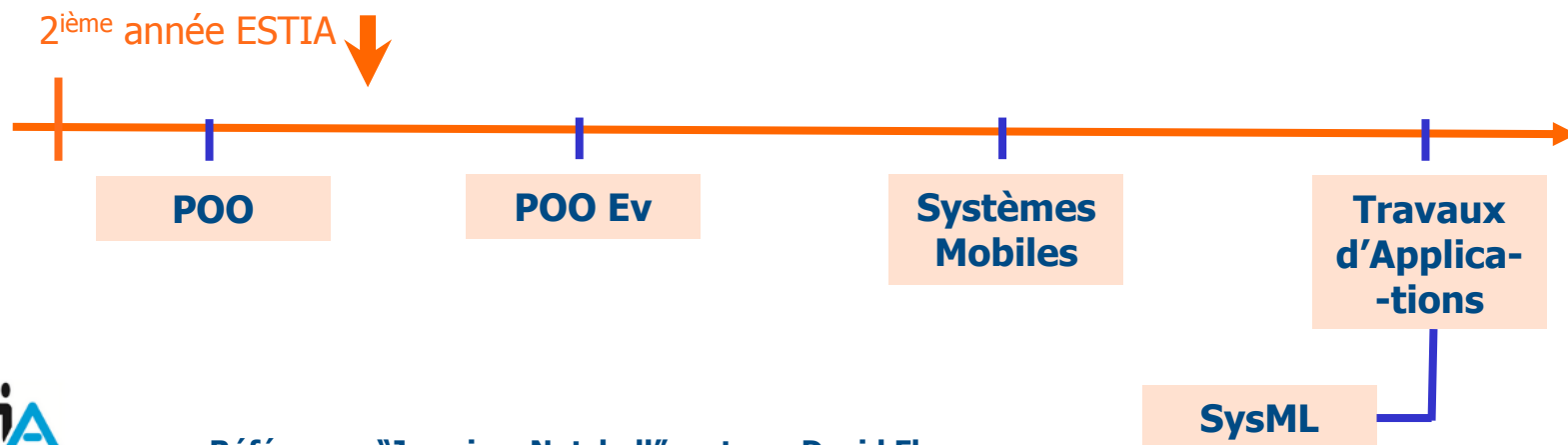


# ***Programmation Orientée Objet (JAVA)***

***Auteur: Nadine Couture, David Gomez***

***Cours amphi: David Gomez, Guillaume Rivière***

***Pour les TD-TP : Clement Merino, Maxime Daniel, Stéphane Perret, Emeric Baldisser, Dimitri Masson, Guillaume Rivière, Daniel Sourgen, Cindy Becher, David Gomez***



Référence: "Java in a Nutshell", auteur: David Flanagan



## ***Programmation Orienté Objet***

A l'issue de ce cours l'élève-ingénieur sera capable d'appliquer les concepts de la POO et d'utiliser la syntaxe du langage Java ainsi que les bibliothèques standard (API) pour développer des applications en Java.

## ***Bibliographie/Webographie***

- Modélisation objet avec UML, Pierre-Alain Muller, Eyrolle
- <http://java.sun.com/>
- Java in a Nutshell, 5 th edition, David Flanagan, O'REILLY
- Learning Java, 4 th Edition, Patrick Niemeyer & Daniel Leuck, O'REILLY

# Plan du cours 1/4



## ***Programmation Orienté Objet***

1. Introduction
  - a. Introduction à la POO
  - b. Classes et objets
  - c. Origines du langage JAVA
  - d. Survol du langage, ses propriétés
  - e. Votre premier programme « Hello World »

# Plan du cours 1/4

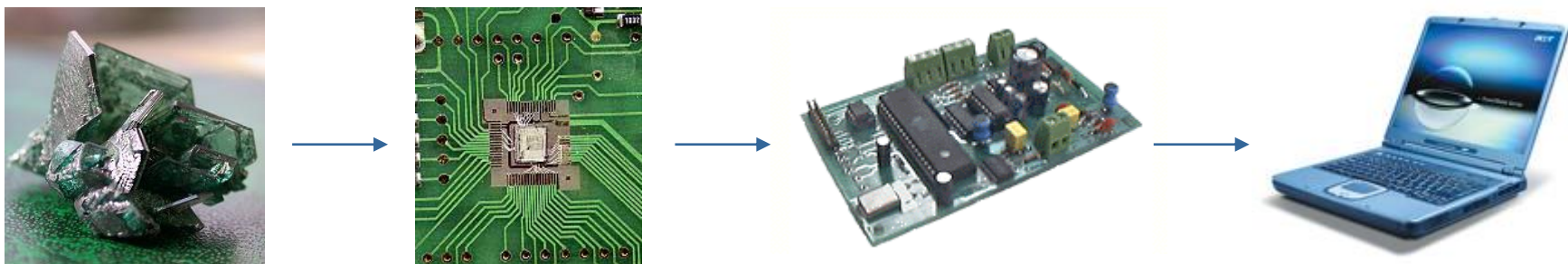


## ***Programmation Orienté Objet***

1. Introduction
  - a. Introduction à la POO
  - b. Classes et objets
  - c. Origines du langage JAVA
  - d. Survol du langage, ses propriétés
  - e. Votre premier programme « Hello World »

# Introduction à la POO

## Une explication : la réutilisation



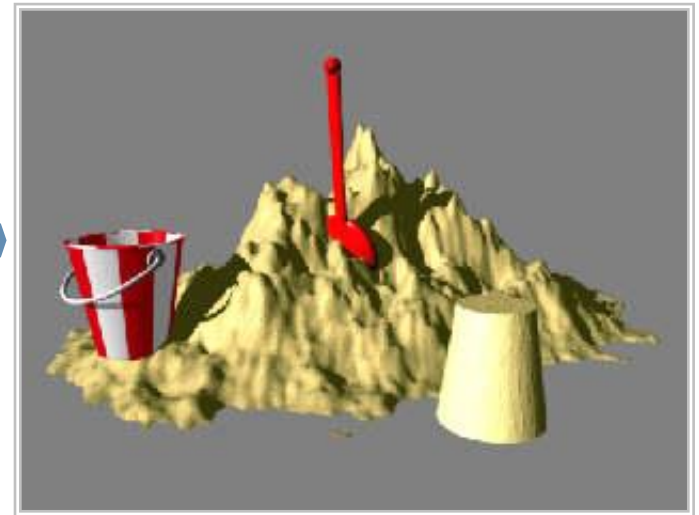
Les ingénieurs électroniciens ne repartent pas de zéro dès qu'ils conçoivent un nouveau dispositif.

Les informaticiens non plus ... mais c'est plus récent !

# Introduction à la POO

Dans tous les domaines on conçoit en assemblant des briques de base.  
En informatique les briques sont toutes petites.

Phénomène « tas de sable »



Projets (trop) complexes

# Introduction à la POO

## Conséquences ...

- Risque de déficience
- Mise au point lente
- Maintenance disproportionnée
- Coût total exorbitant



... en pratique



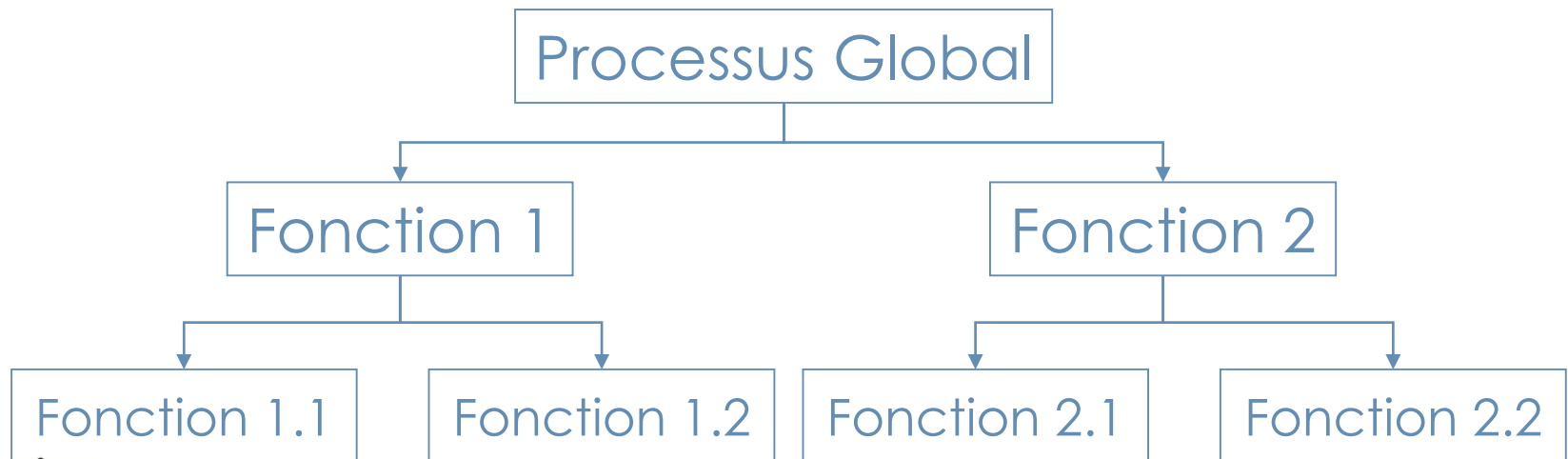
Echecs retentissants

- Explosion d'Ariane 5
- Aéroport de Denver
- Réservation SNCF



# Introduction : Modélisation procédurale

- **Principe** : décomposition des tâches en sous-tâches plus simples à réaliser
- Décomposition unidirectionnelle du Cahier des Charges vers les sous-programmes



# Introduction : Modélisation procédurale



Semble intuitive



Met en avant le « FAIRE »



Adaptée à un processus global parfaitement spécifié



Très rigide



Peu extensible

Ecroutement de l'arbre en cas de remise en cause du processus global

# Complément à la prog. procédurale

---

## Solution

Conception modulaire

Importer, avoir besoin d'un module, séparer les fonctions en les regroupant par champ d'application.

Langages de programmation: C, C++

# Alternative à la prog. procédurale

---

## Solution

Utiliser des composants **existants**, déjà testés.

La **conception par objets** permet d'obtenir des applications modulaires et extensibles, aux composants réutilisables.



Logiciels plus fiables,  
développés plus rapidement.

# Introduction : Modélisation objet

---

## Historique

- 1993 : 2 principales méthodes : BOOCH et OMT
- 1995 : vue unifiée (v.0.8)
- 1997 : version 1.0 de UML : **U**nified **M**odeling **L**anguage.

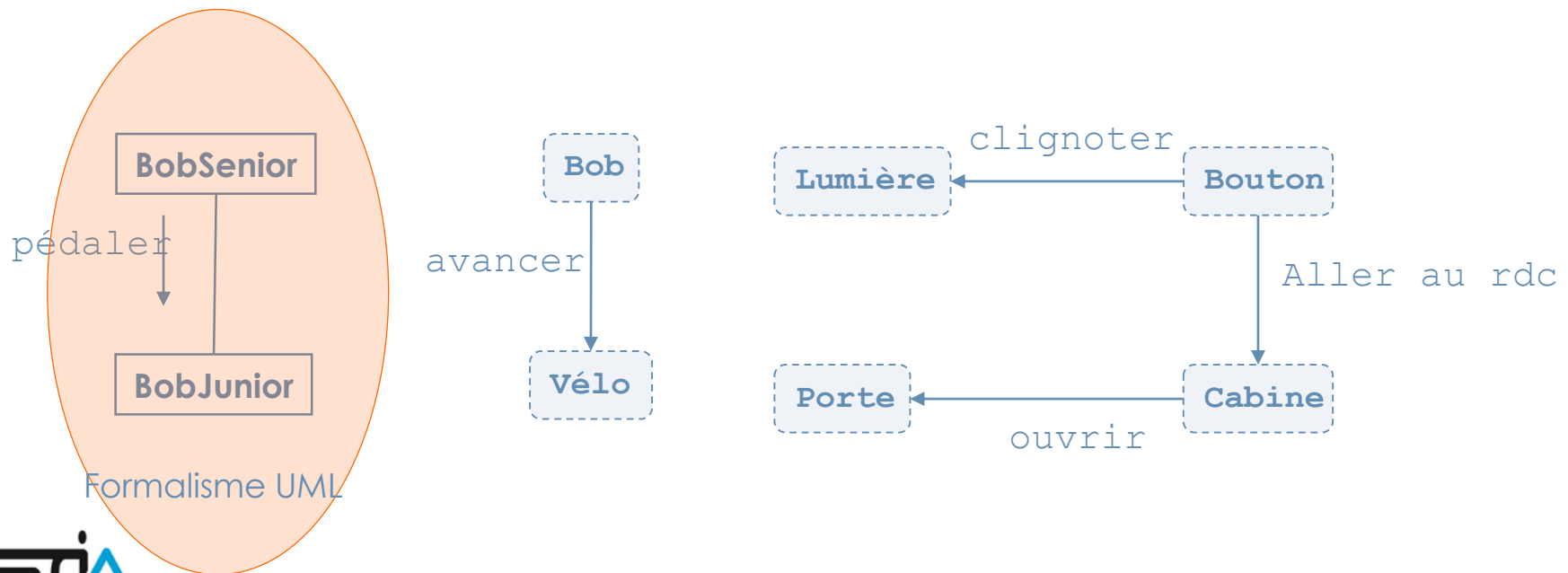


**UML = formalisme**  
de modélisation  
orientée objet  
**pas** une  
**méthode**

**RUP = méthode**

# Introduction : Modélisation objet

**Principe** : décomposition du système en un ensemble d'entités qui collaborent (les **objets**) en s'échangeant des **messages** (qui sont toujours des actions).



# Introduction : Modélisation objet



Colle à la réalité



Met en avant l'« ETRE »



Tout n'est  
pas objet ?



Modularité naturelle



Constructions itérative,  
descendante, ascendante,  
incrémentale ...



Simple (faible nombre de  
concepts)

# Introduction : pourquoi ... ?

---

... pourquoi faire de la POO ?

- Pour des logiciels modulaires
- Pour des logiciels extensibles
- Pour des logiciels plus sûrs
- Pour des logiciels développés plus rapidement



# Plan du cours 1/4



## ***Programmation Orienté Objet***

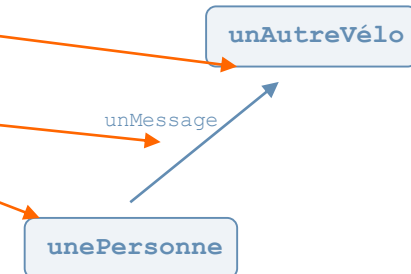
1. Introduction
  - a. Introduction à la POO
  - b. Classes et objets**
  - c. Origines du langage JAVA
  - d. Survol du langage, ses propriétés
  - e. Votre premier programme « Hello World »

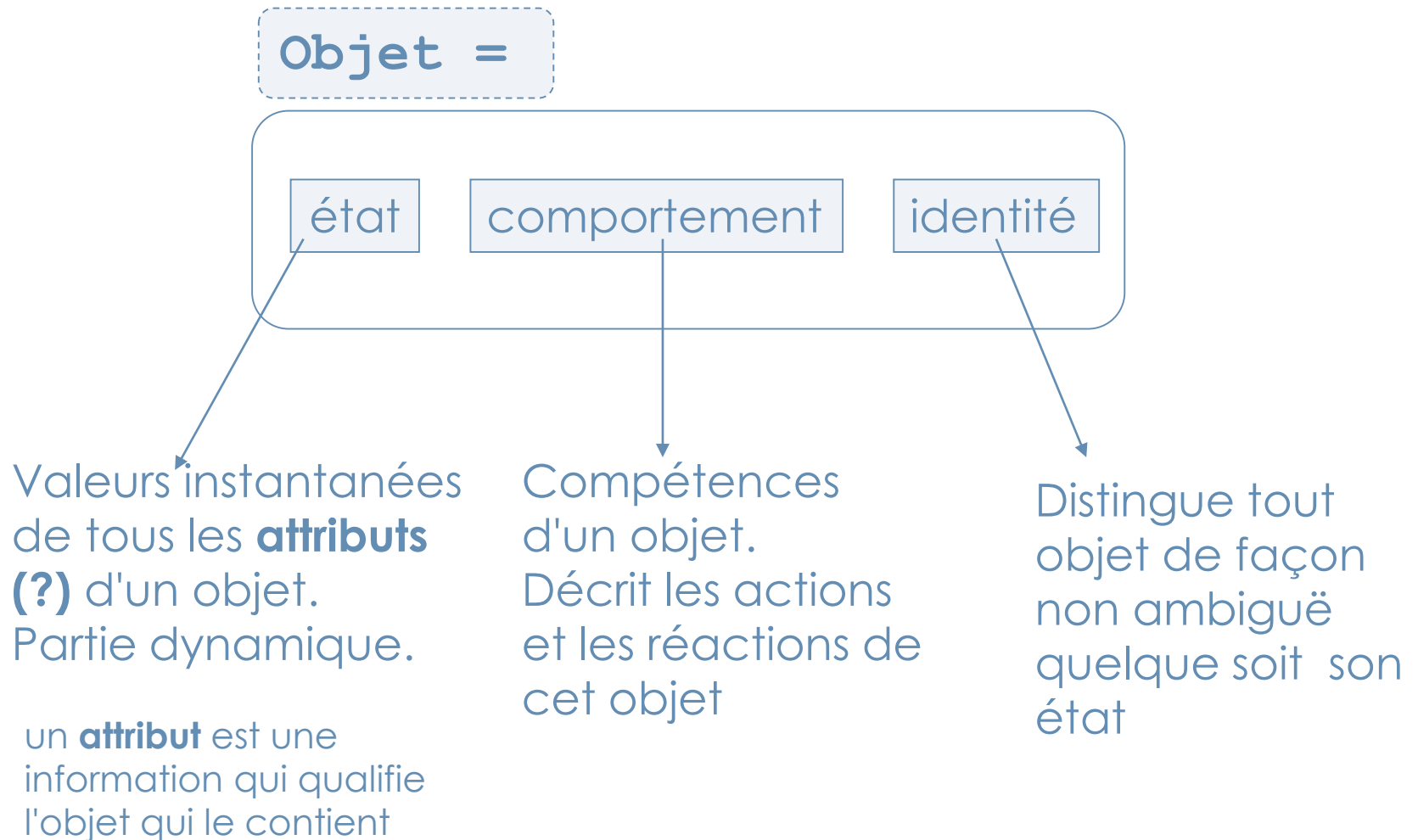
# Introduction : Modélisation objet

## Concepts

- Objets
- Message
- Classe
- Héritage
- Polymorphisme

Velo	
Couleur	: couleur
Poids	: réel
Pignon courant	: entier
Pédaliercourant	: entier
Cadence pédalage	: entier
Vitesse courante	: réel
Changer de pignon (+1 ou -1)	
Afficher vitesse courante	
Tourner (angle)	
Augmenter cadence (incrément)	
Freiner (intensité)	
Calculer vitesse	





# Les objets : exemple

Objet =

état

comportement

identité

monveloprefere

Velo

couleur	rose
poids	11 kg
pignonCourant	17
pedalierCourant	53
cadenceDePedalage	75 tr/mn
vitesseCourante	30 km/h

```

changerDePignon (i)
afficherVitesseCourante ()
tourner ()
augmenterCadence (nouvelle cadence)
freiner ()
calculerVitesse ()
    
```

**Remarque** : pas de méthode pour augmenter directement la vitesse, elle est calculée en fonction du pignon, du pédalier et de la cadence.

Calculer vitesse est appelée avant d'afficher le résultat.

# Les messages

- Les objets communiquent en échangeant des **messages**.
- Les messages représentent le comportement ou les **services** d'un objet.
- Les messages sont composés de :
  - Un nom
  - Une liste de paramètres d'entrée
  - Une liste de paramètres de sortie
- La réception d'un message implique un **traitement**.

→ *signature*

# Les messages

---

Il existe 5 catégories de messages :

- les constructeurs, qui créent les objets
- les destructeurs, qui détruisent les objets
- les sélecteurs qui renvoient tout ou partie de l'**état** d'un objet
- les modificateurs qui changent tout ou partie de l'**état** d'un objet
- les itérateurs qui visitent l'**état** d'un objet ou le contenu d'une **structure de données** qui contient **plusieurs objets**.

# Les messages

---

Un envoi de message peut impliquer deux types de comportement pour l'objet qui le reçoit :

- **Statique** si l'état de l'objet n'intervient pas dans le traitement invoqué ;
- **Dynamique** si le comportement de l'action invoquée dépend de l'état de l'objet

# Les messages : exemple

```
conduireVelo {
    Instruction 1
    Instruction 2
    ...
    changerDePignon (+1)
    ...
}
```

## Velo

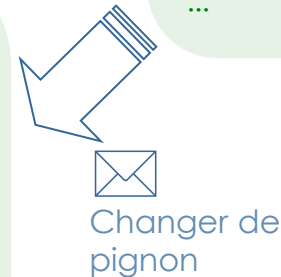
couleur	→	rose
poids	→	11 kg
pignonCourant	→	<del>17</del> 18
pedalierCourant	→	53
cadenceDePedalage	→	75 tr/mn
vitesseCourante	→	<del>30</del> 31

```
changerDePignon (i)
afficherVitesseCourante ()
tourner ()
augmenterCadence (nouvelle cadence)
freiner ()
calculerVitesse ()
```

## Personne

nom	→	Guiesse
prénom	→	Jean-Roch
age	→	30
...	→	..

```
...
conduireVelo
...
```



```
changerDePignon (i) {
    pignonCourant + i
    calculerVitesse ()
}
```



# Objets et messages ...

---



Où est la maîtrise de la complexité ?

On a juste une multitude d'objets indépendants ...

- Pour maîtriser la complexité on **regroupe** les objets qui partagent des propriétés et des comportements communs

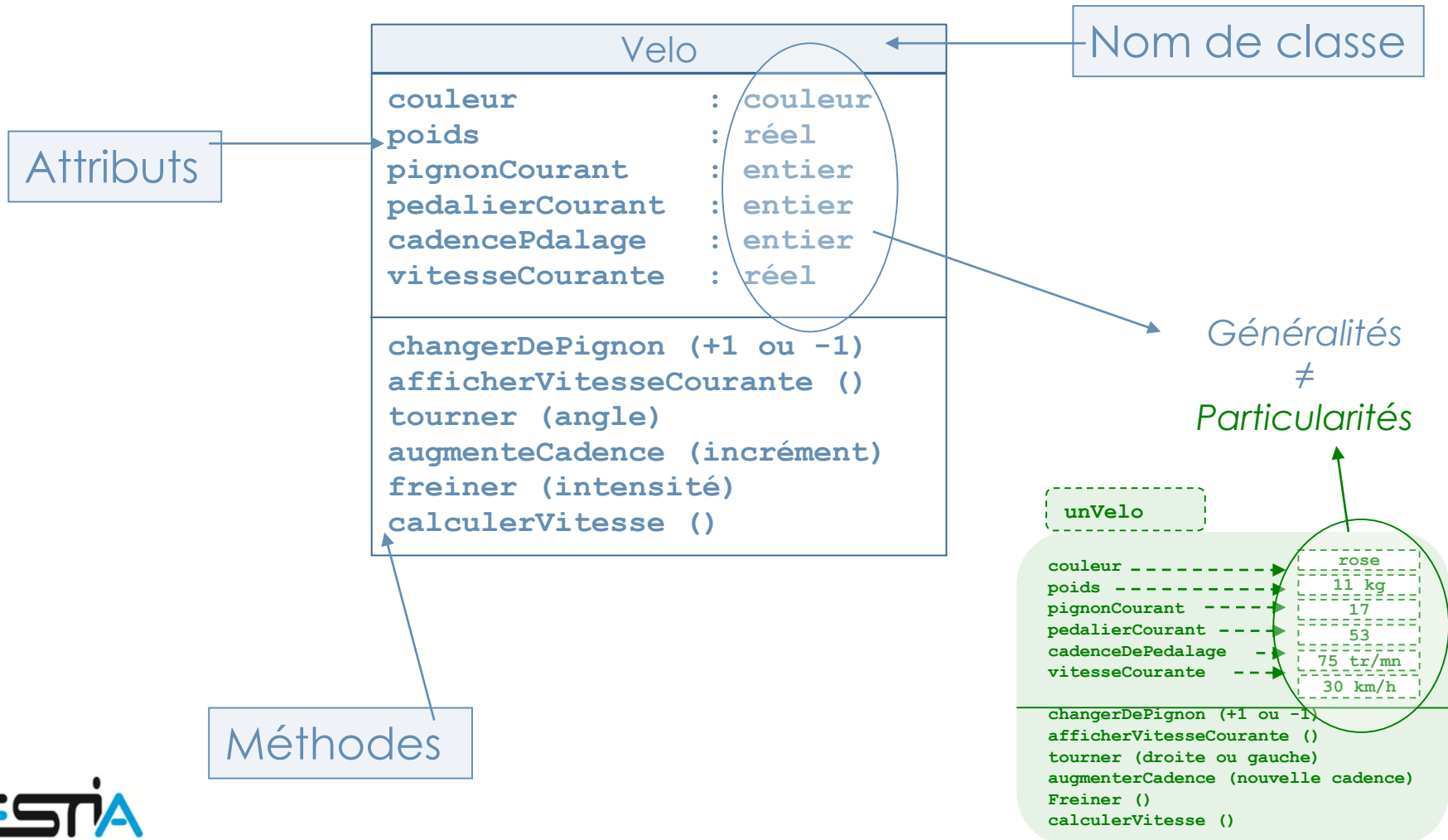
La classe décrit le **domaine de définition** d'un ensemble d'objets.



