this(0,0);
}
```

Résumé

En résumé :

- **Une classe peut être assimilée à un type**
- **Un objet est l'instance d'une classe d'implémentation**
- **L'objet créé est à l'image de sa classe**
- **Un objet existe en mémoire dès qu'il est créé**
- **Un objet est accroché par un ou plusieurs identificateurs (référence)**



Package

- **Notion de paquetage (`package`)**
  - **Regroupement logique d'un ensemble de classes sous un identificateur commun.**
  - **Facilite la cohabitation de logiciels.**
  - **Permet de créer des classes qui ont le même nom sans créer d'interférences.**

  - **La définition d'un nom de paquetage se fait au niveau du fichier source par :**

```
package nomdepaquetage;
```

  - **Pour utiliser une classe d'un paquetage :**

```
monPaquetage.Point p = new monPaquetage.Point(2,5);
```

ou :

```
import monPaquetage.Point;

Point p = new Point(2,5);
```

Pourquoi utiliser des *packages* ?

- **Intérêts**
  - Mise à disposition « structurée » de classes à la communauté
  - Trouver les classes à charger lors du lancement de la MV
    - Les répertoires qui correspondent au nom des packages
    - Un fichier d'archive (.zip ou .jar) contenant une arborescence de packages
  - Éviter les conflits de noms entre classes développées par des programmeurs différents
  - Délimiter un espace de visibilité des variables (public, private …)

Comment utiliser les *packages* ?

- **Visibilité**

  - Toute classe d'un package est accessible depuis toutes les autres classes du même package
  - Seules les classes « public » sont accessibles d'un autre package
  - Un membre d'une classe est accessible depuis une autre classe du package s'il n'est pas « private »
  - Les membres d'une classe sont accessibles depuis un autre package si la classe et les membres sont « public »

API Java

- **Visibilité**

Nom du package	Contenu
java.lang	Bases du langages (type de données)
java.io	Entrées - Sorties
java.math	Fonctions mathématiques (sin, cos, tang …)
java.awt	Gestion des fenètres, GUI
javax.swing	Classes graphiques, IHM
java.util	Classes utilitaires (Vector, Stack …)
java.applet	Codage des applets
java.net	Applications réseau
java.rmi	Réseau avec technologie RMI
java.beans	Objets « métier »
java.security	Classes liées aux aspects sécurité
…	