Chaque objet appartient à une classe.

- Les généralités sont contenues dans la classe
- Les particularités sont contenues dans l'objet

# Les classes : exemple



Les objets sont donc construits à partir de classe par un processus (une technique) appelée instanciation.

« Tout objet est une instance de classe »

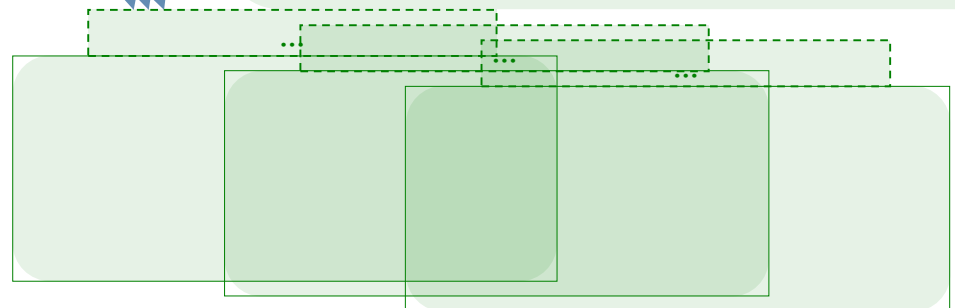
# Exemple d'instanciation

Velo	
couleur	: couleur
poids	: réel
pignonCourant	: entier
pedalierCourant	: entier
cadencePdalage	: entier
vitesseCourante	: réel
changerDePignon (+1 ou -1) afficherVitesseCourante () tourner (angle) augmenterCadence (incrément) freiner (intensité) calculerVitesse ()	

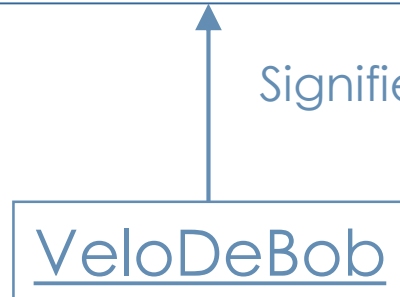
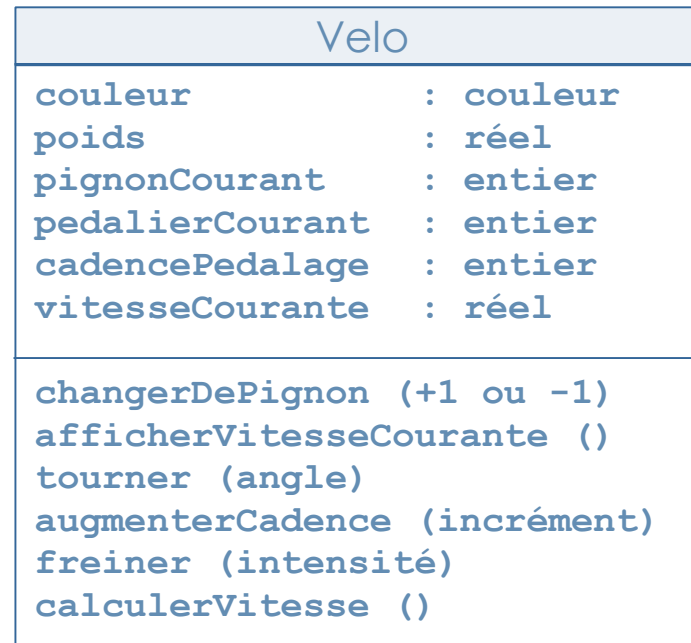
couleur	----->	rose
poids	----->	11 kg
pignonCourant	----->	17
pedalierCourant	----->	53
cadenceDePedalage	----->	75 tr/mn
vitesseCourante	----->	30 km/h

## LeVeloDeCiraptor

couleur	----->	bleu
poids	----->	6,7 kg
nombreDePignons	----->	14
pignonActuel	----->	53
cadenceDePedalage	----->	115 tr/mn
vitesseActuelle	----->	55 km/h



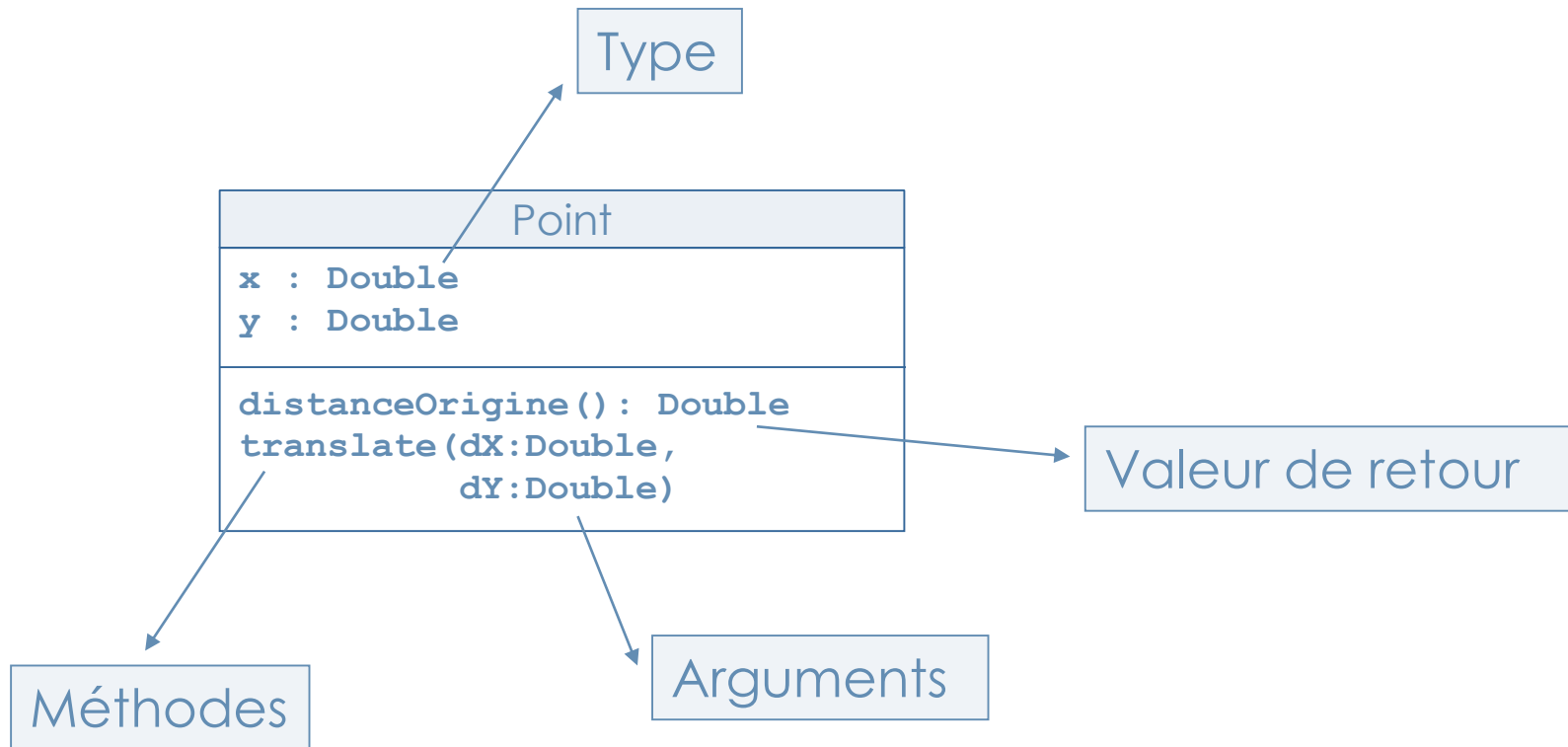
# Rmq : Notation UML



Signifie « est une instance de »

# Autre exemple de classe

- **2 Attributs** : 2 coordonnées cartésiennes
- **2 Méthodes** : la distance à l'origine et déplacement





## Une classe

- définit les possibilités des objets d'un type donné ;
- décrit l'ensemble des données (**état**) et des opérations sur ces données (**traitement**) ;
- sert de "modèle" pour la création d'objets (**instances de la classe**)

# Plan du cours 1/4



## ***Programmation Orienté Objet***

1. Introduction
  - a. Introduction à la POO
  - b. Classes et objets
  - c. Origines du langage JAVA**
  - d. Survol du langage, ses propriétés
  - e. Votre premier programme « Hello World »

# Origines (1/4)

## Besoin

Fabriquer des logiciels destinés à l'électronique grand public :

électroménager (grille-pain, four à micro-ondes, ...), domotique (centrale de surveillance, robots, ...) / téléviseurs interactifs, assistants personnels / téléphones mobiles, ordinateurs portables / tablettes, ...



## Conditions nécessaires

1. Capable de tourner immédiatement sur de nouvelles puces (**portable**) donc économique
2. Très fiable

# 1990

- C/C++ pas vraiment adapté
- James Gosling : conception d'un langage satisfaisant 1. et 2. (au moins)



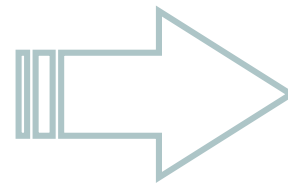
- Naissance de **OAK**
  - Jeu d'instructions réduit
  - Fiable
  - Indépendant de l'architecture matérielle



## Origines (3/4)

- WWW fait son entrée sur Internet
- Mosaïc rend le web de plus en plus populaire
- OAK devient **JAVA**

Java : idéal pour travailler sur Internet, un programme tourne sur toutes les machines connectées au réseau



**HotJava**

1er navigateur supportant les  
**applets (?!?)**

**1993-95**

**Définition** : une applet (ou appliquette) est un programme écrit en JAVA qui peut être embarqué dans une autre application.

*Inclure un applet dans un document HTML crée un élément exécutable et interactif au sein d'une page WWW.*

HotJava a démontré la puissance de Java  
=> sortie de JAVA et de son interface de développement appelée API (**A**pplication **P**rogramming **I**nterface)

1993

# Plan du cours 1/4



## ***Programmation Orienté Objet***

1. Introduction
  - a. Introduction à la POO
  - b. Classes et objets
  - c. Origines du langage JAVA
  - d. Survol du langage, ses propriétés
  - e. Votre premier programme « Hello World »

- SUN définit le langage Java comme
  1. Simple
  2. Orienté objet
  3. Réparti
  4. Interprété
  5. Robuste
  6. Sûr
  7. Indépendant des architectures matérielles
  8. Portable (oui mais ...)
  9. Performant (non mais ...)
  10. Multitâche
  11. Dynamique
- De nombreuses versions  
1.0 (95), 1.4 (2002), 5 (2004), 6 (2006), 7 (2011),  
8 (2014), 9 (2017), 10 (2018)



# Survol – (1) Simple

- ✓ Assimilable rapidement (familier, C++)
- ✓ Suppression des fonctionnalités :
  - ❑ qui entraînent des pratiques douteuses
  - ❑ qui sont mal comprises et mal utilisées
    - Pas de *goto*
    - Pas de fichier d'en tête (.h en C)
    - Pas de pré-processeur (macros : #define)
    - Pas de structure (`struct` en C ou `record` en Ada)
    - Pas de surcharge d'opérateur (+, x, /, ...)
    - Pas d'héritage multiple
    - Pas de pointeurs (gestion cachée)
- ✓ Gestion mémoire par Ramasse-miettes (*garbage collector*)
- ✓ Petit (liaison à l'exécution) => internet, embarqué, ...  
L'interpréteur de base fait 40Ko !

# Survol – (2) Orienté Objet

---

- JAVA entièrement objet ( $\neq$  C++)
- Chaque fichier **source** contient une **classe**, contenant des **données** et des **opérations** de traitement de ces données. Combinée avec d'autres, elle forme une **application**.
- Intérêt :
  - Conception « naturelle » (**pilotée par les données**)  
Exemple : modélisation d'une voiture
  - Réutilisation

# Survol – (2) Orienté Objet

---

JAVA livré avec un vaste ensemble de classes, organisées en **paquetages** (packages)

**Définition** : un package regroupe des classes ayant un lien logique.

*exemple : un ensemble de classes permettant de manipuler les liaisons réseaux, les protocoles internet ...*

En plus d'un langage, Java comprend une **API** (Application Programming Interface) à travers de nombreux packages.

# Survol – (2) Orienté Objet : API

**java.lang** : classes essentielles

objet, type de bases, processus

**java.util** : structures de données

listes, ensembles, arbres, ...

**java.awt** et **java.swing** : interfaces graphiques

fenetres, boutons, ...

**java.io** : entrées / sorties

gestion des fichiers, des affichages à l'écran, ...

**java.net** : réseau

URL, sockets, ...

**java.sql** : JDBC

gestion des bases de données (accès, requêtes, ...)

...

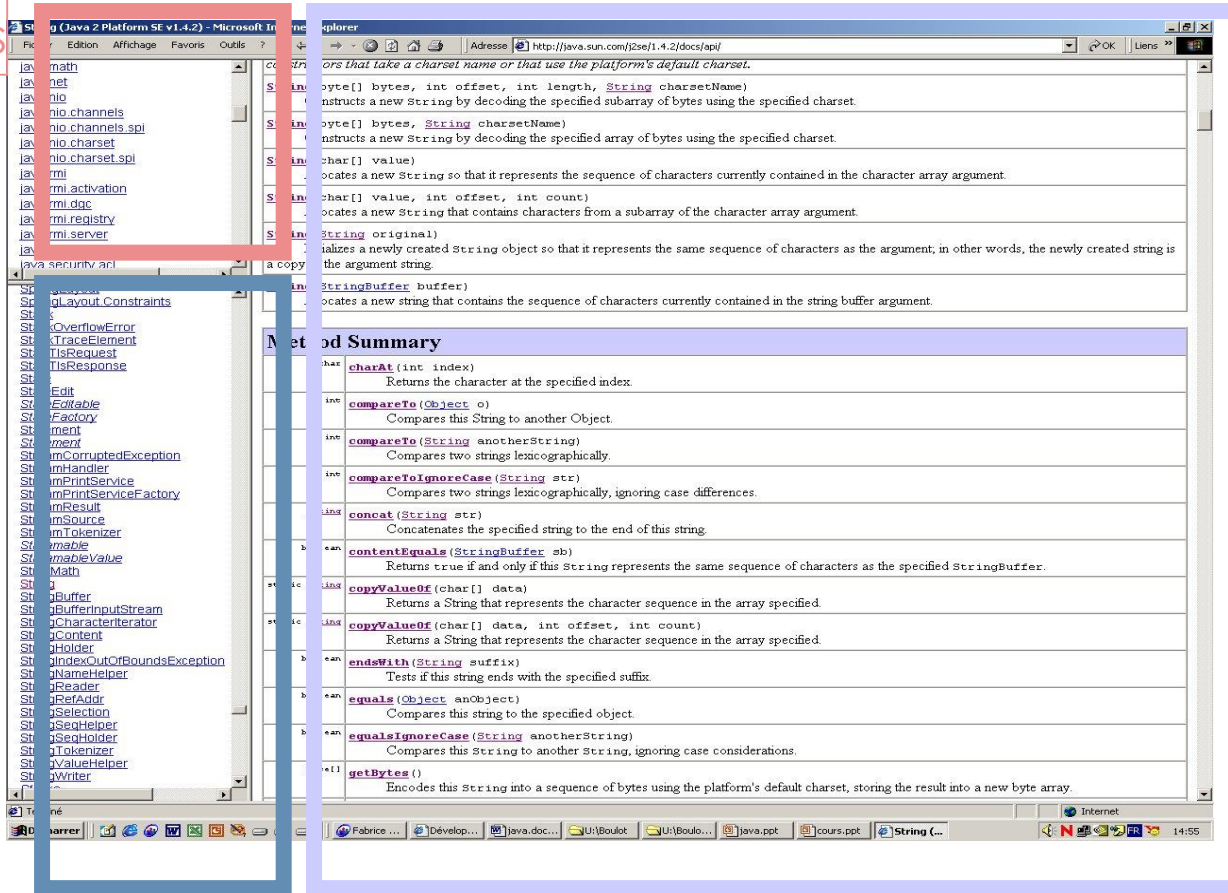
**javax.java3D** : graphique 3D

API  
Standard

# Survол – (2) Orienté Objet : API

packages

classes



The screenshot displays the Java 2 Platform SE v1.4.2 - Microsoft Internet Explorer window. The left pane shows the package hierarchy with 'java.lang' selected. The right pane shows the String class documentation, including the 'Method Summary' section.

**Method Summary**

Method	Description
<code>charAt(int index)</code>	Returns the character at the specified index.
<code>compareTo(Object o)</code>	Compares this String to another Object.
<code>compareTo(String anotherString)</code>	Compares two strings lexicographically.
<code>compareToIgnoreCase(String str)</code>	Compares two strings lexicographically, ignoring case differences.
<code>concat(String str)</code>	Concatenates the specified string to the end of this string.
<code>contentEquals(StringBuffer sb)</code>	Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.
<code>copyValueOf(char[] data)</code>	Returns a String that represents the character sequence in the array specified.
<code>copyValueOf(char[] data, int offset, int count)</code>	Returns a String that represents the character sequence in the array specified.
<code>endsWith(String suffix)</code>	Tests if this string ends with the specified suffix.
<code>equals(Object anObject)</code>	Compares this string to the specified object.
<code>equalsIgnoreCase(String anotherString)</code>	Compares this String to another String, ignoring case considerations.
<code>getBytes()</code>	Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

Description

Données  
(attributs)

Opérations  
(méthodes)

Site web officiel pour la documentation de JDK 10

<https://docs.oracle.com/javase/10/docs/api/index.html?overview-summary.html>

# Survol – (3) Réparti

---

## Langage pour applications réparties

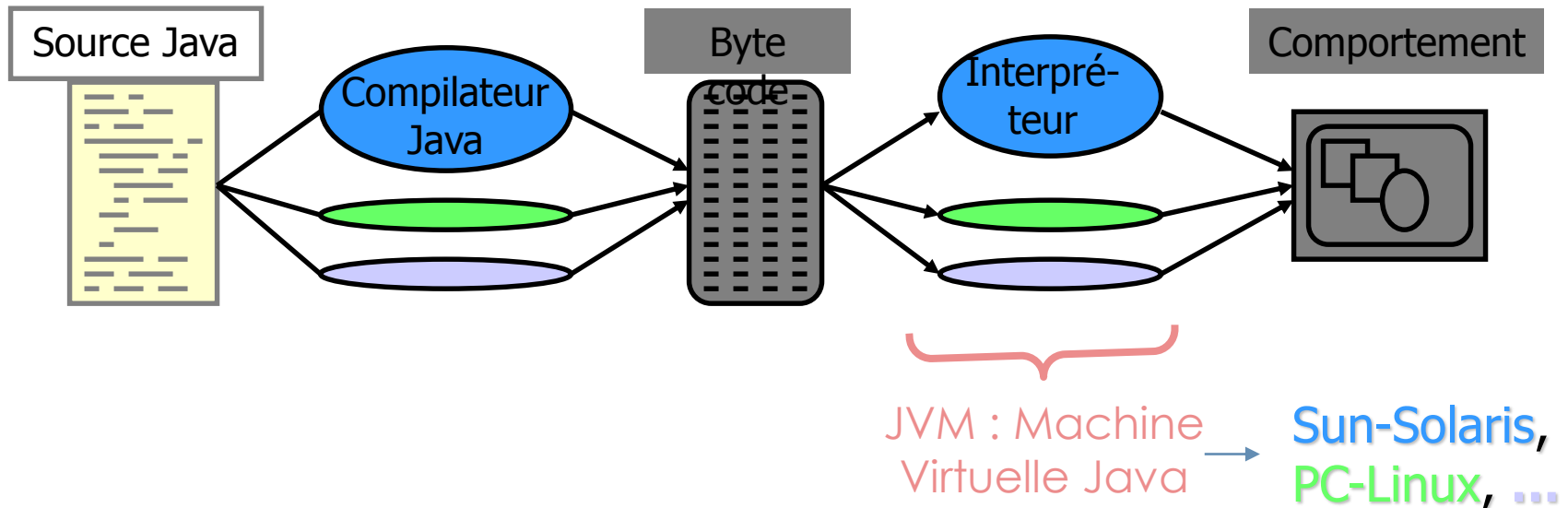
- Avec paquetage `java.net` aussi facile d'ouvrir un fichier local que distant
- Avec paquetage `java.rmi`, on peut faire communiquer deux classes situées sur deux machines distantes

Exemple : la Classe « Socket » permet créer des applications de type client/serveur

# Survol – (4) Interprété

Par opposition à **compilé** (ex : C, C++, Ada, ...)

En fait le source est : compilé, puis interprété ...



Portable mais plus lent ...

# Survol – (5) Robuste

Rappel : Destiné à l'électronique grand public ... robuste !

- **Java** élimine certains type d'erreur de programmation,
- est **fortement typé** (pas de conversion automatique risquant une perte de données),
- est fiable (gestion des pointeurs et de la mémoire),
- supporte la gestion des **exceptions** (récupération erreurs).

**Définition** : une exception est comme un signal indiquant que quelque chose d'exceptionnel est arrivé

Remarque : Dans tous les langages on peut et on **DOIT** gérer les exceptions : ^ (

*Exemple : Saisie d'un nombre*



# Survol – (6) Sûr ... au sens de la sécurité

---

Important car environnement distribué !

Concerne **les applets** ... pour les autres programmes, la question ne se pose pas.

aucun programme ne peut ouvrir, lire, écrire ou effacer un fichier sur le système de l'utilisateur

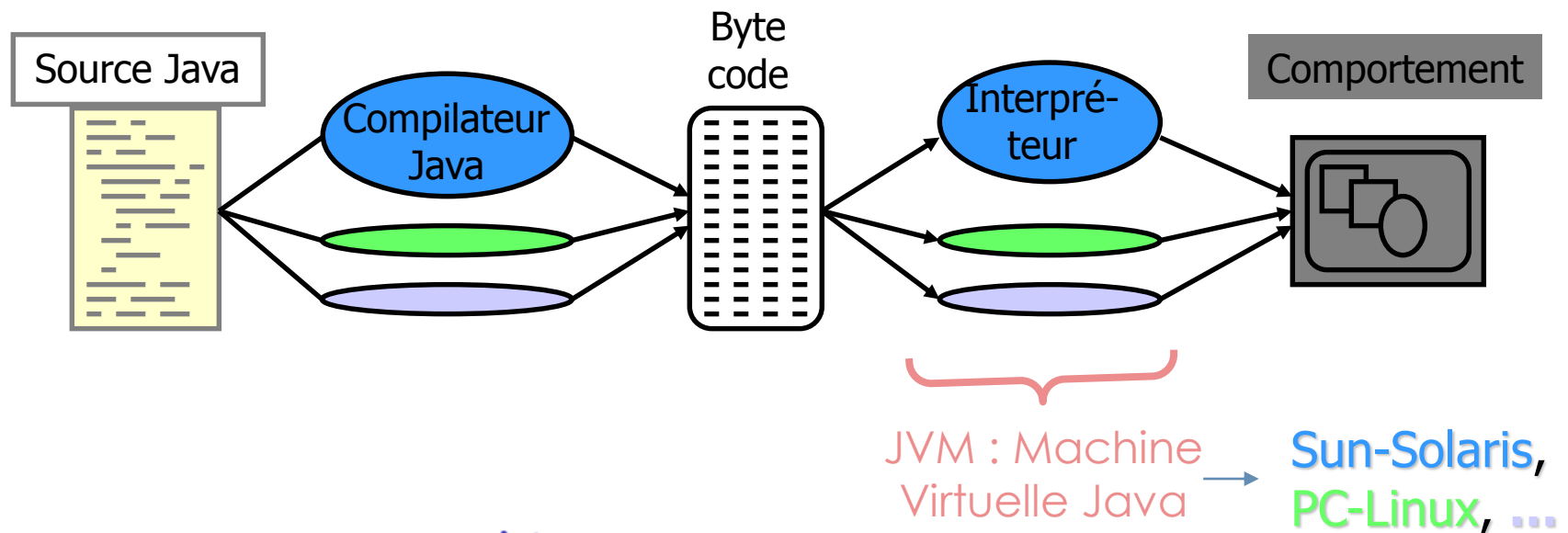
aucun programme ne peut lancer un autre programme sur le système de l'utilisateur

toute fenêtre créée par le programme est clairement identifiée comme étant une fenêtre Java, ce qui interdit par exemple la création d'une fausse fenêtre demandant un mot de passe

les programmes ne peuvent pas se connecter à d'autres sites Web que celui dont ils proviennent

# Survol – (7) Indépendant des AM

Découle du point 4 (Interprété) :



**WORA**  
(Write Once, Run Anywhere)

# Survol – (8) Portable

---

- ✓ Conséquence de l'indépendance de l'A.M.
- ✓ Java s'assure que rien dans les spécifications du langage ne dépend de la plateforme d'implémentation

Exemple : La taille de tous les types primitifs est fixe. Windows 98 stocke ses entiers sur 32 bits, Windows XP ou un DEC Alpha sur 64 ... Java définit une unique taille (32 bits) pour toutes les plateformes.

Exemple : Java définit des composants « abstraits ». L'Abstract Windowing Toolkit (java.awt) définit des «Windows» correspondant aussi bien aux fenêtres MacOS qu'à celles de XP.

# Survol – (9) Performant

---

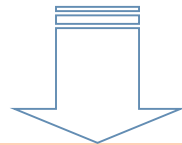
- Performant ... pour un langage interprété (jusqu'à 20x plus lent que du C)
- Suffisant pour applications interactives, avec IG, et basées sur le réseau
- Pas toujours suffisant tout de même (Systèmes critiques) : dans ce cas langage de bas niveau (C, C++, Fortran)
- Pour rendre Java plus rapide :
  - Compilateurs (perte de la portabilité)

# Survol – (10) Multitâche

Imaginons :

Alors que vous êtes en train de **télécharger** un fichier, vous visualisez un document sous Word et vous **utilisez l'ascenseur** tout en **écoutant** un morceau de musique !

- |             |   |                                |
|-------------|---|--------------------------------|
| 1. Loader   | } | • 3 « choses » en même temps   |
| 2. Scroller |   | • 3 pseudo-processus           |
| 3. Player   |   | • 3 threads (java.lang)        |
|             |   | • 3 unités d'exécution isolées |



Un programme Java peut faire plusieurs choses simultanément.

Rmq : en mono processeur, c'est du pseudo-parallélisme

# Survol – (11) Dynamique

---

- Adaptation à un environnement en constante évolution
- Java s'adapte à l'évolution du système sur lequel il s'exécute. Les classes sont chargées au fur et à mesure des besoins, à travers le réseau s'il le faut. Les mises à jour des applications peuvent se faire classe par classe sans avoir à recompiler le tout en un exécutable final.
- Techniquement : il charge les classes en cours d'exécution !

- Version 10.0.2, Java 10
- Une version standard (J2SE)
- Une version entreprise (J2EE)
- Pour développer :
  - le JDK (Java Developer Kit) est gratuit
  - les spécifications détaillées (API) :  
<https://docs.oracle.com/javase/9/>
  - + de 3000 classes
  - + les API entreprises, web, xml ...
  - + toutes les contributions OpenSource

## Java présent dans de très nombreux domaines d'application

- Serveurs d'application (Java Entreprise)
- Systèmes embarqués
- Cartes à puces
- Domotique
- ...





# Plan du cours 1/4



## ***Programmation Orienté Objet***

1. Introduction
  - a. Introduction à la POO
  - b. Classes et objets
  - c. Origines du langage JAVA
  - d. Survol du langage, ses propriétés
  - e. Votre premier programme « Hello World »

# Créer un programme

---

Pour créer un programme, il faut :

- Un éditeur de texte pour écrire le **code source**
- Deux outils de développement au minimum :
  - Un compilateur : **javac**
  - Un interpréteur d'application Java : **java**
- D'autres outils de développement :
  - Un interpréteur d'applets java : **appletviewer**
  - Un débogueur java : **jdb**
  - Un décompilateur : **javap**
  - Un générateur de documentation : **javadoc**
  - etc. ...

# Créer un programme

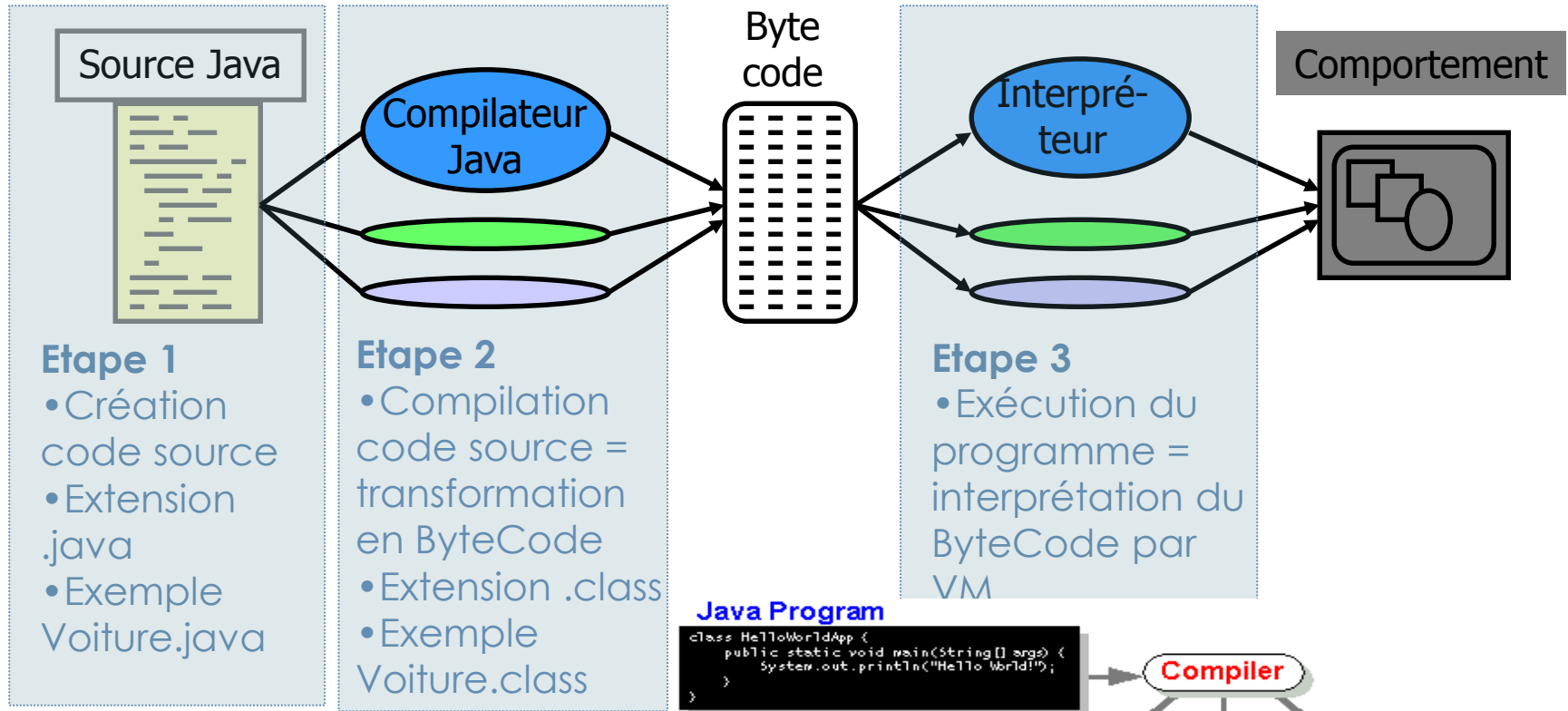
---

Il existe des environnements de développements comprenant l'éditeur ET les outils (Eclipse, JBuilder, Kawa, Forte, ...)

Avantages : Deux en un, coloration syntaxique du code, complétion automatique, GUI Builder ...

ATTENTION : ne pas confondre pour autant l'environnement ET le compilateur.

# Les étapes de développement

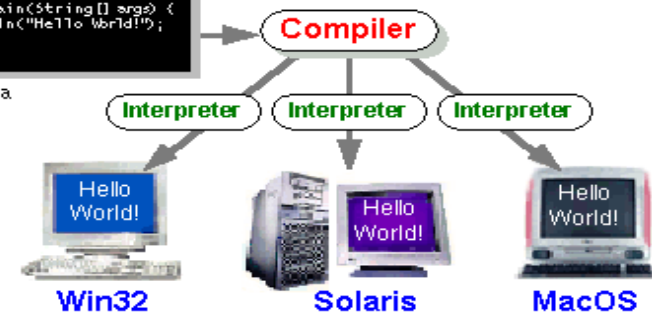


## Java Program

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

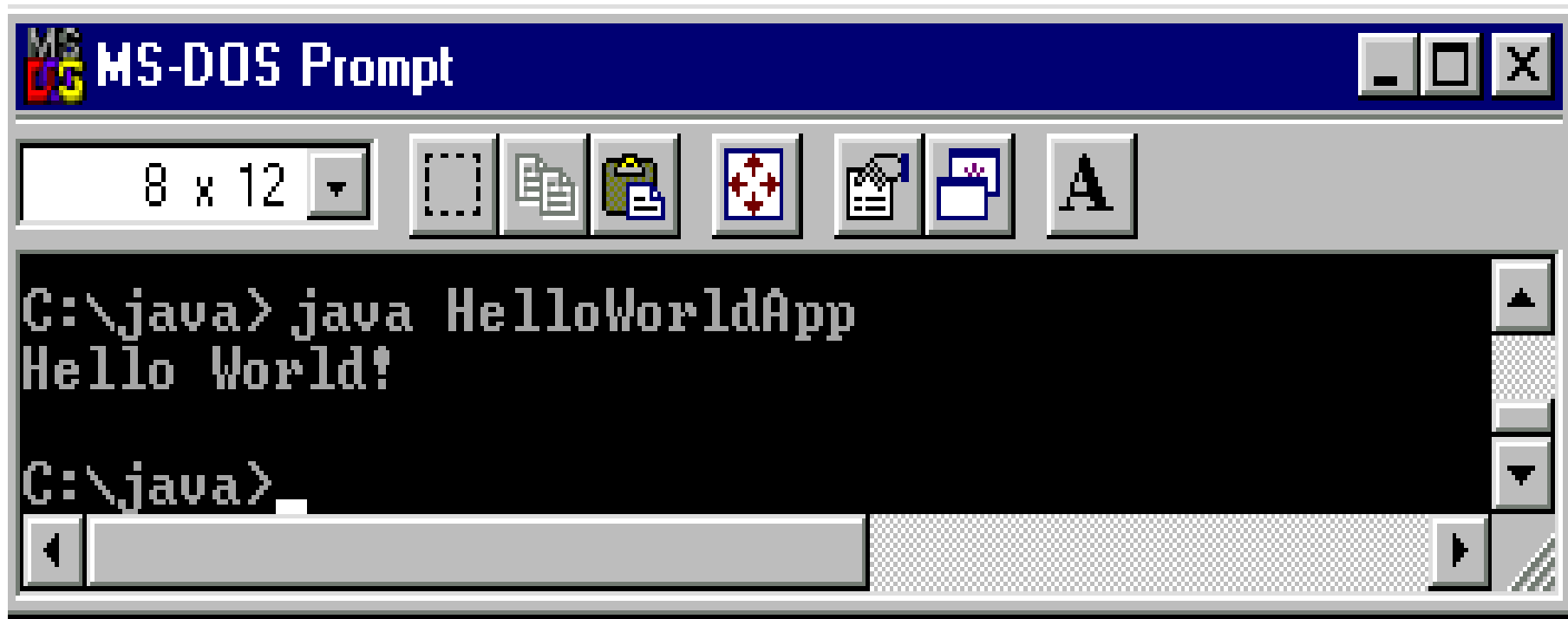
HelloWorldApp.java

**bytecode**  
ni exécutable  
ni lisible par l'homme



# Mon premier programme

HelloWorld !

A screenshot of a Windows 95-style MS-DOS Prompt window. The title bar is blue with the text "MS-DOS Prompt" and standard window controls. Below the title bar is a toolbar with icons for font size (8 x 12), font face, bold, italic, underline, strikethrough, and background color. The main area is black with white text. It shows the command "C:\java> java HelloWorldApp" being entered, followed by the output "Hello World!". The prompt "C:\java>" is shown again on the next line. A scroll bar is visible on the right side of the text area.

```
MS-DOS Prompt
8 x 12
C:\java> java HelloWorldApp
Hello World!
C:\java>
```



## Créer un fichier source en JAVA

### 1. Démarrer Editeur et taper le code Java suivant :

```
/**
 * La classe HelloWorldApp
 */
public class HelloWorldApp {
    public static void main(String[] args) {
        /* Affiche "Hello World!" */
        System.out.println("Hello World!");
    }
}
```

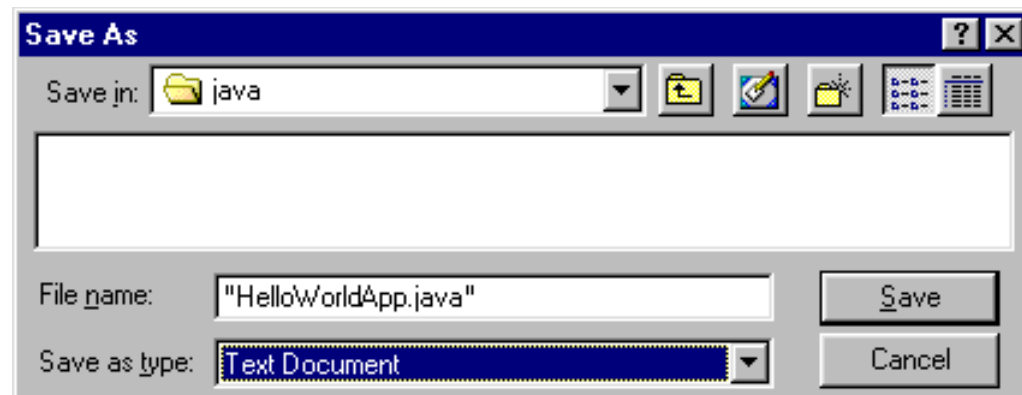
Remarque : Java est case-sensitive !

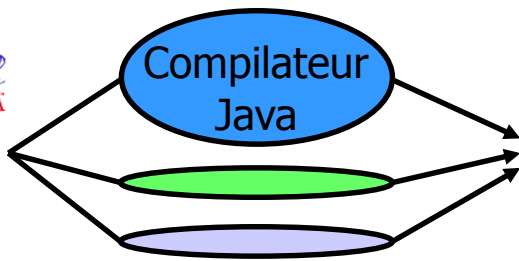


# Etape n°1 (suite et fin)

## 2. Enregistrer le fichier

- Nom du fichier = `nomDeLaClasse.java`
- Un fichier par classe et une classe par fichier



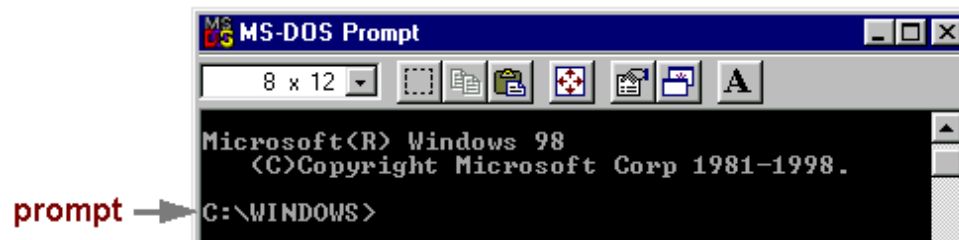


## Etape n°2

### Compiler le code source → transformer en *bytecode*

#### ➤ Lancer l'invite de commande

Menu Démarrer -> Programme -> Invite de Commande



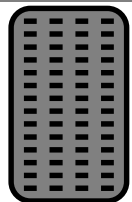
#### ➤ Aller dans le répertoire contenant le code source

#### ➤ Taper : `javac HelloWorldApp.java`

argument

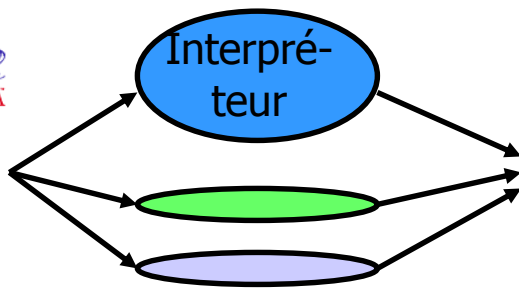
commande

Byte code



HelloWorldApp.class





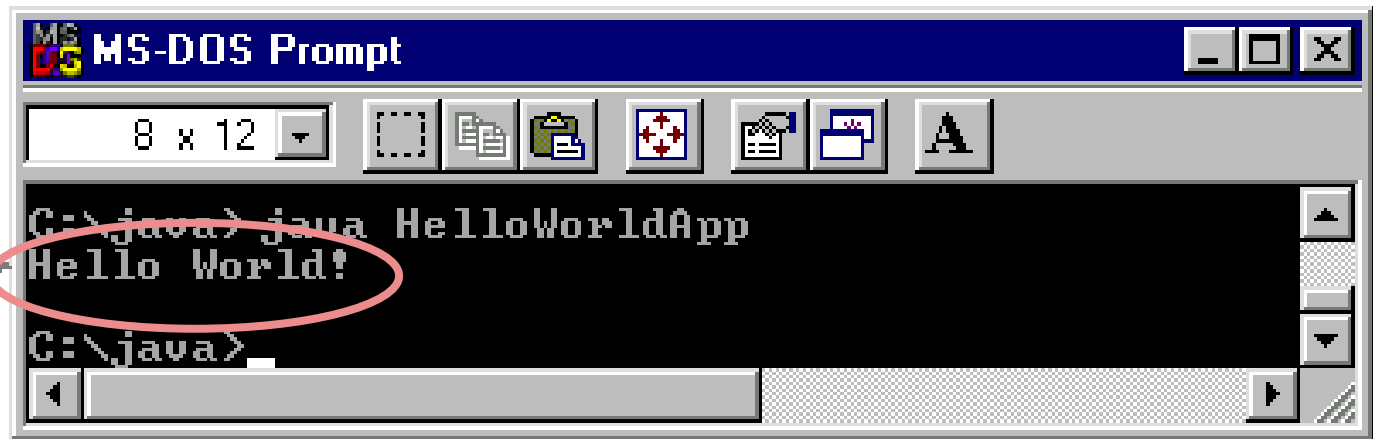
## Etape n°3

Exécuter le programme → interpréter le *bytecode*

Dans le même répertoire faire

> `java HelloWorldApp`

result →



```

MS-DOS Prompt
8 x 12
C:\java> java HelloWorldApp
Hello World!
C:\java>
  
```

# Juste pour le voir une fois ...

Créer une applet à partir de cet exemple est aussi simple que ça :

```
import java.awt.*;
import java.applet.*;

public class HelloWorld extends Applet {
    public void init() {
        add(new Label("Hello World !"));
    }
}
```

```
<html>
  <head>
    <title> A Simple Program </title>
  </head>
  <body>Here is the output of my program:
    <applet code="HelloWorld.class">
    </applet>
  </body>
</html>
```

# Allez encore une fois !

**Bonjour !!!**



```

Bonjour.java

import java.awt.Graphics;
import java.awt.Font;
import java.awt.Color;
import java.awt.Image;

public class Bonjour extends java.applet.Applet {
    Font f = new Font("TimesRoman", Font.BOLD, 36); //def de ma police
    Image duke; // def d'une image
    Color jaunepale = new Color(255,255,200); //def d'une couleur

    public void init(){
        duke = getImage(getCodeBase(), "duke.gif");
    }

    public void paint(Graphics screen){

        setBackground(jaunepale); // le fond devient jaune pale

        int iWidth = duke.getWidth(this);
        int iHeight = duke.getHeight(this);
        screen.drawImage(duke, 10, 35, iWidth, iHeight, this);

        screen.setFont(f);
        screen.setColor(Color.red);
        screen.drawString("Bonjour !!!", 10, 30);
    }
}

```

*Questions*



# Plan du Cours



## ***JAVA***

1. Syntaxe de base
2. Classes et objets
3. E/S
4. Héritage/Polymorphisme
5. Normes de développement
6. Exceptions

# Plan du Cours



## ***JAVA***

1. Syntaxe de base
2. Classes et objets
3. E/S
4. Héritage/Polymorphisme
5. Normes de développement
6. Exceptions

- Java est sensible à la casse
- Les instructions se terminent par un ;

```
Voiture v;
```

- Les blocs d'instructions sont encadrés par des accolades { }

```
if (v.vitesse > 130) {  
    v.vitesse--;  
    System.out.println("Vitesse > 130!");  
}
```

# Les commentaires

- Commentaire sur une ligne : //

```
Voiture v; // commentaire sur une ligne
```

- Commentaire sur plusieurs lignes : /\* \*/

```
if (v.vitesse > 130) {  
    /* Ce commentaire commence ici  
    v.vitesse--;  
  
    et se termine là */  
    System.out.println("Vitesse > 130!");  
}
```



# Les Types Elémentaires

---

- Entiers
  - byte : -128 à 127
  - short : -32768 à 32767
  - int : -2147483648 à 2147483647
  - long : -9223372036854775808 à 9223372036854775807
- Réels
  - float : 1.401e-045 à 3.40282e+038
  - double : 2.22507e-308 à 1.79769e+308
- Valeurs logiques
  - boolean : true ou false
- Caractère unique
  - char

# Les Types Elémentaires

---

## Caractéristiques

Occupent une place fixe en mémoire ;

Ne sont pas des objets ;

N'ont pas besoin d'être construits ;

**Attention** : certaines classes de la JDK ont le même nom à une majuscule près.

Exemple : Double, Float ...

# Opérations sur les types élémentaires

---

- Arithmétiques

Unaires :  $-b$

Binaires :  $a+b$   $a-b$   $a*b$   $a/b$   $a\%b$

Incrémentation :  $a++$   $b--$

- Comparaisons

$a==b$   $a!=b$   $a>b$   $a<b$   $a>=b$   $a<=b$

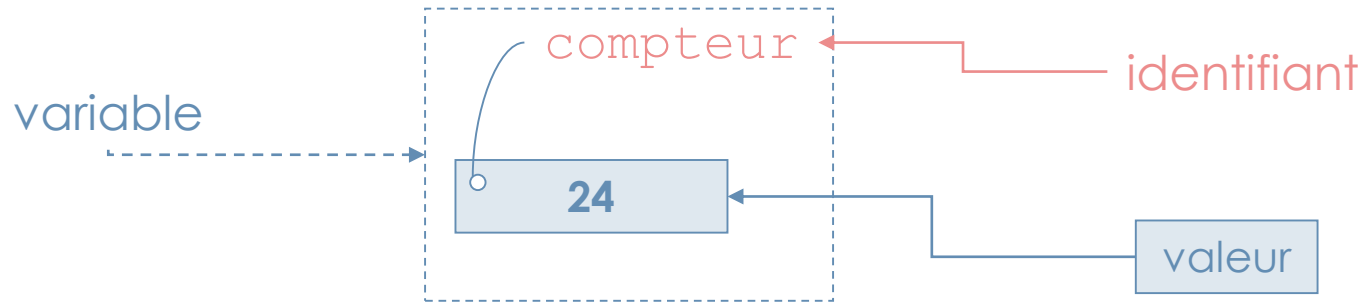
- Logiques

And :  $a\&\&b$

Or :  $a||b$

# Déclaration de variables

Chaque **variable** est associée à un **identifiant** qui permet de manipuler sa **valeur** dans le programme.



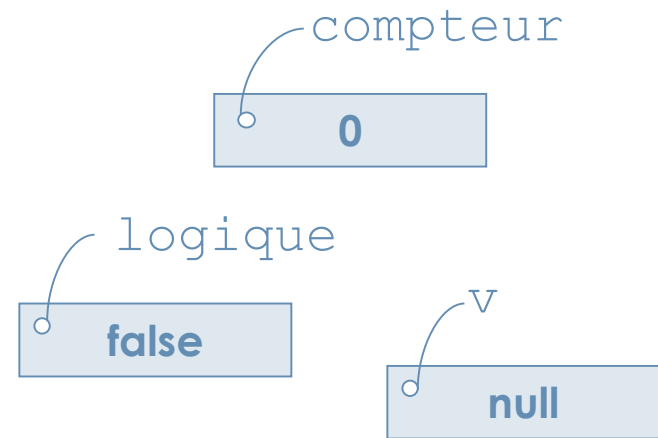
```
int compteur; // variable de type entier  
boolean logique; // variable de type booléen  
Voiture v; // variable référençant un objet
```

# Sauf un mot-réservés du langage

<code>abstract</code>	<code>default</code>	<code>goto</code>	<code>null</code>	<code>synchronized</code>
<code>boolean</code>	<code>do</code>	<code>if</code>	<code>package</code>	<code>this</code>
<code>break</code>	<code>double</code>	<code>implements</code>	<code>private</code>	<code>throw</code>
<code>byte</code>	<code>else</code>	<code>import</code>	<code>protected</code>	<code>throws</code>
<code>case</code>	<code>extends</code>	<code>instanceof</code>	<code>public</code>	<code>transient</code>
<code>catch</code>	<code>false</code>	<code>int</code>	<code>return</code>	<code>true</code>
<code>char</code>	<code>final</code>	<code>interface</code>	<code>short</code>	<code>try</code>
<code>class</code>	<code>finally</code>	<code>long</code>	<code>static</code>	<code>void</code>
<code>continue</code>	<code>float</code>	<code>native</code>	<code>super</code>	<code>volatile</code>
<code>const</code>	<code>for</code>	<code>new</code>	<code>switch</code>	<code>while</code>

# Déclaration de variables

**Remarque** : pour les types simples, la déclaration d'une variable suffit à **réserver la mémoire** nécessaire au stockage de ses valeurs sans que l'initialisation par défaut soit systématique. Pour les objets, l'**allocation** à lieu plus tard (cf. suite du cours).



# Les structures de contrôle

---

- L'exécution d'un programme Java est **séquentiel** : les instructions sont lues et exécutées dans l'ordre de déclaration
- On utilise des **structures de contrôle** :
  - les boucles
  - et les conditions

Permet d'exécuter ou non un bloc d'instructions.

```
if (condition){  
    /* bloc d'instructions lorsque la  
    condition est vraie */  
} else {  
    /* bloc d'instructions lorsque la  
    condition est fausse */  
}
```

Rmq : la condition est une expression booléenne (l'évaluation de son résultat rend vrai ou faux)



# L'opérateur switch

- Variante du if ... else ...
- Permet d'éviter une imbrication de else if ...
- Fonctionne sur les types « discrets » (en fait : byte, int, short, char, boolean)

```
switch (expression) {  
    case valeur1 : /* bloc instructions */  
                    break;  
    case valeur2 : /* bloc instructions */  
                    break;  
    ...  
    default : /* bloc instructions */  
              break;  
}
```

Permet d'exécuter plusieurs fois un bloc d'instructions donné (nombre d'itérations **connu**)

```
for
(initialisation;condition_arret;modification){
    /* bloc d'instructions lorsque la
    condition d'arrêt n'est pas vérifiée */
}
```

```
/* typiquement : */
for (i=0; i<10; i++){
    /* traitement */
}
```

# Boucle while

Permet d'exécuter plusieurs fois un bloc d'instructions donné (nombre d'itérations **inconnu**)

```
while (condition) {  
    /* bloc d'instructions lorsque la  
    condition  
    est vérifiée */  
}
```

Rmq : la condition est une expression booléenne (l'évaluation de son résultat rend vrai ou faux)

# Affectation, comparaison

---

Ne pas confondre :

affectation : =

comparaison : <, >, !=, ==

## Déclaration

```
type[] nomVariable;  
type nomVariable[];
```

## Dimensionnement

```
nomVariable = new type[5];
```

## Longueur disponible par

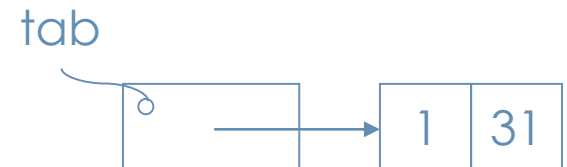
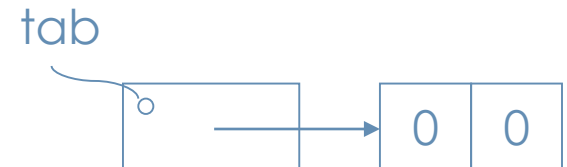
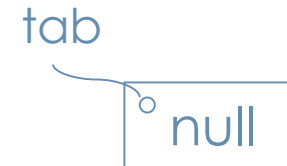
```
nomVariable.length
```

## Exemple avec un type primitif

```
/* déclaration */
int[] tab;
```

```
/* dimensionnement */
tab = new int[2];
```

```
/* initialisation */
tab[0] = 1;
tab[1] = 31;
```

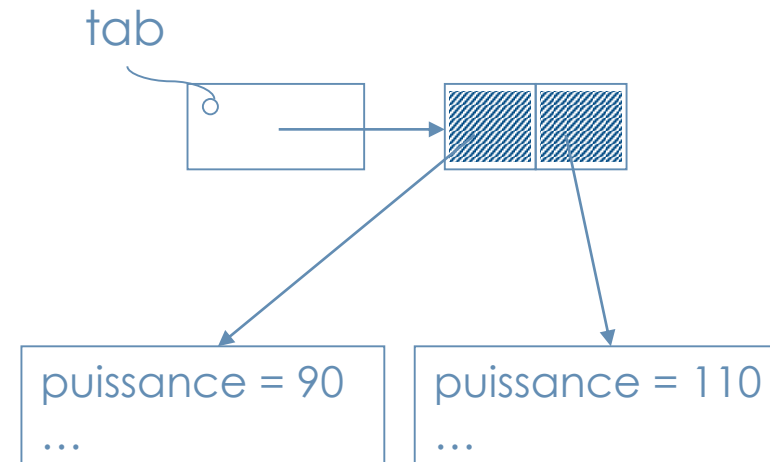
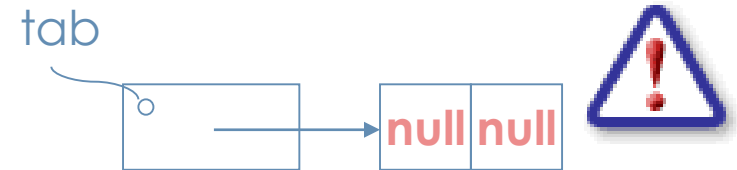
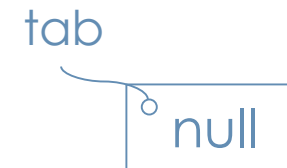


## Exemple avec un objet

```
/* déclaration */
Voiture[] tab;
```

```
/* dimensionnement */
tab = new Voiture[2];
```

```
/* initialisation */
tab[0] = new Voiture(90);
tab[1] = new Voiture(110);
```



## *Java : Syntaxe de base*

*Questions*





# Plan du cours



## ***JAVA***

1. Syntaxe de base
2. **Classes et objets**
3. E/S
4. Héritage/Polymorphisme
5. Normes de développement
6. Exceptions

# Rappel : Classe

- Sert à définir les possibilités des **objets** d'un type donné
- Décrit un ensemble de **données** et les **opérations** sur ces données

## Spécification UML

données

Voiture	
- puissance : entier	opérations
- est_démarrée : booléen	
- vitesse : réel	
+ deQuellePuissance () : entier	
+ démarre ()	opérations
+ accélère (réel)	

opérations

# Déclaration de Classe

```
public class Voiture { // déclaration classe
    /* Variables d'instance */
    private int puissance;
    private boolean est_demarre;
    private double vitesse;

    /* Méthodes */
    public int deQuellePuissance () {
        return this.puissance;
    }
    public void demarre () {
        est_demarre = true;
    }
    public void accelere (double v) {
        if (est_demarre) {
            this.vitesse = this.vitesse + v;
        }
    }
}
```

← Code Java

Spécification UML



**Voiture**

- puissance : entier
- est_démarrée : booléen
- vitesse : réel
+ deQuellePuissance () : entier
+ démarre ()
+ accélère (réel)

# Rappels : Objet

---

- Un objet est l'**instance** d'une classe.
- Une classe peut être **instanciée** plusieurs fois pour créer autant d'objets que désirés.
- Chaque objet possède ses propres variables d'instance et est **indépendant des autres objets**.

# Objet : Cycle de vie

---

- Création
  - Utilisation d'un constructeur
  - Objet créé en mémoire
  - Attributs (*ie* variables d'instance) initialisés
- Manipulation
  - Utilisation des méthodes
  - Modification des attributs
  - Consultation des attributs
- Destruction
  - L'objet n'est plus référencé, la mémoire est libérée

# Objet : Déclaration

- La manipulation d'un objet se fait grâce à une variable : chaque variable est déclarée

```
/* déclaration de la variable ma_voiture
*/
Voiture ma_voiture;
```

- C'est sa **référence**
- Après la déclaration, l'objet n'existe pas (il vaut la valeur **null**), la place mémoire n'est pas réservée



**Attention** : comme l'objet n'existe pas, l'appel à `ma_voiture.demarre()` provoque une **erreur d'exécution** `NullPointerException`

# Objet : Déclaration

---

**IMPORTANT**  
**(même si on se répète)**

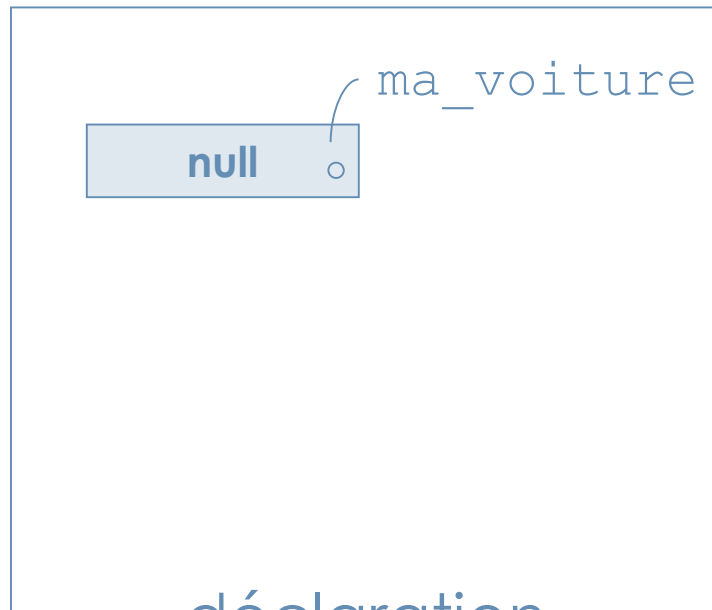
Une référence n'est pas un objet !

Une référence est **SIMPLEMENT** un nom  
qui permet d'accéder à un objet !

# Objet : Création

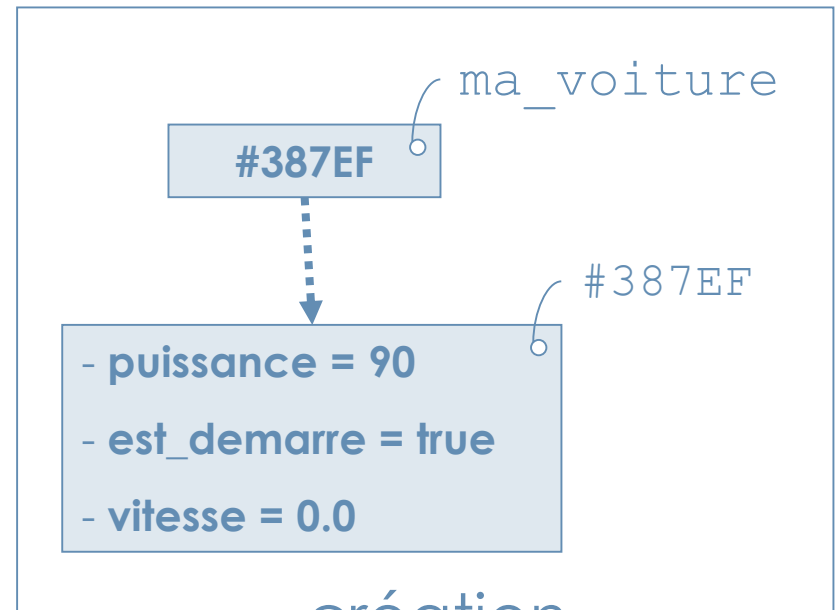
En Java, les objets doivent être alloués dynamiquement : opérateur **new**.

```
/* déclaration */
Voiture ma_voiture;
```



déclaration

```
/* création */
ma_voiture = new Voiture();
```



création

```
/* déclaration + création */
Voiture ma_voiture = new Voiture();
```



# Objet : Constructeur

- On crée un objet avec l'opérateur **new**
- **new** fait appel à une méthode particulière de la classe : le **constructeur**
- Constructeur :
  - porte le nom de la classe
  - alloue la mémoire nécessaire
  - initialise les variables d'instance

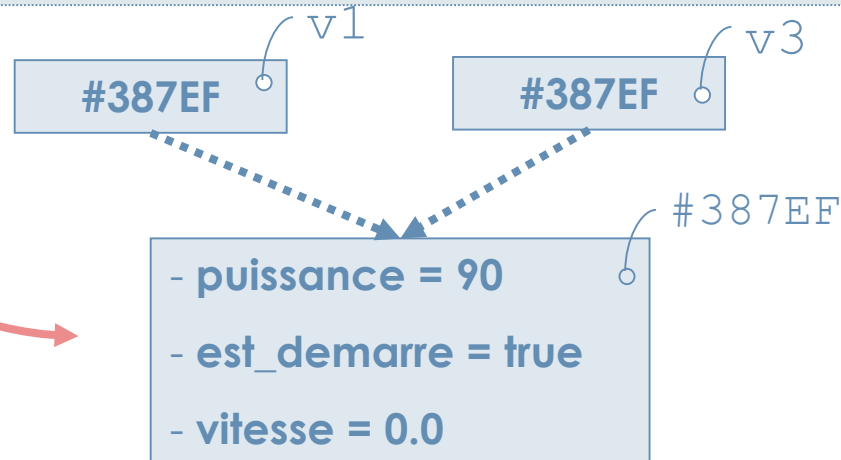
```
public class Voiture {  
    [...]  
    /* Constructeur */  
    public Voiture () {  
        this.puissance = 90;  
        this.est_demarre = true;  
        this.vitesse = 0;  
    }  
    [...]  
}
```

# Objet : Exemple

```
Voiture v1;    // référence vers un objet de la classe Voiture
Voiture v2,v3; // deux autres références

/* A ce stade du programme aucun objet n'a été créé */


v1 = new Voiture (); // création d'un objet Voiture
                       // désigné par la référence v1
v2 = new Voiture (); // création d'un autre objet Voiture
                       // désigné par la référence v2
v3 = v1;              // v3 est un deuxième nom pour désigner
                       // l'objet déjà référencé par v1
```



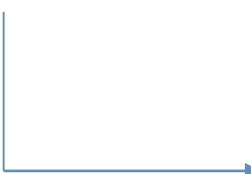
# Objet : Ramasse-miettes

Contrairement à d'autres langages (ex: C++), la mémoire est libérée automatiquement par le Ramasse-miettes (Garbage Collector), lorsque :

- plus aucune variable ne référence l'objet
- le bloc dans lequel l'objet est défini se termine
- l'objet est affecté à la valeur **null**



```
if (condition) {  
    /* allocation de la mémoire */  
    Voiture ma_voiture = new Voiture();  
    [...]  
} /* libération de la mémoire */
```



```
/* allocation de la mémoire */  
Voiture ma_voiture = new Voiture();  
[...]
```

```
/* libération de la mémoire */  
ma_voiture = null;
```

# Objet : Remarque

Pour les types élémentaires, la déclaration coïncide avec la création

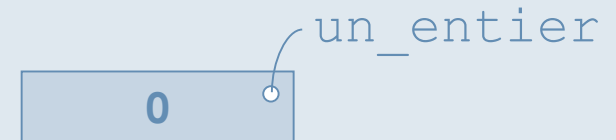
```
Voiture ma_voiture;
```



```
ma_voiture.demarre();
```

Résultat : `NullPointerException`

```
int un_entier;  
// int un_entier = 0;
```



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

Résultat : `C:\java> 0`

# Objet : Accès aux attributs

Pour accéder aux données d'un objet on utilise une notation pointée :

*nomDObjet.nomDAttribut*  
*nomDObjet.nomDeMethode*

```
/* déclaration et création */
Voiture v1 = new Voiture();
Voiture v2 = new Voiture();
Voiture v3 = v1; // v1 et v3 référencent le même objet

/* accès aux attributs en écriture */
/* ces deux instructions modifient le même objet !!! */
v1.puissance = 110;
v3.est_demarre = false ;

/* accès aux attributs en lecture */
System.out.println ("Puissance de v1 = " + v1.puissance);
```

# Objet : Appel de méthode

- Pour "demander" à un objet d'effectuer une opération (i.e. exécuter l'une de ses méthodes) il faut lui envoyer un message
- Un message est composé de trois parties
  - La référence désignant l'objet à qui le message est envoyé
  - Le nom de la méthode à exécuter (définie dans la classe de l'objet)
  - Les éventuels paramètres de la méthode
- Syntaxe

*nomDeObjet.nomDeMethode(<liste de paramètres>)*

```
/* appel de la méthode accélère */  
ma_voiture.accelere (10.0);
```

# Objet : Appel de méthode

```
public class Voiture {
    [...]
    /* Méthodes */
    public int deQuellePuissance () {
        return puissance;
    }
    public void démarre () {
        est_demarre = true;
    }
    public void accelere (double v) {
        if (est_demarre) {
            vitesse = vitesse + v;
        }
    }
}
```

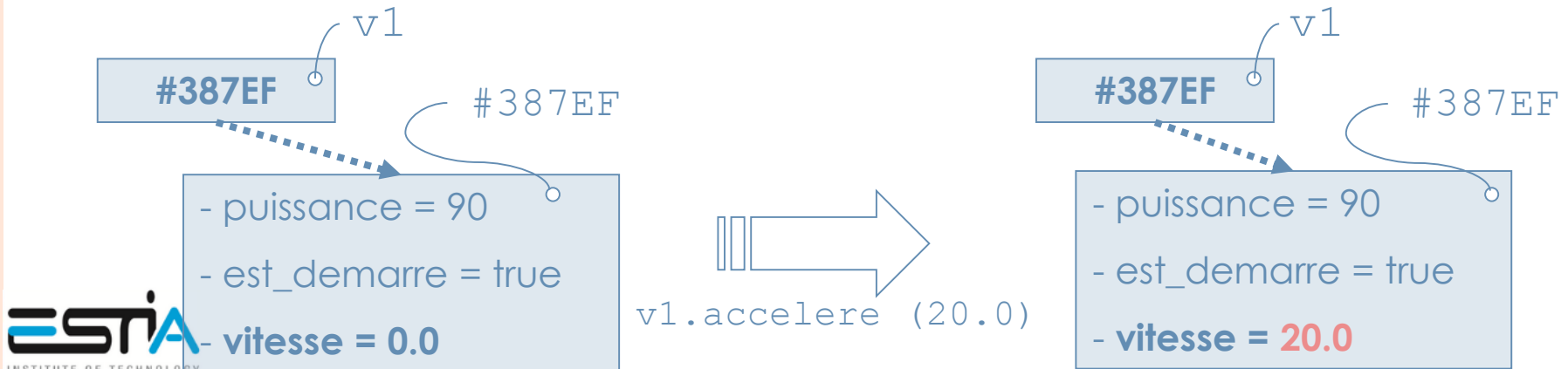
Définition

Appel

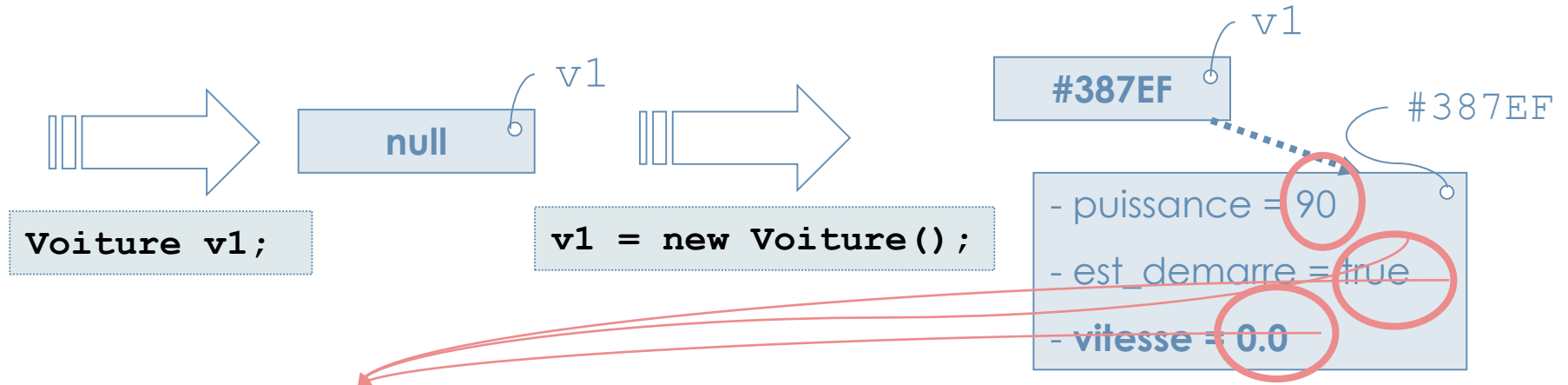
```
/* Création d'un objet de
la classe Voiture */
Voiture v1 = new Voiture();

/* Modification de la vitesse
par la méthode accelere */
v1.accelere(20.0);

/* Accès à la puissance par
la méthode deQuellePuissance */
System.out.println ("Puissance="
+ v1.deQuellePuissance());
```

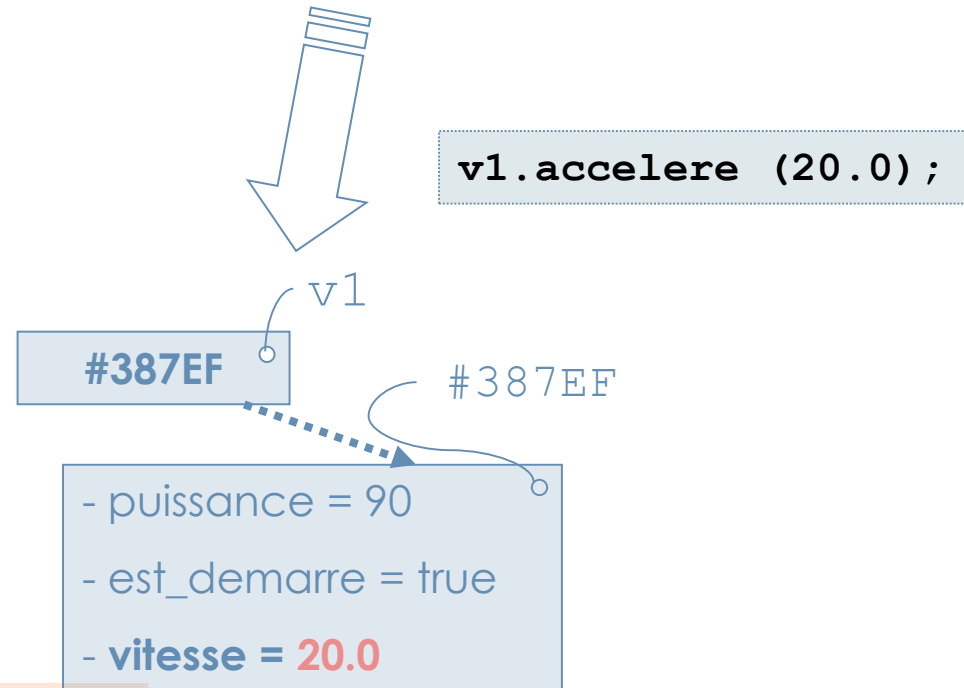


# Objet : Appel de méthode



Viennent du constructeur

```
/* Constructeur */
public Voiture () {
    puissance = 90;
    est_demarre = true;
    vitesse = 0;
}
```





# Objet : Passage de paramètre

- Rappels - 2 situations :  
Définition d'une méthode  
Appel d'une méthode
- Exemple en math :  
Définition :  $f(x) = 2 * x$  , pour tout  $x$  appartenant à  $N$   
  
Utilisation :  $f(2) = 2 * 2$ ,  $f(8) = 2 * 8$ , ...  
On remplace le paramètre formel ( $x$ ) par une valeur

c'est juste un nom

# Objet : Passage de paramètre

---

A l'appel d'une méthode, en java, la valeur du paramètre est recopiée

Passage de paramètre **par valeur**

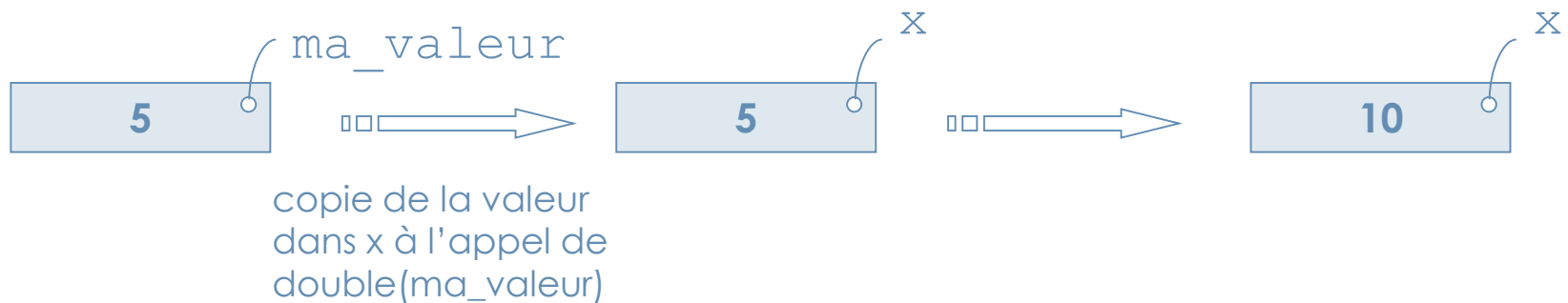
# Objet : Passage de paramètre

## Types simples

```
public class Math {
    [...]
    /* Définition d'une méthode */
    public void double (int x) {
        x = 2*x;
    }
}
```

```
[...]
Math m = new Math();

int ma_valeur = 5;
/* Appel d'une méthode */
m.double (ma_valeur);
```



La variable `ma_valeur` reste inchangée  
L'instruction est sans effet ...

# Objet : Passage de paramètre

## Types simples

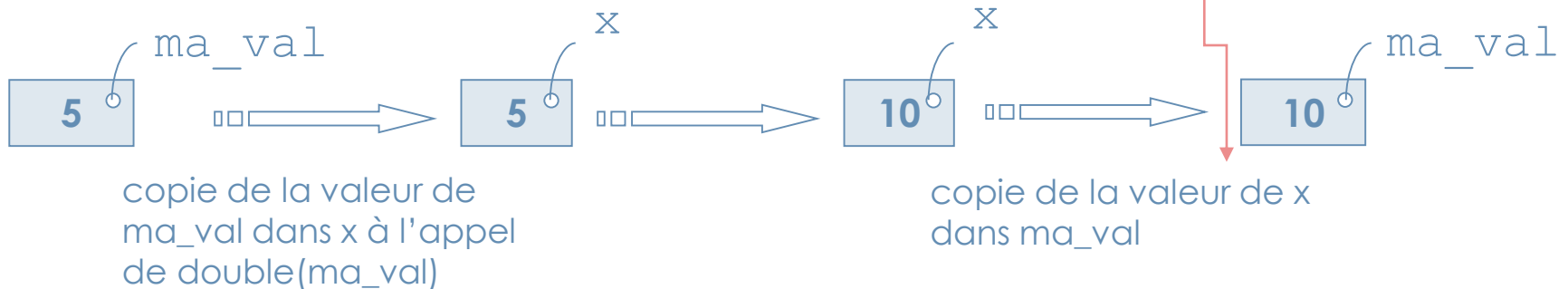
```
public class Math {
[...]
```

/\* Définition d'une méthode \*/

```
public int double (int x) {
    x = 2*x;
    return x;
}
```

```
[...]
Math m = new Math();

int ma_val = 5;
/* Appel d'une méthode */
ma_val = m.double (ma_val);
```



La variable `ma_val` est modifiée

# Objet : Passage de paramètre

L'objet référencé par `ma_v` est modifié !

## Objets

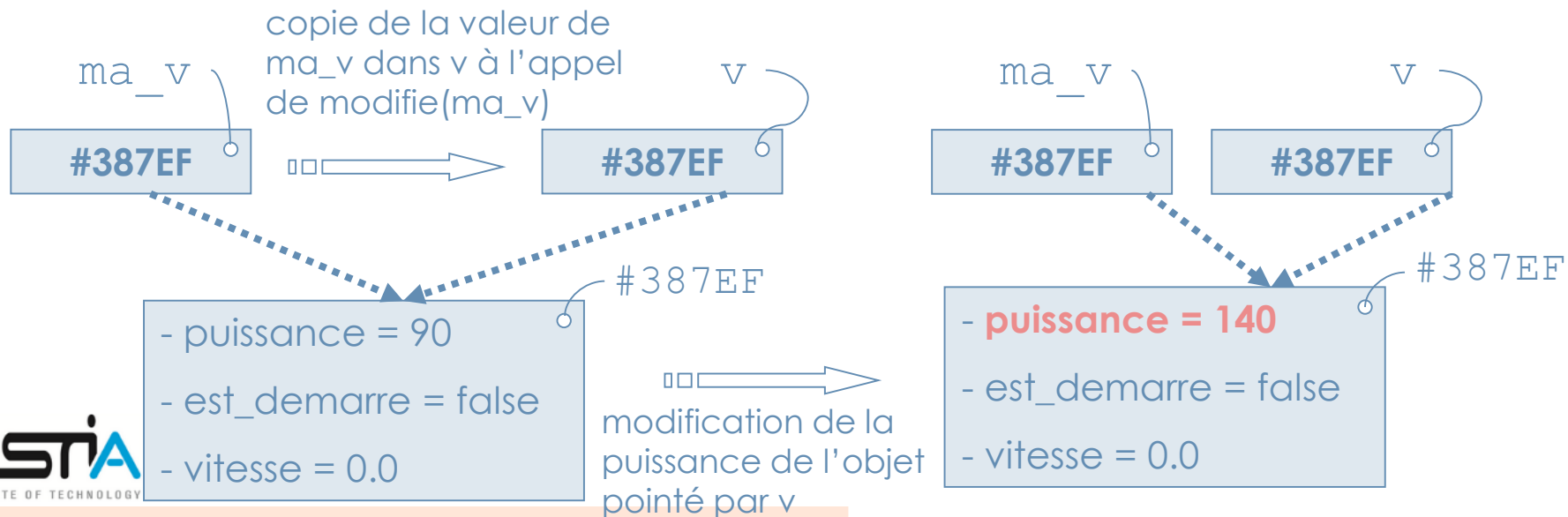
```
public class Garagiste {
[...]
```

/\* Définition d'une méthode \*/

```
public void modifie(Voiture v) {
    v.puissance = 140;
}
```

```
[...]
Garagiste g = new Garagiste();
Voiture ma_v = new Voiture();

/* Appel d'une méthode */
g.modifie (ma_v);
```



# Objet : la référence this

- Référence l'objet qui appelle la méthode
- Cas concret (et courant) : utilisation de **this** dans le cas où un paramètre formel de méthode a le même nom qu'une variable d'instance

```
public class Voiture {  
    private int puissance;  
    [...]  
    public void setPuissance(int puissance) {  
        this.puissance = puissance;  
    }  
}
```

# Objet : encapsulation

Objet =

état

comportement

identité

## Abstraction de données

Les données sont cachées.  
L'objet n'offre à l'extérieur qu'un ensemble de traitements (ou services), invocables par messages.

Caché ?



## Abstraction procédurale

Le détail de fonctionnement des traitements associés aux messages est caché.  
Un objet extérieur ne peut savoir si des objets temporaires sont créés, si des messages sont envoyés, ...

# Objet : encapsulation

---

La liste des messages auxquels est capable de répondre un objet constitue son **interface** : c'est la partie **publique** de l'objet.

Tout ce qui concerne son implémentation doit rester caché : c'est la partie **privée** de l'objet.



# Objet : encapsulation

---

A quoi bon encapsuler ?

**Logique** : L'utilisateur d'un « Moteur » n'a pas besoin (et souvent ne veut pas) savoir comment il fonctionne

**Modularité** : Si l'on veut remplacer une pièce du moteur, on n'a pas besoin de savoir comment elle fonctionne à l'intérieur mais seulement « comment elle se connecte avec l'extérieur »

**Modifications ultérieures** : Si la pièce que l'on remplace ne fonctionne pas de la même manière que l'ancienne, ce n'est pas important si elle se connecte de la même manière

# Objet : encapsulation

- Exemple 1

GroupeDeTD
etudiants : tableau [Etudiant] ...
appartient(Etudiant) : boolean ...

=

GroupeDeTD
etudiants : liste [Etudiant] ...
appartient(Etudiant) : boolean ...

- Exemple 2

Point
x : Double y : Double
abscisse() : Double ordonnee() : Double

=

Point
angle : Double distance : Double
abscisse() : Double ordonnee() : Double

# Objet : encapsulation

## 3 niveaux d'encapsulation

- Privé
- Protégé
- Public

+ : publique  
# : protégé  
- : privé

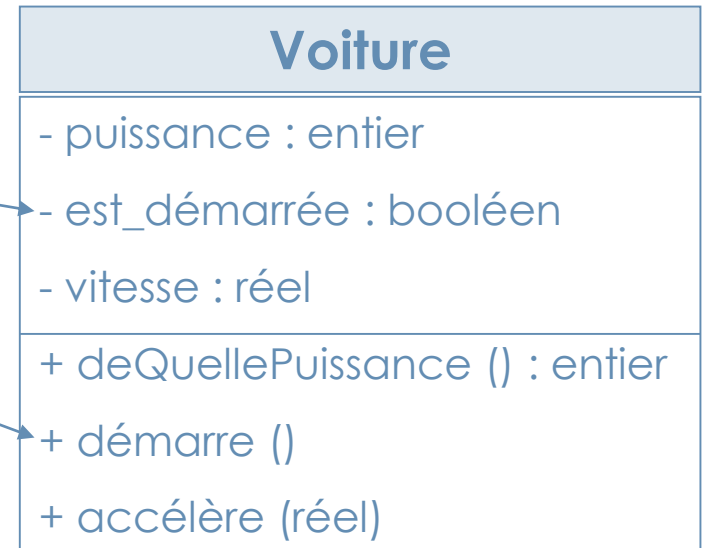
Point
- x : Double - y : Double
+ distanceOrigine() : Double + translate(dx:Double, dy:Double)

Attention  
Différences selon les langages



# Objet : encapsulation

- L'accès direct aux variables d'un objet est possible en JAVA, mais il n'est pas recommandé.
- Il est en effet contraire au principe d'**encapsulation** qui veut que les données d'un objet lui soit privées (c'est à dire accessibles uniquement par des méthodes prévues à cet effet)
- Plusieurs niveaux de visibilité peuvent être définis :
  - **private**
  - **public**
  - **protected (#)**



# Objet : encapsulation

Les variables d'instance déclarées comme privées (**private**) sont totalement protégées :

- elles ne sont plus directement accessibles depuis une autre classe ;
- pour accéder à leur valeur il faut passer par une méthode.

```
public class Voiture {  
    private double vitesse;  
    [...]  
    public void accelere(double vitesse) {  
        this.vitesse = this.vitesse + vitesse;  
    }  
}
```

→ autorisé car  
on est dans la  
classe Voiture

Interdit (ne compile pas)  
car on est à l'extérieur  
de la classe Voiture

```
public class UneAutreClasse {  
    [...]  
    Voiture ma_voiture;  
    ma_voiture.vitesse = 50;  
    [...]
```

# Objet : encapsulation

Pour chaque attribut « privé » on définit deux méthodes :  
un **accesseur** et un **modificateur**

```
public class Voiture {  
    private double vitesse;  
    [...]  
    public double getVitesse() {  
        return vitesse;  
    }  
    public void setVitesse(double vitesse) {  
        this.vitesse = vitesse;  
    }  
}
```

→ accesseur

→ modifieur

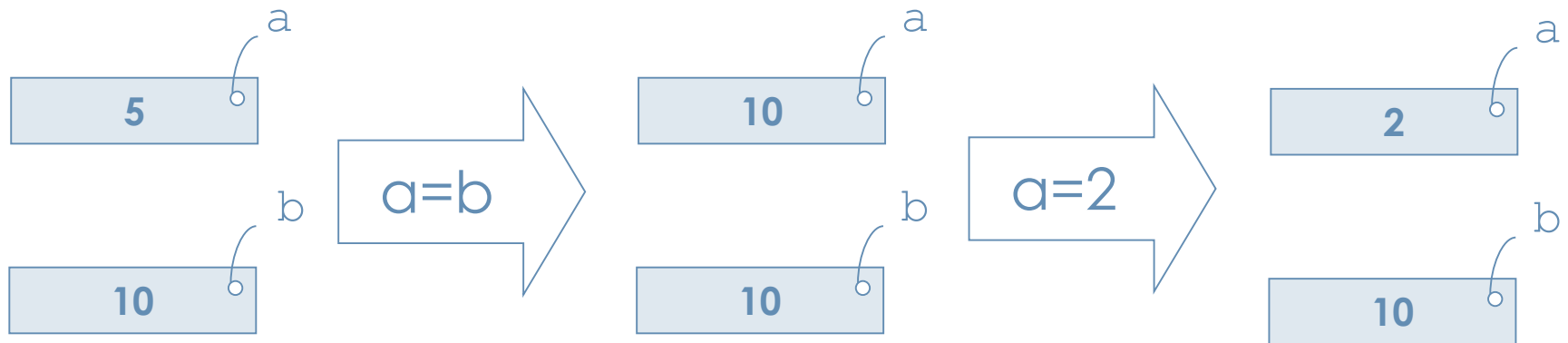
Interdit (ne compile pas)  
car on est à l'extérieur  
de la classe Voiture

Autorisé car la méthode  
setVitesse publique

```
public class UneAutreClasse {  
    [...]  
    Voiture ma_voiture;  
    ma_voiture = new Voiture();  
    ma_voiture.vitesse = 50;  
    ma_voiture.setVitesse(50);  
    [...]
```

# Types élémentaires : affectation, comparaison

- $a=b$  signifie : a prend la valeur de b
- a et b sont indépendants et distincts : modifier a ne modifie pas b

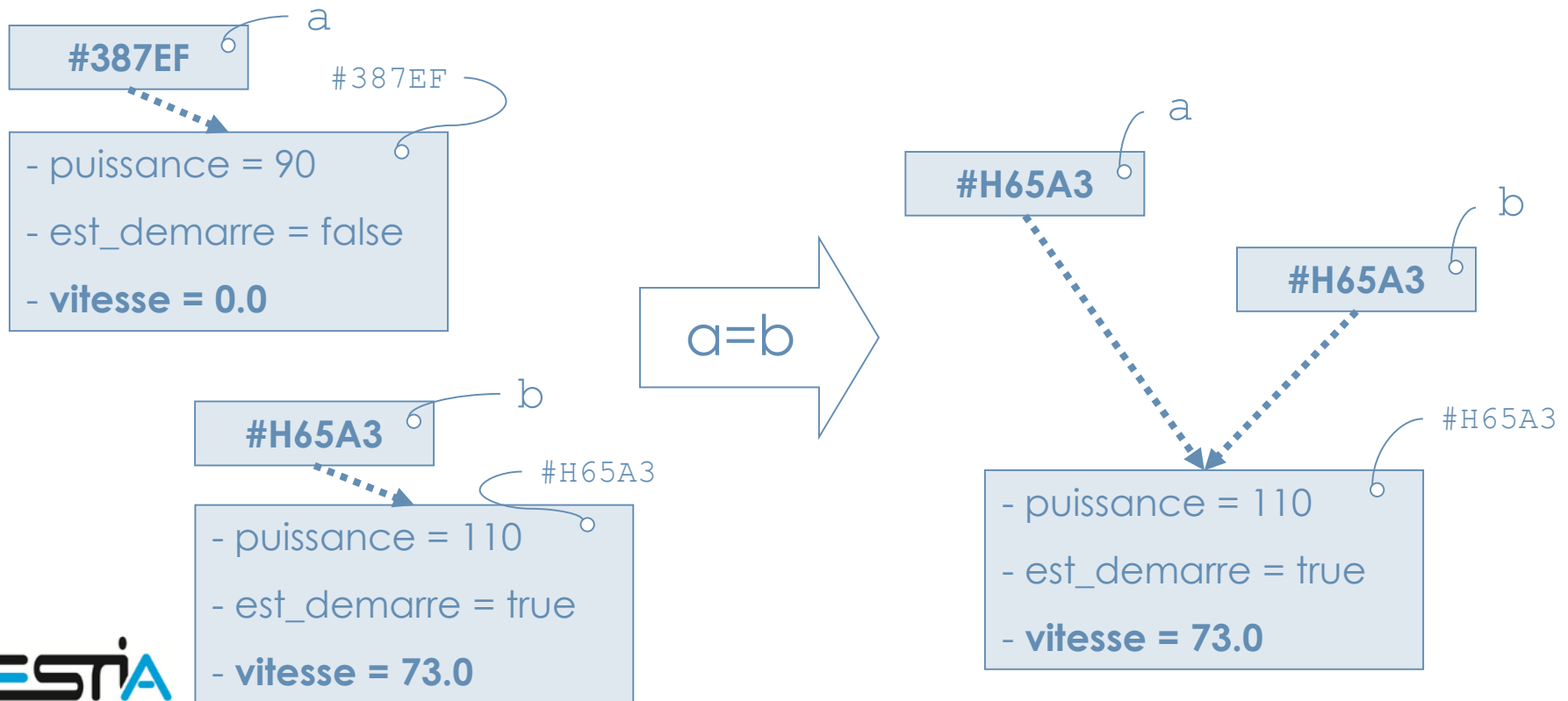


- $a==b$  signifie : est-ce que a et b ont la même valeur ? Retourne true si c'est le cas.

# Objets : affectation, comparaison

$a=b$  signifie :  $a$  et  $b$  référencent le même objet

modifier «  $a$  » (l'objet référencé par  $a$ ) modifie «  $b$  » (l'objet référencé par  $b$ )





# Relations entre classes

## Association

Connexion sémantique  
bidirectionnelle entre classes

Les deux classes jouent le même  
rôle l'une par rapport à l'autre



# Relations entre classes

## Agrégation

Couplage fort. Une classe joue un rôle plus fort (relations maître-esclaves).



# Relations entre classes

## Composition

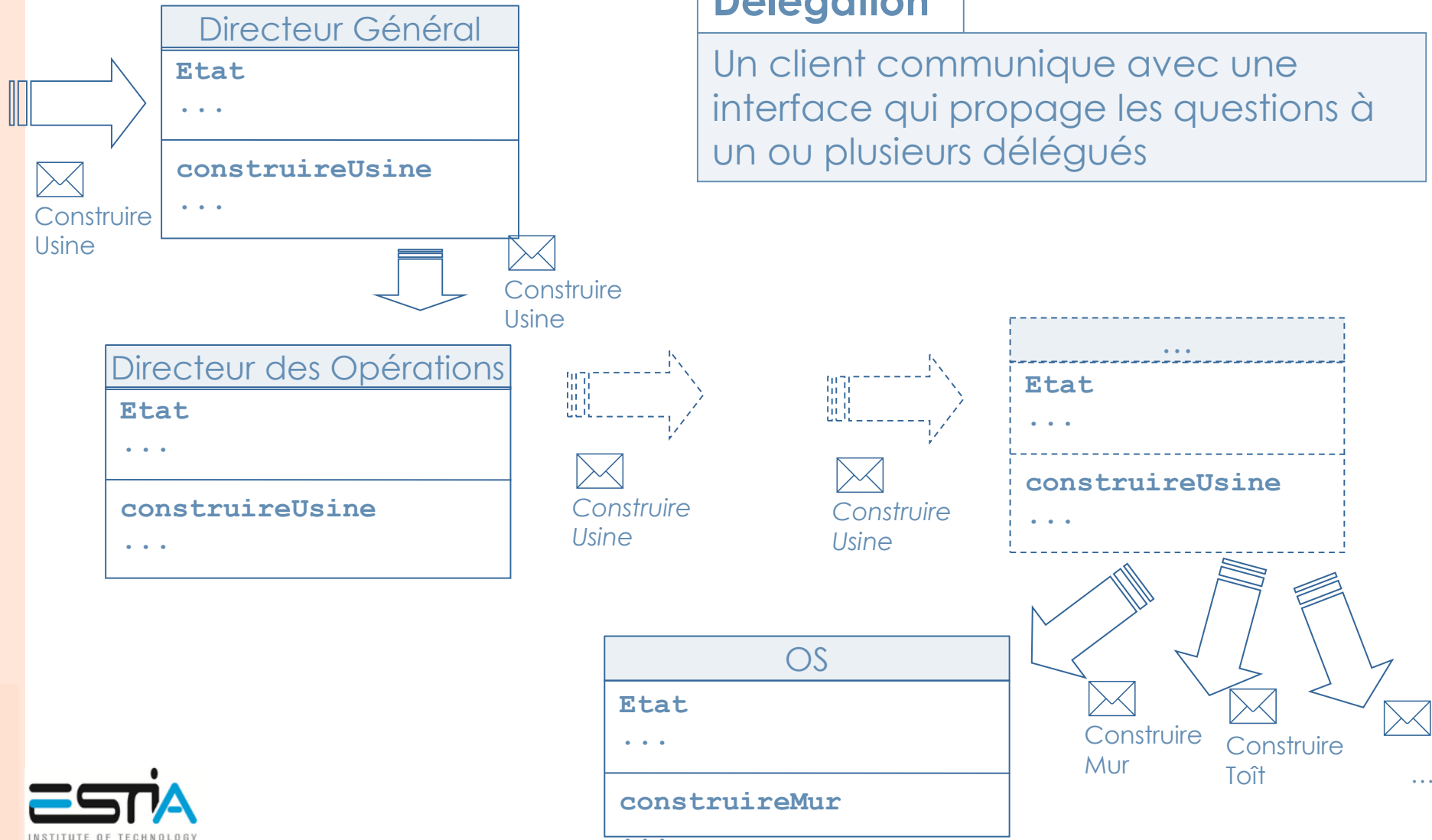
Sorte d'agrégation dans laquelle les objets sont liés par leur cycle de vie : la disparition de l'un entraîne celle de l'autre.



# Relations entre classes

## Délégation

Un client communique avec une interface qui propage les questions à un ou plusieurs délégués



...ou **multiplicité** : précise le nombre d'instances qui participent à la relation.

## Valeurs les plus courantes :

1 : un et un seul

0..1 : zéro ou un

M..N : de M à N (entiers naturels)

\* : de zéro à plusieurs

1..\* : plusieurs

Exemples :



*Java : classes et objets*

*Questions*



# Plan du cours



## ***JAVA***

1. Syntaxe de base
2. Classes et objets
3. **E/S**
4. Héritage/Polymorphisme
5. Normes de développement
6. Exceptions



- Les Entrées/Sorties :
  - Ecrire dans un fichier
  - Ecrire à l'écran
  - Lire dans un fichier
  - Lire ce qui est saisi au clavier

# E/S, écriture dans un fichier

- Package IO
- Classe File (pour fichiers et répertoires) et ses sous-classes (...)
- Les méthodes :

```
// constructeur
File destinationFile = new File(<chemin>,
                                <nom fichier>) //

// existence
destinationFile.exists()

// est ce un fichier ?
destinationFile.isFile()

// accès en écriture
destinationFile.canWrite()
```

# E/S, lecture dans un fichier

- Package IO
- Classe File (pour fichiers et répertoires) et ses sous-classes (...)
- Les méthodes :

```
// constructeur
File destinationFile = new File(<chemin>,
                                <nom fichier>) //

// accès en lecture
destinationFile.canRead()
```

# E/S, sortie sur écran

---

- Package : lang
- Classe : System
- Variable de classe : **PrintStream out**
- Méthode de classe **println()**

```
System.out.println("Le message") ;
```

# Package Java.io : Flux E/S

	LECTURE	ECRITURE
TEXTE	Reader	Writer
BINAIRE	InputStream	OutputStream

	LECTURE	ECRITURE
TEXTE	Reader +- BufferedReader +- StringReader +- CharArrayReader +- ...	Writer +- BufferedWriter +- StringWriter +- CharArrayWriter +- FileWriter +- PrintWriter +- ...
BINAIRE	InputStream +- FileInputStream +- ByteArrayInputStream +- ObjectInputStream +- PipedInputStream +- ...	OutputStream +- FileOutputStream +- ByteArrayOutputStream +- ObjectOutputStream +- PipedOutputStream +- ...

# E/S, écriture dans un fichier

- Java.io.DataOutputStream, pour format binaire.
  - Les méthodes pour écrire :

```
// constructeur
DataOutputStream destinationFile = new
                                DataOutputStream(...)

destinationFile.write(<binary>)
destinationFile.write<type>(<binary>) ;
```

- Java.io.PrintStream pour format texte
  - Les méthodes pour écrire :

```
// constructeur
PrintStream destinationFile = new PrintStream(...)

destinationFile.print(<String>)
destinationFile.println(<String>) ;
```

# E/S, lecture dans un fichier

Java.io.DataInputStream, pour format binaire et texte

- Les méthodes pour lire :

```
// constructeur
DataInputStream destinationFile = new
                                DataInputStream(...)

// lit dans un tableau d'octets
tmp = destinationFile.read();

// lit des caractères jusqu'à newline ou retour chariot.
tmp = destinationFile.readLine();
```

# JAVA : E/S, lecture au clavier

---

- Package : lang
- Classe : System
- Variable de classe : InputStream **in**
- Nécessite un reclassement, méthode :

```
// reclassement
BufferedReader in = new BufferedReader(new
    InputStreamReader(System.in));

// lecture au clavier tmp = destinationFile.read();
Response = in.readLine();
```



***Java : E/S***

***Questions***



# Plan de cours



## ***JAVA***

1. Syntaxe de base
2. Classes et objets
3. E/S
4. **Héritage/Polymorphisme**
5. Normes de développement
6. Exceptions
7. Aller plus loin ...

Présentation de la notion d'héritage et mise en oeuvre dans le langage JAVA.

*Référence : chapitre 3 du livre "JAVA in a Nutshell" de David Flanagan, Ed. O'Reilly 1997.*

- Classe
- Extension d'une classe : la notion d'héritage
- Redéfinition des méthodes (overriding)
- Surcharge des méthodes (overloading)
- Constructeurs et héritage
- Encapsulation, visibilité des données et des méthodes

# Héritage – Rappel : classe

---

Une classe :

- définit les possibilités des objets d'un type donné ;
- décrit l'ensemble des données (**variables d'instance**) et des opérations sur ces données (**méthodes**) ;
- sert de "modèle" pour la création d'objets (**instances de la classe**)

# Héritage – classe

```
public class Point {           // définition de la classe
    // variables d'instances
    double x;  // abscisse du point
    double y;  // ordonnée du point

    // translate le point de dx en abscisse et dy en ordonnée
    public void translate(double dx, double dy) {
        x = x + dx;
        y = y + dy;
    }

    // calcule et retourne la distance du point à l'origine
    public double distance() {
        double dist;
        dist = Math.sqrt(x * x + y * y);
        return dist;
    }
}
```

# Héritage – Extension de classe

---

- Problème :  
une application a besoin de services dont une partie seulement est proposée par une classe déjà définie
- Contrainte :  
ne pas réécrire le code
- Solution en POO :  
définir une nouvelle classe à partir de la classe déjà existante, c'est l'héritage

# Héritage – Extension de classe

Exemple : On veut manipuler des points (comme le permet la classe Point) mais **en plus les dessiner** sur l'écran.

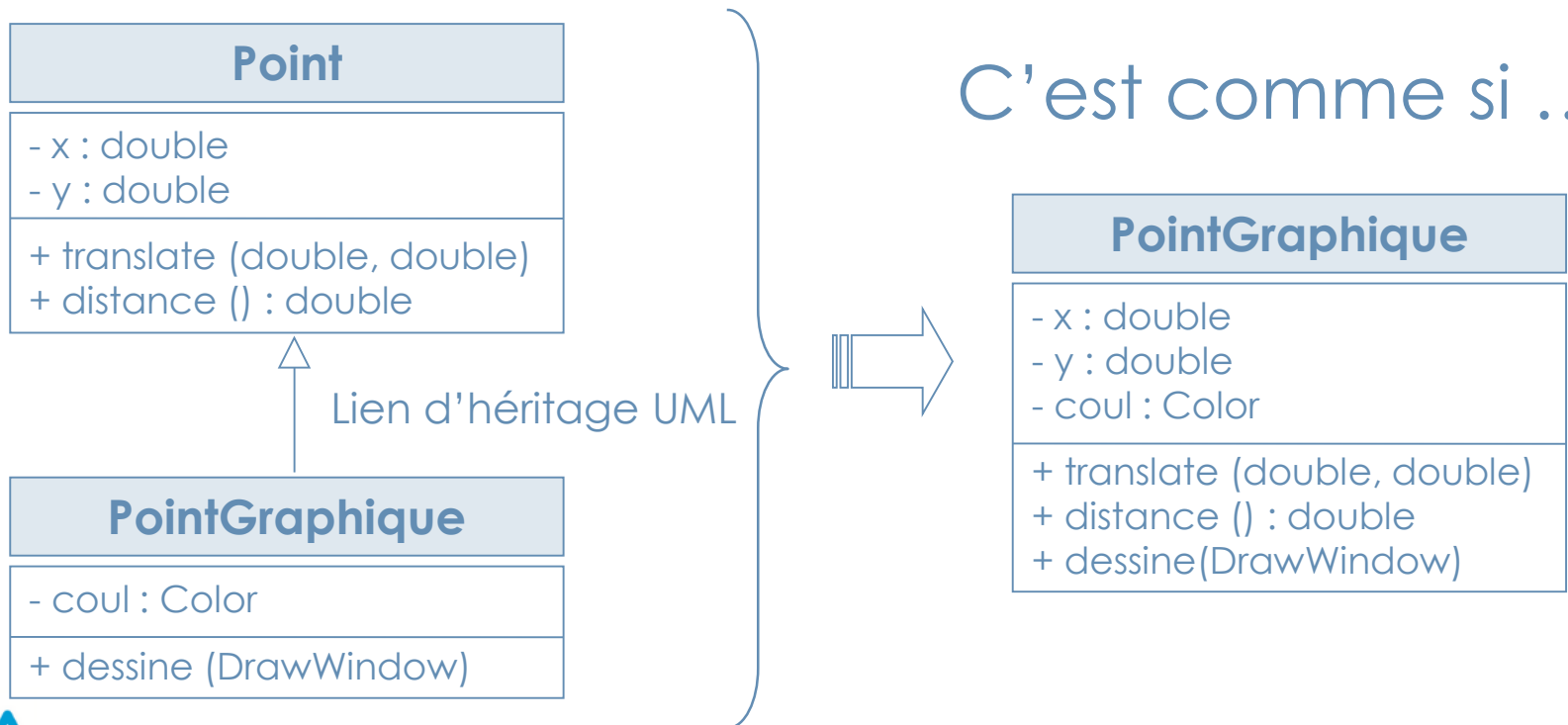
## PointGraphique

- x : double
- y : double
- coul : Color

- + translate (double, double)
- + distance () : double
- + dessine (DrawWindow)

# Héritage – Extension de classe

- ✓ la classe PointGraphique **étend** la classe Point
- ✓ la classe PointGraphique est une **sous-classe** de la classe Point
- ✓ la classe Point est la **super-classe** de la classe PointGraphique
- ✓ la classe PointGraphique **hérite** de la classe Point





# Héritage – Extension de classe

On veut manipuler des points (comme le permet la classe Point) mais en plus les dessiner sur l'écran.

```
//définition de la classe  
public class PointGraphique extends Point {  
    Color coul; // variable d'instance  
  
    // méthode : affiche le point  
    public void dessine(DrawWindow dw) {  
        dw.drawPoint(x,y,coul);  
    }  
}
```

Exemple

# Héritage – Extension de classe

---

```
PointGraphique p = new PointGraphique();  
p.coul = new Color(1.0,0.0,0.0);  
  
/* utilisation d'une méthode héritée */  
double dist = p.distance();  
System.out.println("distance à l'origine : " + dist);  
p.dessine(maFenetre);
```

# Héritage – Extension de classe

- la classe **PointGraphique** hérite des variables et méthodes définies dans la classe **Point** (en fait pas de celles qui sont définies comme privées)

PointGraphique hérite de Point

C'est comme si ...

Cas 1

f est « **public** » dans Point

f était déclarée  
«**public**» dans  
PointGraphique

Cas 2

f est « **protected** » dans Point

f était déclarée  
«**protected**» dans  
PointGraphique

Cas 3

f est « **private** » dans Point

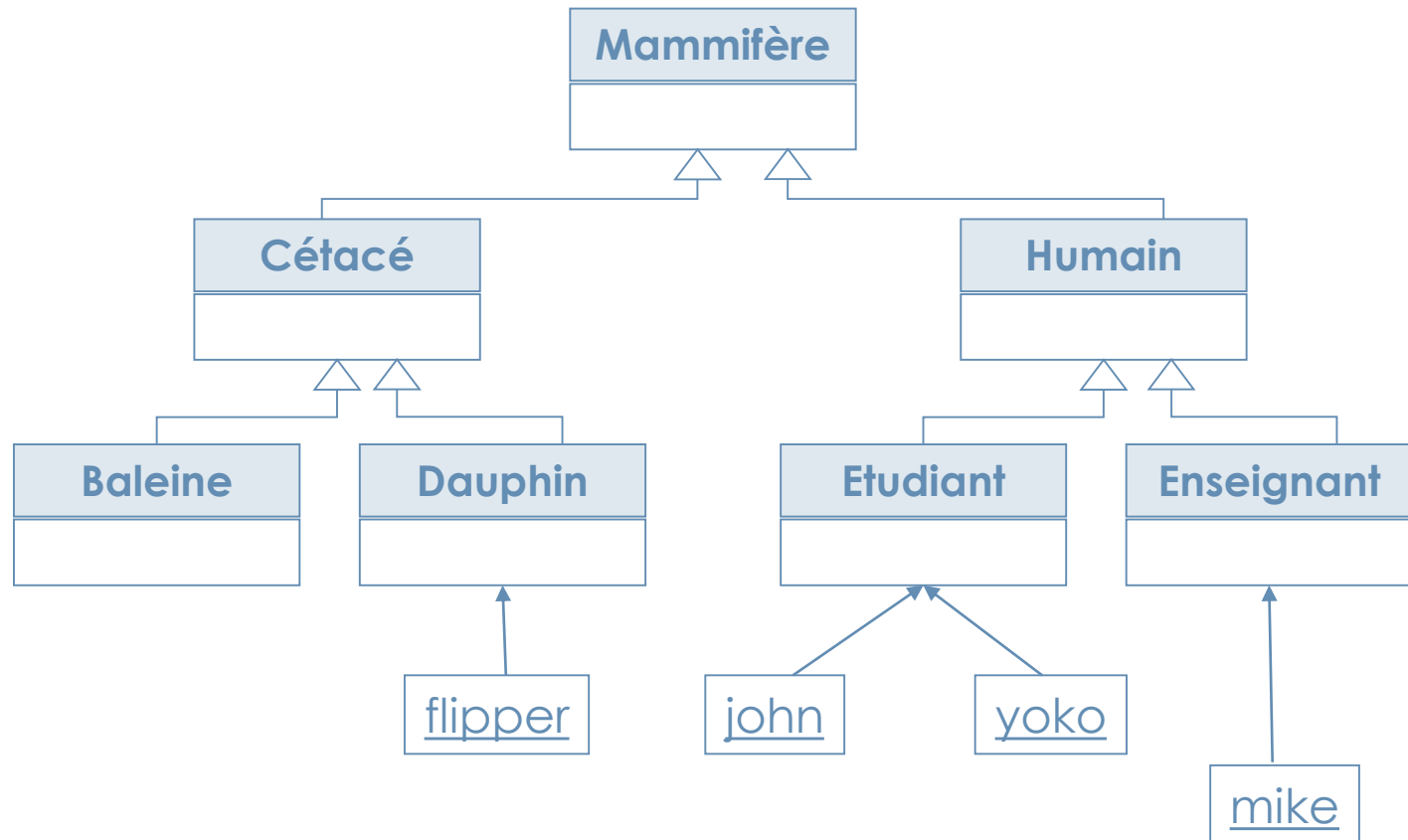
f n'était pas déclarée  
dans PointGraphique

# Héritage – Extension de classe

---

- un objet de la classe **PointGraphique** est un objet de la classe **Point** avec des caractéristiques supplémentaires
- la relation d'héritage peut être vue comme une relation de **spécialisation**
- une sous-classe peut ajouter des variables et/ou des méthodes à celles qu'elle hérite de sa super-classe (ou classe mère, parent, ...)
- pas de limitations dans le nombre de niveaux dans la hiérarchie d'héritage
- méthodes et variables sont héritées au travers de tous les niveaux

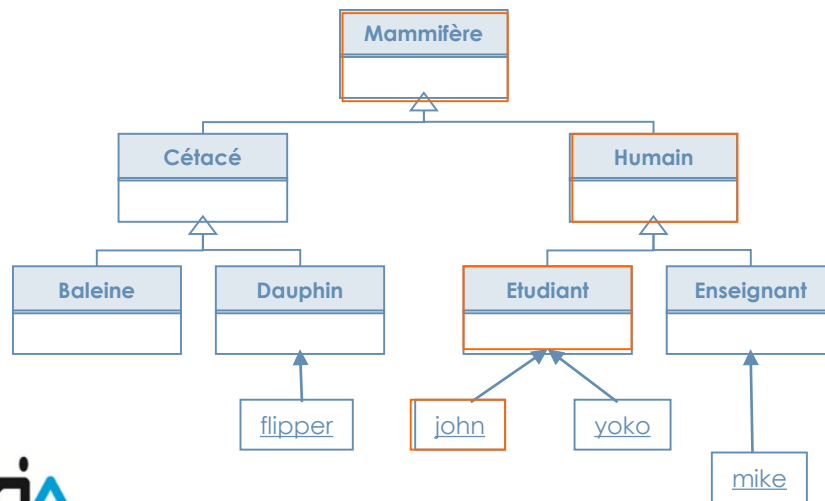
Exemple :



# Héritage - Surclassement

Une classe **B** qui hérite de la classe **A** peut être vue comme un sous-type (sous ensemble) du type défini par la classe **A**.

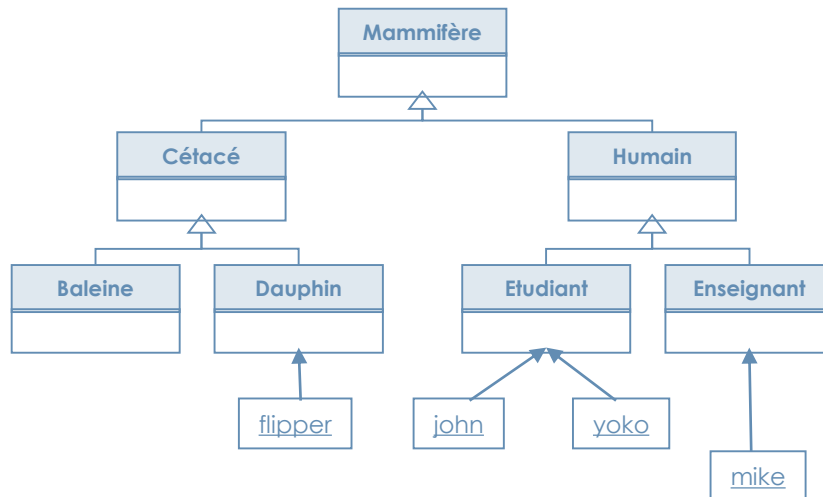
Tout objet instance de la classe B peut être vu comme une instance de la classe A



John est une instance d'Etudiant, mais c'est aussi un Humain et un Mammifère

# Héritage - Surclassement

Lorsqu'un objet est "**sur-classé**" (*upcasted*) il est vu comme un objet du type de la référence utilisée pour le désigner et ses fonctionnalités sont restreintes à celles proposées par la classe du type de la référence .



Si on surclasse John, en tant que Mammifère on ne pourra pas lui demander d'écrire (opération spécifique aux humains)

# Héritage simple

---

En Java **héritage simple** uniquement.

Une classe ne peut hériter que d'une seule classe.

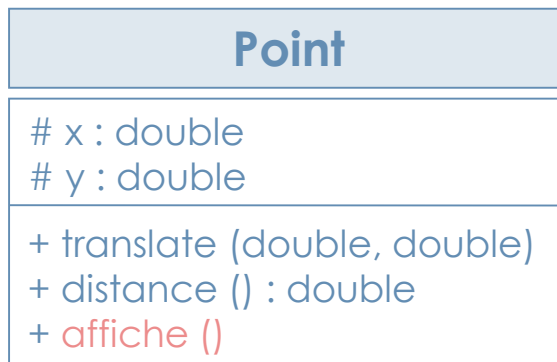
- Classe **Object** : classe racine hiérarchie d'héritage
- Toute classe possède une super-classe (cf **Object**)
- Toute classe hérite directement ou indirectement de la classe **Object**
- Par défaut une classe qui ne définit pas de clause **extends** hérite de la classe **Object**



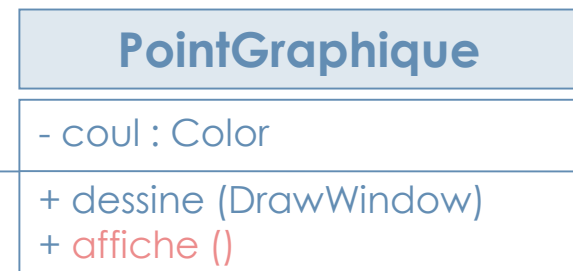
# Héritage – Redéfinition de méthode

## Overriding

Lorsqu'une classe définit une méthode dont la signature est la même que celle d'une méthode d'une classe dont elle hérite, on dit qu'elle la **redéfinit** (*overrides*)



Redéfinition de la méthode affiche



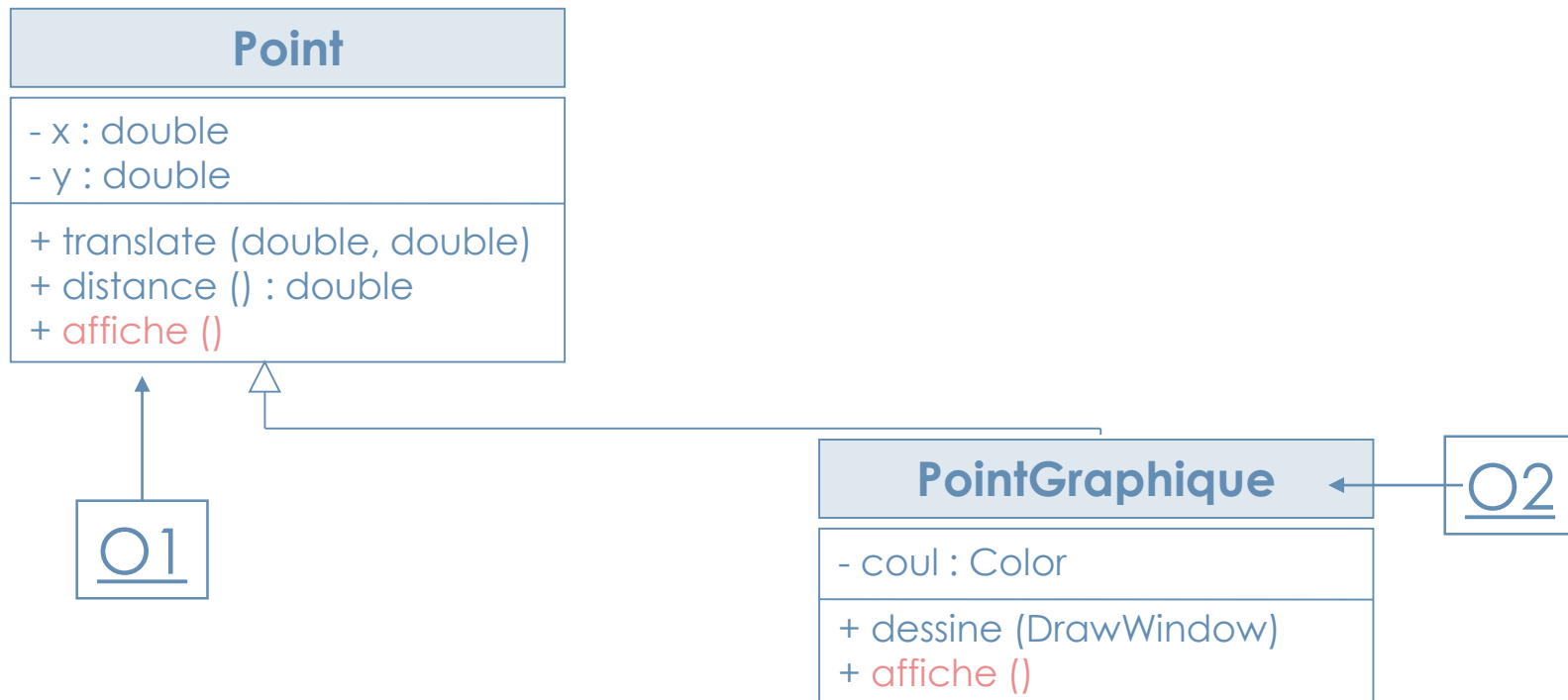
# Héritage – Redéfinition de méthode

```
public class Point {  
    [...]  
    public void affiche() {  
        System.out.println("Je suis un point");  
    }  
}
```

```
public class PointGraphique extends Point {  
    [...]  
    // redéfinition de la méthode affiche héritée de Point  
    public void affiche() {  
        System.out.println("Je suis un point graphique");  
    }  
}
```

# Héritage – Redéfinition de méthode

Lorsqu'un traitement défini dans une classe A et redéfini dans une classe B est invoquée pour une instance de B, c'est la nouvelle définition du traitement (celui de B) et non pas celui de la super classe qui est invoquée



# Héritage – Redéfinition de méthode

---

Lorsqu'une méthode définie dans une classe A et redéfinie dans une classe B est invoquée pour une instance de B, c'est la **nouvelle** définition de la méthode (celle de B) et non pas celle de la super classe qui est invoquée

```
PointGraphique b = new PointGraphique();  
b.affiche();
```

```
résultat :  
> Je suis un point graphique
```

# Héritage – Redéfinition de méthode

---

- Lorsqu'une classe redéfinit une méthode deux possibilités lui sont offertes :
  - **remplacer** complètement la méthode héritée par la nouvelle définition
  - **étendre** la méthode utilisée en lui **ajoutant** du code
- L'invocation d'une méthode qui est redéfinie par la classe se fait au travers du mot réservé **super**

# Héritage – Redéfinition + Extension

```
public class Etudiant {
    private String nom, prenom;
    private int age;

    ...

    public void affiche() {
        System.out.println("Nom : " + nom + " Prénom : " +
            prenom + "Age : " + age);}

    ...
}

public class EtudiantSportif extends Etudiant {
    private String sportPratique;

    ...

    public void affiche() {
        /* invocation de la méthode affiche définie dans Etudiant */
        super.affiche();
        System.out.println("Sport pratiqué : " + sportPratiqué);
    }
}
```

# Héritage – Redéfinition + Extension

```
Etudiant e = new Etudiant();  
[...]  
e.affiche();
```

Résultat

```
> Nom : DUPONT-DURAND  
Prénom : Jean-Luc  
Age : 26
```

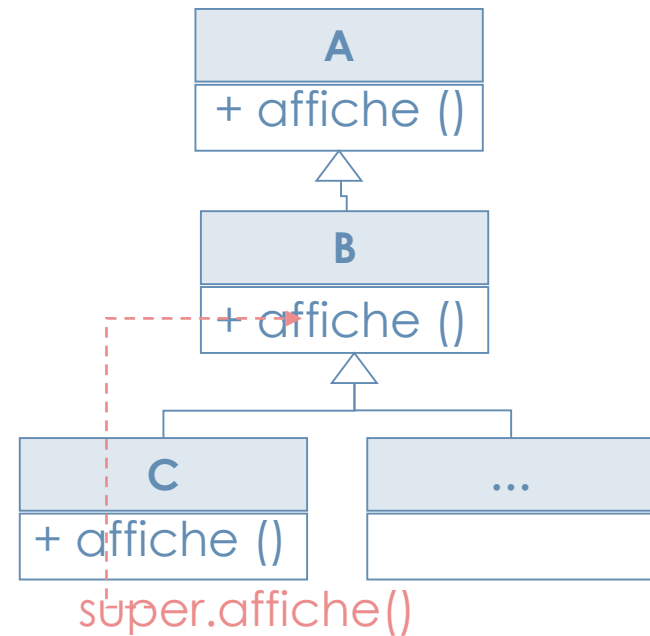
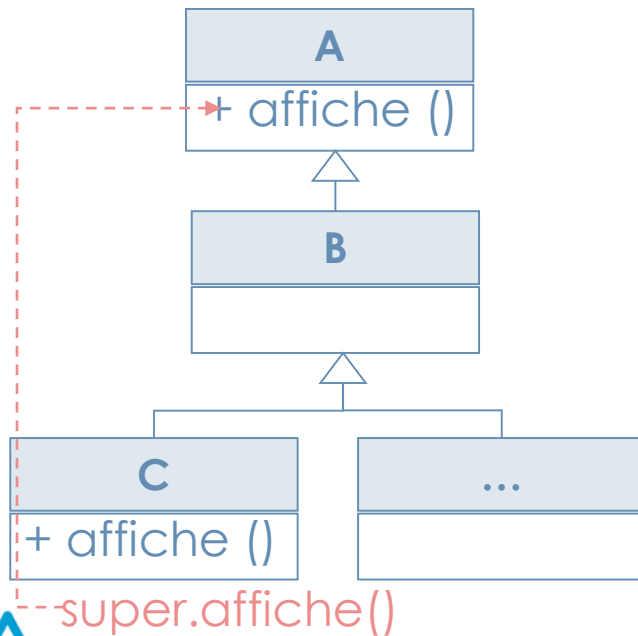
```
EtudiantSportif e = new EtudiantSportif();  
[...]  
e.affiche();
```

Résultat :

```
> Nom : DUPONT-DURAND  
Prénom : Jean-Luc  
Age : 26  
Sport pratiqué : ski
```

# Héritage – Redéfinition + Extension

Le message `super.nomDeMéthode(...)` dans la définition d'une classe C, fait référence à la définition de la méthode `nomDeMéthode(...)` située au niveau le plus proche de la classe C dans la hiérarchie des super classes de C.





# Héritage – Redéfinition + Extension

```
public class ClasseA {  
    public void affiche() {  
        System.out.println("affiche ClasseA");  
    }  
}
```

## Cas 1: affiche() n'est pas redéfinie

```
/* ClasseB étend ClasseA sans redéfinir la méthode affiche()  
*/  
  
public class ClasseB extends ClasseA { ... }
```

## Cas 2 : affiche() est redéfinie

```
/* ClasseB étend ClasseA et redéfinit la méthode affiche() */  
  
public class ClasseB extends ClasseA {  
    public void affiche() {  
        System.out.println("affiche ClasseB");  
    }  
}
```

# Héritage – Redéfinition + Extension

Suite ...

```
// ClasseC étend ClasseB et redéfinit la méthode affiche()  
// héritée de ClasseA  
  
public class ClasseC extends ClasseB {  
  
    public void affiche() {  
        super.affiche();  
        System.out.println("affiche ClasseC"); } }  

```

```
ClasseC c = new ClasseC();  
c.affiche();
```

Cas 1

Résultat

```
> affiche ClasseA  
    affiche ClasseC
```

Cas 2

Résultat

```
> affiche ClasseB  
    affiche ClasseC
```

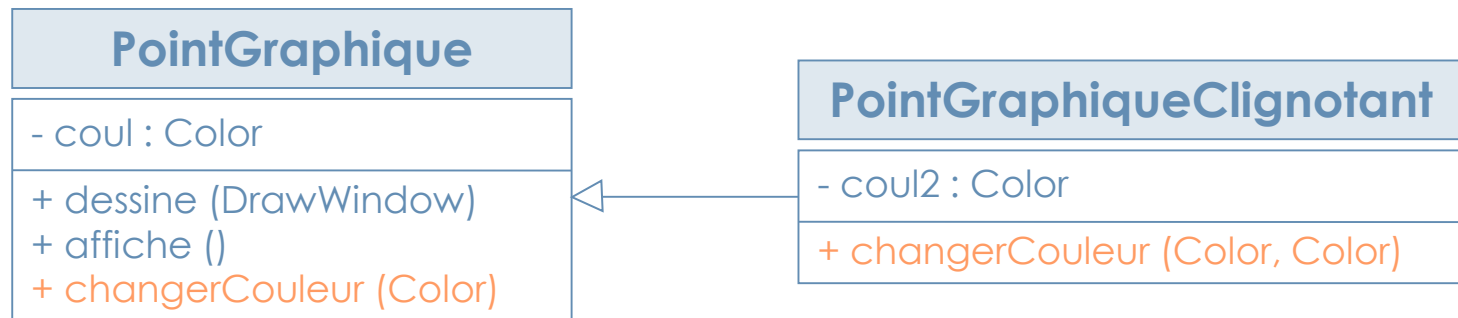
# Héritage – Surcharge de méthode

Si dans une classe une méthode possède

- Le même nom
- Un nombre d'arguments **et/ou** des types d'arguments différents

qu'une méthode existante définie dans la classe ou héritée, alors il y a

## surcharge de la méthode (Overloading)



# Héritage – Surcharge de méthode

```
public class A {  
    public void affiche() {  
        System.out.println("Je suis un objet de A");}  
    public void affiche(String s) {  
        System.out.println(s);}}}
```

```
public class B extends A {  
    public void affiche(int nombre) {  
        System.out.println("nombre : " + nombre);}}}
```

Surcharge dans une  
classe dérivée

Surcharge dans une  
même classe

# Héritage – Constructeurs et héritage

---

- On peut réutiliser le code des constructeurs de sa super classe dans la définition des constructeurs d'une nouvelle classe
- L'invocation d'un constructeur de la super classe se fait à l'aide de l'appel **super** (*paramètres du constructeur*)
- L'utilisation de super est analogue à celle de this


# Héritage – Constructeurs et héritage

```
public class Point {  
    double x;    // abscisse du point  
    double y;    // ordonnée du point  
  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

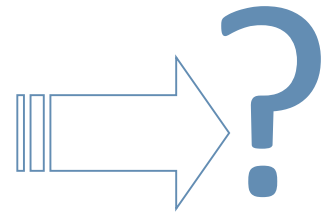
```
public class PointGraphique extends Point {  
    Color coul;  
  
    public PointGraphique(double x, double y, Color coul){  
        // appel au constructeur de la classe Point  
        super(x, y);  
        this.coul = coul ;}}}
```

# Héritage – Constructeurs et héritage

- L'appel au constructeur de la super classe **doit toujours être la première instruction** dans le corps du constructeur
- Si la première instruction d'un constructeur n'est pas un appel à un constructeur de la superclasse, alors Java insère implicitement l'appel à **super()** (constructeur par défaut de la classe mère)

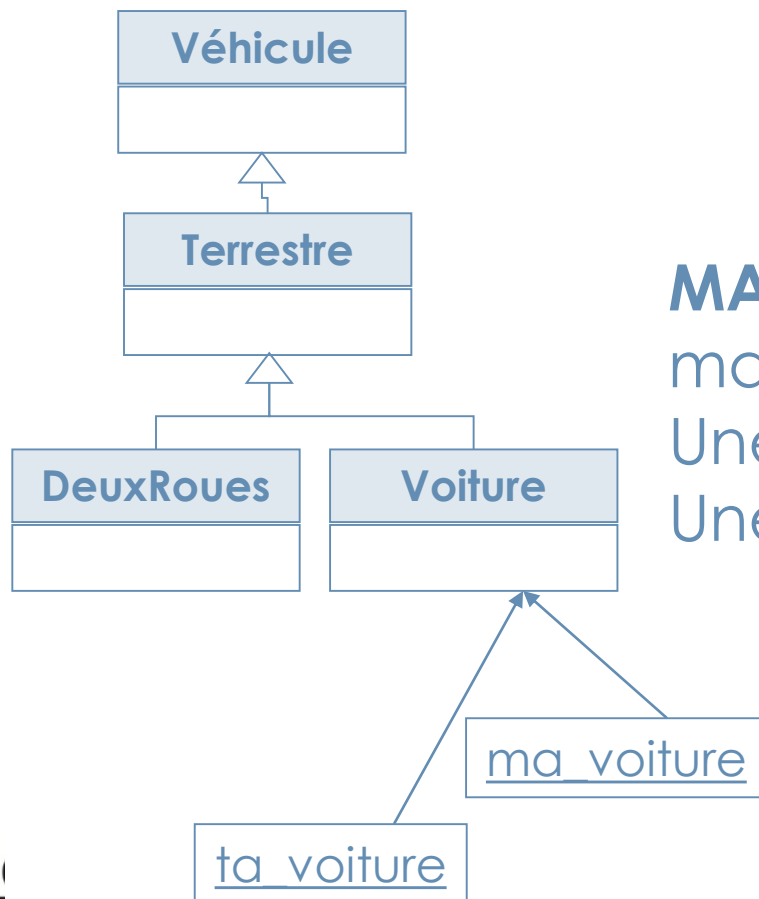


un constructeur d'une classe est toujours appelé lorsqu'une instance de l'une de ses sous classes est créée



# Héritage – Constructeurs et héritage

## Conséquence 1



ma\_voiture est une instance de la classe Voiture

**MAIS**

ma\_voiture est aussi :  
 Une instance de la classe Vehicule  
 Une instance de la classe Terrestre

Ce qui est finalement logique ...



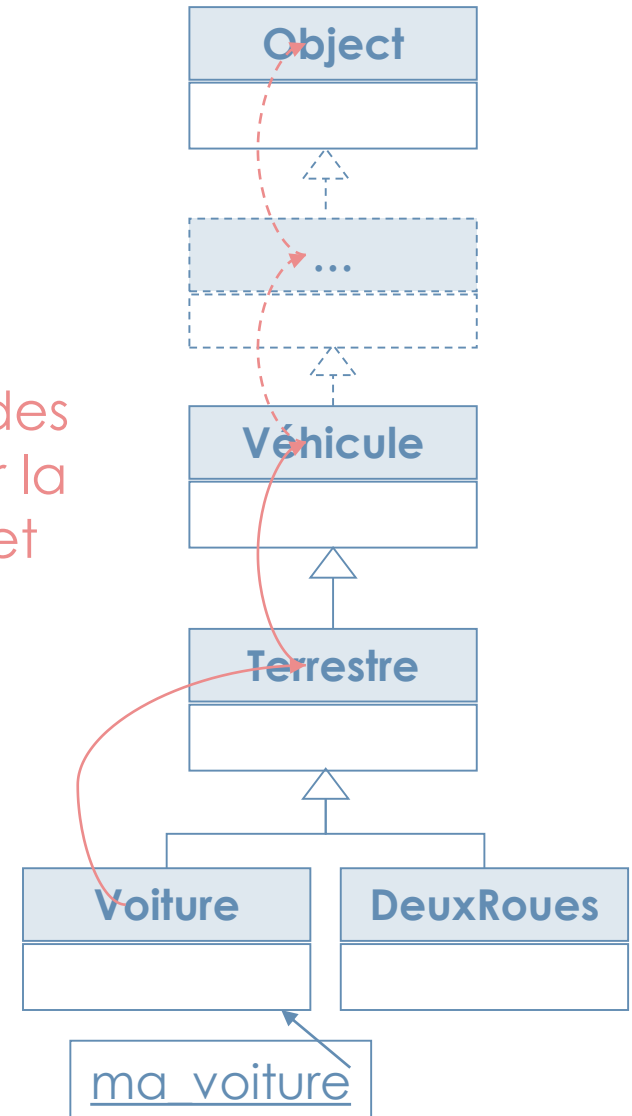
# Héritage – Constructeurs et héritage

## Conséquence 2

Les appels des constructeurs sont chaînés.

Appels successifs des constructeurs pour la création de l'objet ma voiture

Les corps sont exécutés en ordre inverse. Ici : Object, ..., Véhicule, Terrestre et enfin Voiture



# Héritage – Constructeurs et héritage

```
public class Point {
```

```
    double x, y;
```

```
    // constructeurs
```

```
    public Point(double x, double y) {
```

```
        // appel implicite à super();
```

```
        this.x = x;
```

```
        this.y = y;
```

```
    }
```

```
    ...
```

```
public class PointGraphique extends Point {
```

```
    Color coul;
```

```
    // constructeurs
```

```
    public PointGraphique(double x,
```

```
                           double y,
```

```
                           Color coul) {
```

```
        super(x,y) ;
```

```
        this.coul= coul;
```

```
    }
```

```
    ... }
```

# Héritage – Constructeurs et héritage

Lorsqu'une classe ne définit pas explicitement de constructeur, elle possède un **constructeur par défaut**, ce constructeur ne fait rien mis à part l'invocation du constructeur de la super-classe

- Corps du constructeur par défaut : `{ super(); }`
- **Si** la super-classe définit ses propres constructeurs, mais ne définit pas de constructeur sans argument
  - l'insertion de ce constructeur par défaut provoque **une erreur de compilation**.
  - **Et**, toutes les sous-classes de celle-ci doivent définir des constructeurs qui invoquent explicitement l'un des constructeurs de la super-classe

# Héritage – Constructeurs et héritage

```
class ClasseA {  
    double x;  
  
    // constructeur  
    public ClasseA(double x) {  
        this.x = x;}}
```

```
class ClasseB extends ClasseA {  
    int y = 0;  
  
    // pas de constructeur  
    // constructeur par défaut  
}
```

```
public class Test {  
    public static void main(String[] args){  
        ClasseB b = new ClasseB();}}
```

```
> javac Test.java
```

```
Test.java:10: No constructor matching  
ClasseA() found in class ClasseA.
```

```
class ClasseB extends ClasseA{  
    ^  
1 error
```

# Héritage – intérêts

---

- Evite les redondances de code
- On peut rajouter des classes très facilement en bénéficiant de ce qui a déjà été écrit
- On peut rajouter des méthodes qui sont utilisables immédiatement par toutes les classes filles

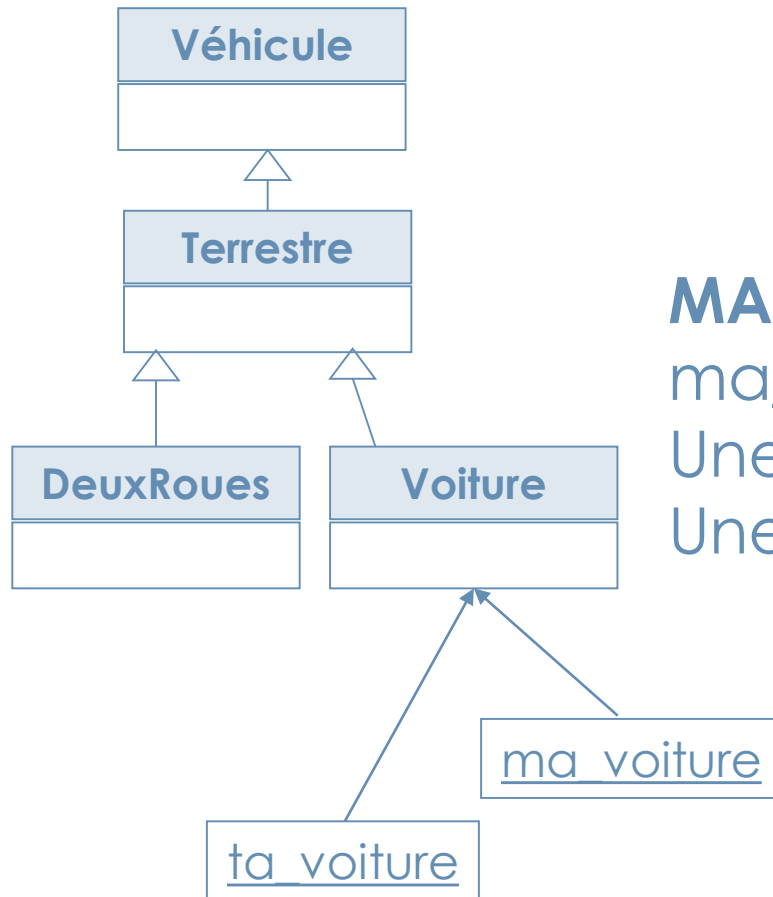
# Polymorphisme

**Définition :** Un langage orienté objet est dit polymorphique, s'il offre la possibilité de pouvoir percevoir un objet en tant qu'instance de classes variées, selon les besoins.

Une classe B qui hérite de la classe A peut être vue comme un sous ensemble (sous-type) de l'ensemble (ou type) défini par la classe A.

Tout objet instance de la classe B peut être aussi vu comme une instance de la classe A.

# Polymorphisme



ma\_voiture est une instance de la classe Voiture

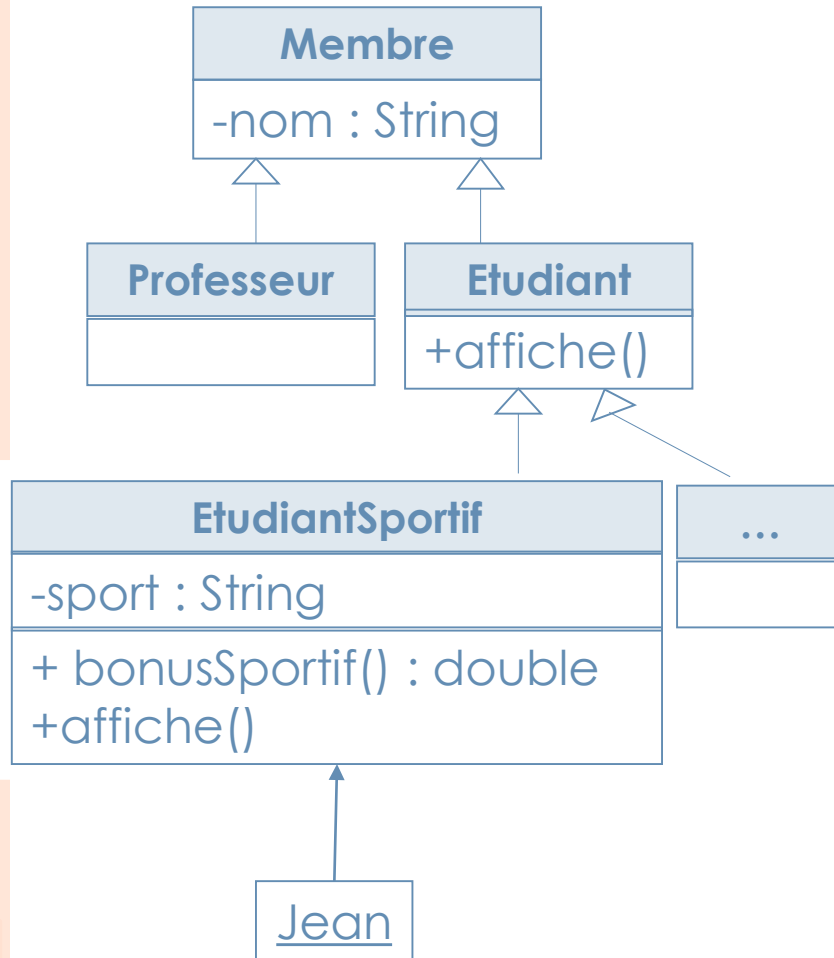
**MAIS**

ma\_voiture est aussi :

Une instance de la classe Vehicule

Une instance de la classe Terrestre

# Polymorphisme



- à une référence déclarée de la classe Etudiant, il est possible d'affecter une valeur qui est une référence vers un objet de la classe EtudiantSportif
- plus généralement à une référence d'un type donné, soit A, il est possible d'affecter une valeur qui correspond à une référence vers un objet dont le type effectif est n'importe quelle sous-classe directe ou indirecte de A.



# Polymorphisme

- Lorsqu'un objet est "surclassé", il est vu (par le compilateur) comme un objet du type de la référence utilisée pour le désigner.
- Ses fonctionnalités sont alors restreintes à celles proposées par la classe du type de la référence.

# Polymorphisme

```
Etudiant e = new EtudiantSportif("DUPONT",...);
```

```
e.affiche();
```

```
// oui : cette méthode fait partie de la classe Etudiant
```

```
System.out.println("nom " + e.nom);
```

```
// oui : nom est un attribut défini dans la classe Etudiant
```

```
System.out.println("sport pratique " + e.sport_pratique);
```

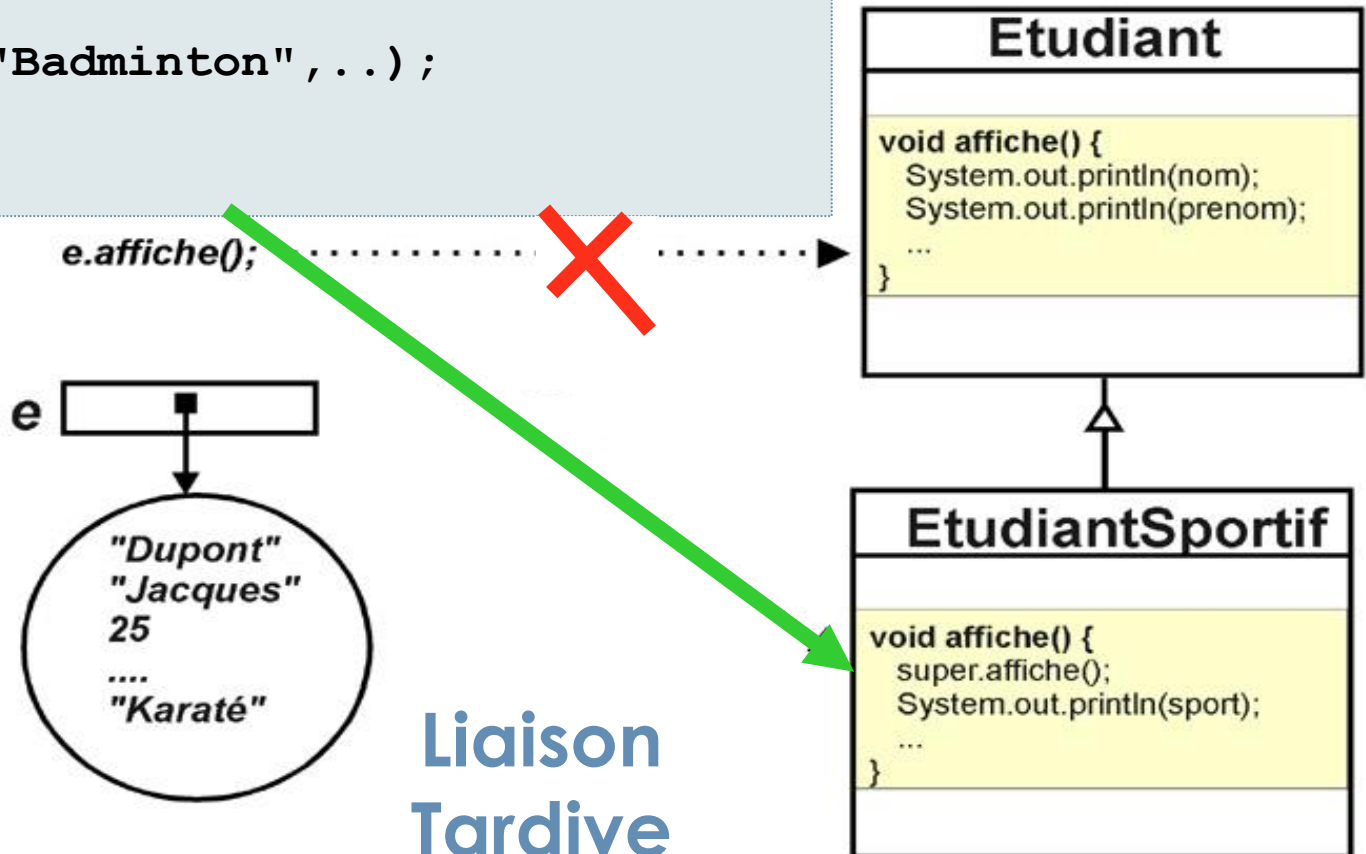
```
// non : sport est une variable de la classe EtudiantSportif
```

```
double b = e.bonusSportif();
```

```
// interdit : bonusSportif() est une méthode de la classe  
EtudiantSportif
```

# Polymorphisme

```
Etudiant e = new EtudiantSportif(
    "DUPONT",
    "JEAN", 25, ..., "Badminton", ...);
...
e.affiche();
```



**Liaison  
Tardive**  
Dynamic binding

# Polymorphisme

```
Etudiant e = new EtudiantSportif(  
    "DUPONT",  
  
    "JEAN", 25, ..., "Badminton", ...);  
...  
e.affiche();
```

```
Résultats :  
Etudiant Sportif  
nom: DUPONT  
prénom : Jacques  
age : 25  
...  
sport pratiqué : Badminton
```

# Polymorphisme *ex en Java*

## Statique et dynamique ... récapitulons

```
Etudiant e = new EtudiantSportif(  
    "DUPONT",  
    "JEAN", 25, ..., "Badminton", ...);  
  
...  
e.affiche();
```

1. Je déclare une variable **e** comme étant une référence vers un objet de la classe Etudiant ;
2. Je crée un objet de la classe EtudiantSportif ;
3. Pour le compilateur, **e** reste une référence vers un objet de la classe Etudiant, et il m'empêche d'accéder aux méthodes et attributs spécifiques à EtudiantSportif ;
4. A l'exécution, **e** est bel et bien une référence vers un objet de la classe EtudiantSportif ;

## Liaison tardive : bilan

Lorsqu'une méthode d'un objet est accédée au travers d'une référence "surclassée", c'est la méthode telle qu'elle est définie au niveau de la classe effective de l'objet qui est invoquée et exécutée.

## Comment est-ce possible ?

- Le compilateur ne dispose pas de l'information nécessaire pour associer le code d'une méthode à un message.
- En fait les messages sont résolus à l'exécution (*run-time*). A cet instant :
  - Le type exact de l'objet est connu.
  - Le mécanisme de **lien-dynamique** (*dynamic binding*, *late-binding* ou *run-time binding*) permet de retrouver le type d'un objet et d'appeler la méthode appropriée.

# Polymorphisme

```
class Etudiant {  
    String nom, prénom;  
    int age;  
    ...  
  
    public void affiche() {  
        System.out.println("Nom: " + nom + " Prénom : " + prénom);  
        System.out.println("Age: " + age);  
        ...    }  
}
```

```
class EtudiantSportif extends Etudiant {  
    String sport;  
    ...  
    public void affiche() {  
        super.affiche();  
        System.out.println("Sport pratiqué:" + sport);  
        ...    }  
}
```

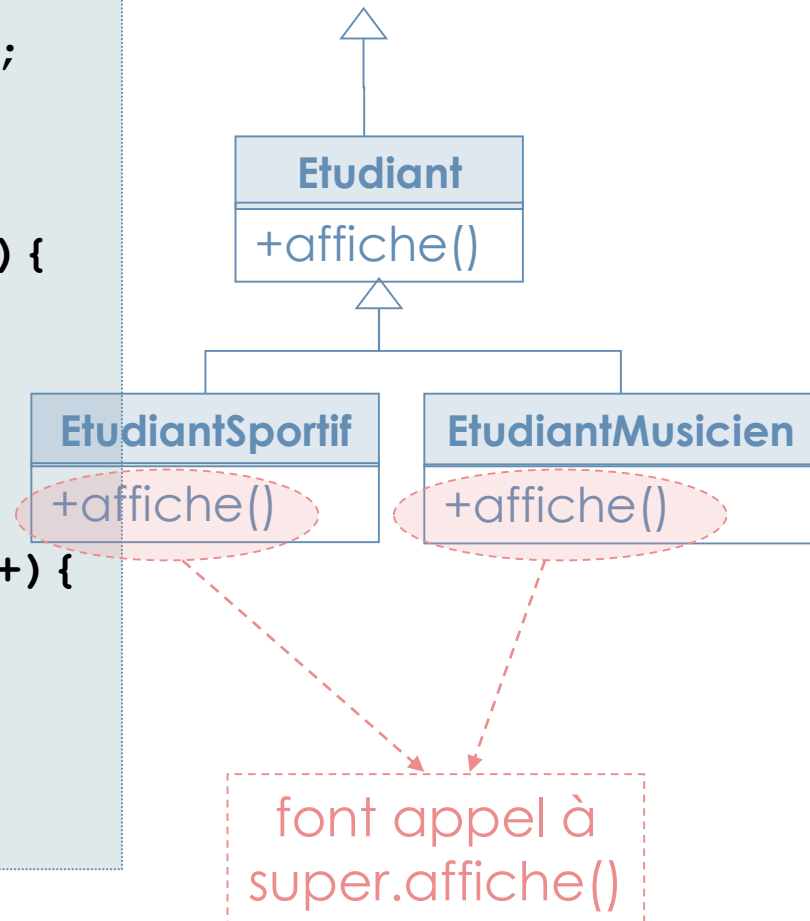


# Polymorphisme

```
class GroupeTD {
    // tableau d'Etudiants
    Etudiant[] liste = new Etudiant[];
    int nbEtudiants = 0;

    public void ajouter(Etudiant e) {
        if (nbEtudiants < liste.length){
            liste[nbEtudiants++] = e;
        }
    }

    public void afficherListe() {
        for (int i=0;i<nbEtudiants; i++){
            liste[i].affiche();
        }
    }
}
```



# Polymorphisme

---

*A quoi servent le polymorphisme et les liaisons dynamiques ?*

A écrire le code en s'adressant à la classe de base tout en sachant que toutes les classes dérivées fonctionneront correctement avec le même code.

*"You send a message to an object and let the object figure out the right thing to do"*  
[Bruce Eckel, Thinking in Java]

# Polymorphisme

---

Le polymorphisme associé à la liaison dynamique offre :

- une plus grande **simplicité** du code : plus besoin de distinguer les cas en fonction de la classe des objets,
- une plus grande **facilité d'évolution** du code : on peut définir de nouvelles fonctionnalités en héritant de nouveaux types de données à partir d'une classe de base commune sans avoir besoin de modifier le code qui manipule la classe de base.

# Classes abstraites

**CONSTAT** : On ne connaît pas toujours le comportement par défaut d'une opération commune à plusieurs sous-classes.

- Exemple : toit d'une voiture décapotable. On sait que toutes les décapotable peuvent ranger leur toit, mais le mécanisme est différent d'une décapotable à l'autre.

**SOLUTION** : déclarer une méthode abstraite (on peut voir cela comme un potentiel qui s'exprime dans les classes filles).

**Rôle d'une classe abstraite** : spécifier un prototype (un ensemble de méthodes) commun à toutes les classes qui en dérivent.

# Classes abstraites

---

- En java, c'est prévu : mot clef **abstract**
- Une classe est abstraite si elle contient au moins une méthode abstraite
- Pour créer une méthode abstraite, on déclare sa signature (nom et paramètres) sans spécifier le corps et en ajoutant le mot clef **abstract**

# Classes abstraites

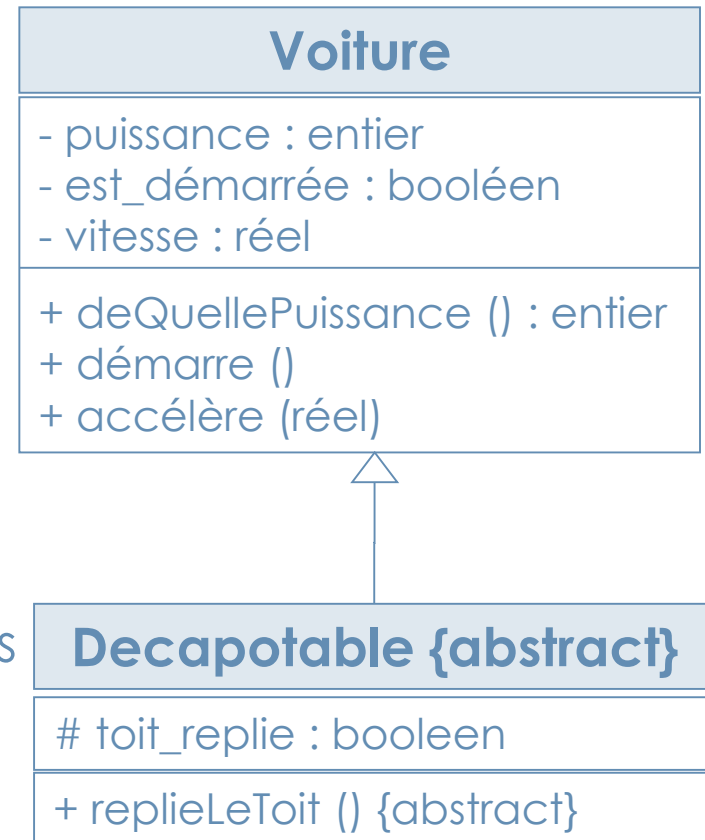
## Exemple

La classe Decapotable :

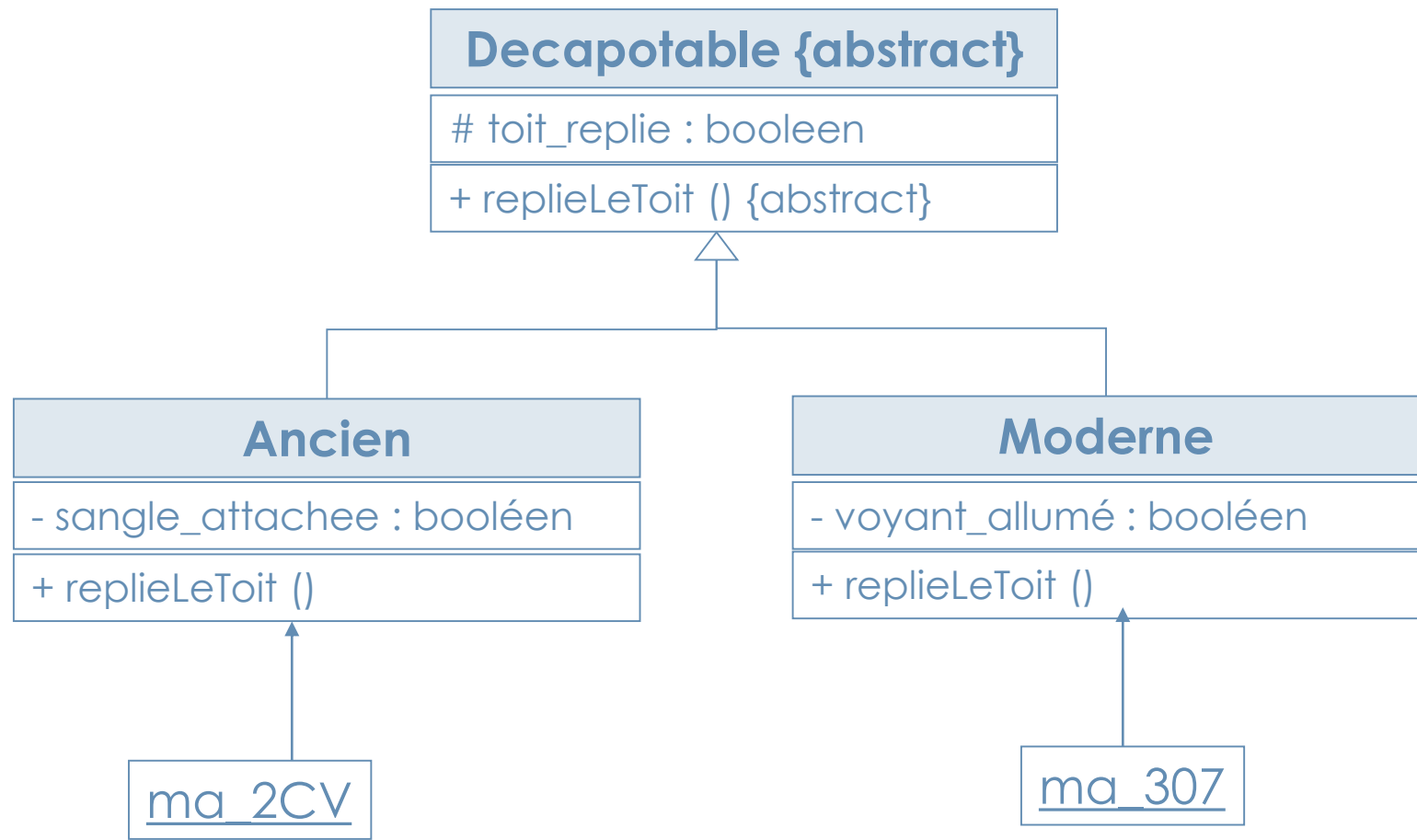
- Hérite de Voiture
- Définit un attribut
- Définit une méthode abstraite

Caractéristiques :

- Méthodes peuvent être muettes (corps vide)
- Pas d'instance



# Classes abstraites



# Classes abstraites

```
public abstract class Decapotable extends Voiture {  
    protected boolean toit_replie;  
    public abstract void replieToit ();  
}
```

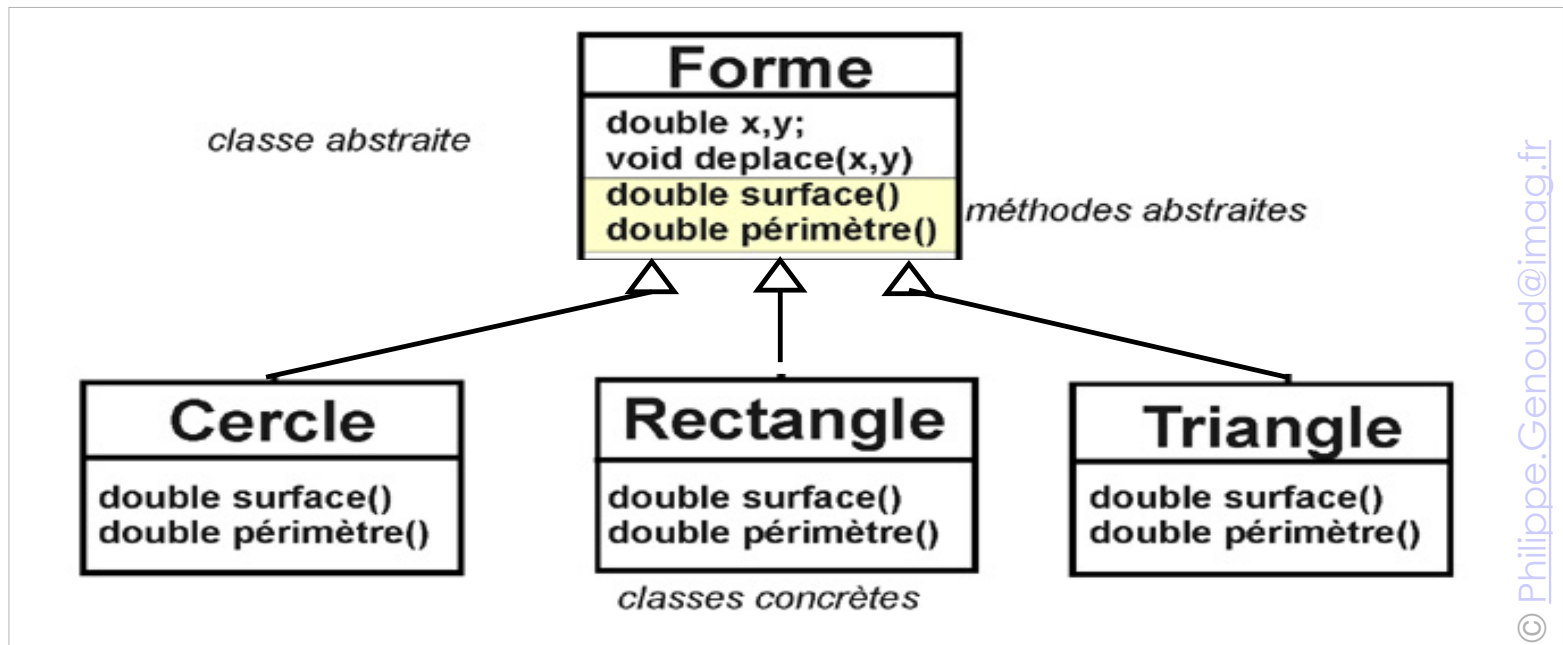
```
public class Ancien extends Decapotable {  
    private boolean sangle_attachee;  
  
    public void replieToit () {  
        this.toit_replie = true;  
        this.sangle_attachee = true;}}}
```

```
public class Moderne extends Decapotable {  
    private boolean voyant_allume;  
  
    public void replieToit () {  
        this.toit_replie = true;  
        this.voyant_allume = false;}}}
```



# Classes abstraites

Autre exemple classique :



© Philippe.Genoud@imag.fr

# Classes abstraites

---

- Une classe abstraite ne peut être instanciée.
- Si une sous-classe d'une classe abstraite fournit une implémentation pour chacune des méthodes abstraites dont elle hérite est alors une ***classe "concrète"*** et elle peut être instanciée.
- Si une sous-classe d'une classe abstraite n'implémente pas toutes les méthodes abstraites dont elle hérite, elle est elle-même une classe abstraite.

## Remarque :

Les classes abstraites sont intéressantes parce que Java est polymorphique.

Dans l'exemple classique des formes géométriques : on peut créer des tableaux de « Forme » et imaginer des traitements sur les futurs éléments qu'il contiendra (Cercles, rectangles, ...) qui ne seront pas des objets de la classe Forme.

- PROBLEME : une classe ne peut hériter que d'une seule classe abstraite
- SOLUTION : les interfaces
- On peut voir une interface comme un comportement commun à plusieurs classes

Ne pas confondre avec  
les interfaces graphiques



**Définition** : Une interface est une classe qui ne possède que des méthodes publiques abstraites (*et des variables statiques finales*)

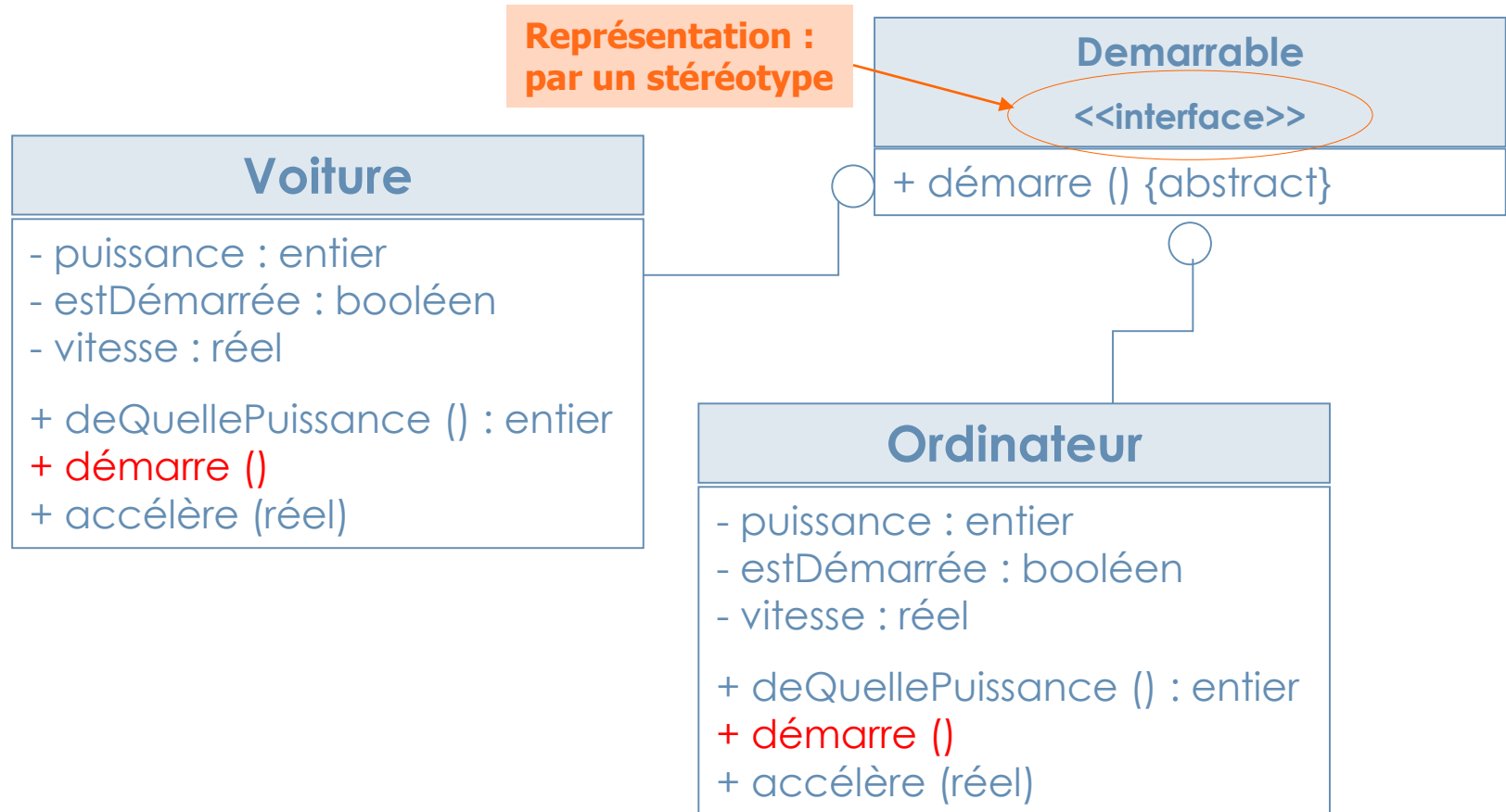
## Conséquences :

- pas de données
- pas de mot clef **abstract**
- pas besoin de mentionner **public**
- pas instanciable

## Caractéristiques :

- héritage par le mot clef **implements**
- une classe peut **implémenter/réaliser** plusieurs interfaces
- une classe implémentant une interface doit spécifier le corps de **toutes** les méthodes de l'interface

# Interfaces



# Les Interfaces

Gain de productivité et de simplicité :  
les interfaces permettent d'écrire un code plus propre  
et plus proche de la philosophie Objet.

Une interface se définit comme la

*"logique intermédiaire entre deux traitements distincts",*

ou plus généralement comme une

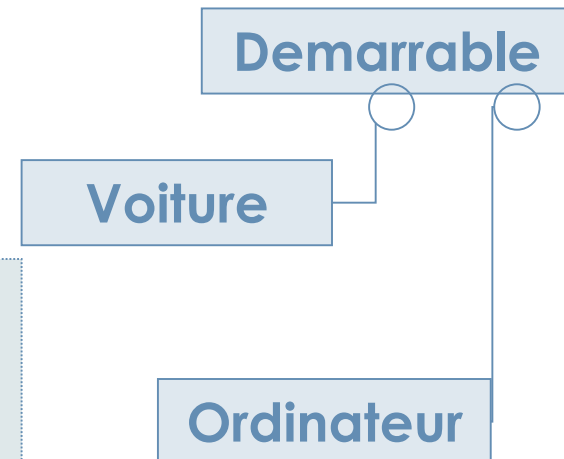
*"zone de contact et d'échange".*

# Interfaces

```
public interface Demarrable {  
    void demarre();}
```

```
public class Voiture implements Demarrable {  
    ...  
    void demarre(){  
        est_demarre = true;  
    }  
}
```

```
public class Ordinateur implements Demarrable {  
    ...  
    void demarre(){  
        System.out.println("Hello");  
    }  
}
```





***Java : héritage et polymorphisme***

***Questions***



# Plan du cours



## ***JAVA***

1. Syntaxe de base
2. Classes et objets
3. E/S
4. Héritage/Polymorphisme
5. **Normes de développement**
6. Exceptions

# N. D  v. : Pourquoi des conventions ?

---

- 80% du temps de code : maintenance et   volution
- C'est rarement l'auteur qui modifie le code
- But : faciliter la compr  hension et donc la maintenance du code
- Pas de r  gles absolues

## R  f  rence

<http://java.sun.com/docs/codeconv/index.html>

# N. Dév. : Fichiers

- Un fichier = une seule classe publique
- Nom du fichier = nom de la classe
- Pas de caractères spéciaux ou accentués
- Extension : .java (à cause de javac)

```
public class Voiture  
{  
    ...  
}
```



Voiture.java



**Respecter la casse**

# N. Dév. : Fichiers : structuration

---

- Max : 2 000 lignes
- Contenu (dans l'ordre)
  - commentaire sur le fichier
  - déclaration et importation de packages
  - déclaration de la classe

```
/**  
 * Nom de classe : MaClasse  
 * Description : <description de la classe et de son rôle>  
 * Version : 1.0  
 * Date : 10/03/2005  
 * Copyright : <votre nom>  
 */
```

# N. Dév. : Classes : structuration

---

## Ordre des éléments d'une classe :

- commentaires au format **javadoc** de la classe
- déclaration de la **classe**
- **variables de classe** (static) dans l'ordre d'accessibilité : public, protected, private
- **variables d'instance** dans l'ordre d'accessibilité : public, protected, private
- déclaration du/des **constructeurs**
- déclaration des **méthodes** regroupées par fonctionnalités

# N. Dév. : Commentaires

- Deux types
  - commentaires de **documentation** (javadoc de la JDK : génération automatique de pages html)
  - commentaires de **traitements** : précision sur le code lui-même
- Ne pas encadrer les commentaires (ex : lignes entières d'étoiles)

```

/*****
 * Nom de classe : MaClasse      **
 * Description : description     **
 * Version : 1.0                 **
 * Date : 18/02/2004             **
 * Copyright : fabrice           **
 *****/

```

# N. Dév. : Commentaires de documentation

## Classes, constructeurs, méthodes et champs

- Compris entre `/**` et `*/`
  - Première ligne : uniquement `/**`
  - Suivantes : un espace suivi d'une étoile
  - Dernière ligne : uniquement `*/` précédé d'un espace.

```
/**  
 * Description de la methode  
 */  
public void maMethode() {
```

- **Javadoc** définit des **tags** permettant de typer certaines informations (utilisation possible de balises HTML)
- L'entité documentée est précédée par son commentaire



# N. Dév. : Commentaires : méthode

@see, @param, @return, @exception, @author

```
/**
 * nomMethode - description de la méthode
 *               - explication supplémentaire si nécessaire
 *               - exemple d'appel de la methode
 *
 * @return description de la valeur de retour
 * @param arg1 description du 1er argument
 * [...]
 * @param argN description du Neme argument
 * @exception Exception1 description de la première exception
 * [...]
 * @exception ExceptionN description de la Neme exception
 *
 * @see UneAutreClasse#UneAutreMethode
 * @author John Woo
 * @date 12/02/2004
 */
```

Object (Java 2 Platform SE v1.4.2) - Microsoft Internet Explorer

Fichier Edition Affichage Favoris Outils ? Adresse http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Ob OK Liens »

**Overview Package Class Use Tree Deprecated Index Help**

[PREV CLASS](#) [NEXT CLASS](#) [FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: NESTED | FIELD | [CONSTR](#) | [METHOD](#) DETAIL: FIELD | [CONSTR](#) | [METHOD](#)

*Java™ 2 Platform Std. Ed. v1.4.2*

---

java.lang

## Class Object

java.lang.Object

---

public class Object

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Since:  
JDK1.0

See Also:  
[Class](#)

---

### Constructor Summary

[Object](#) ()

---

### Method Summary

protected <a href="#">Object</a>	<a href="#">clone</a> ()	Creates and returns a copy of this object.
boolean	<a href="#">equals</a> ( <a href="#">Object</a> obj)	Indicates whether some other object is "equal to" this one.
protected void	<a href="#">finalize</a> ()	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
<a href="#">Class</a>	<a href="#">getClass</a> ()	Returns the runtime class of an object.
int	<a href="#">hashCode</a> ()	Returns a hash code value for the object.
void	<a href="#">notify</a> ()	Wakes up a single thread that is waiting on this object's monitor.
void	<a href="#">wait</a> ()	Causes the current thread to wait until another thread calls <a href="#">notify</a> or <a href="#">notifyAll</a> on this object.

@see

@param

@return

# N. Dév. : Commentaires de code

- But : Ajouter du sens et des précisions au code
- Commentaires
  - sur une ligne `/* */`
  - sur une portion de ligne `//`
  - multi-lignes `/* */`
- Utiliser des espaces entre les délimiteurs
- Laisser un espace conséquent entre le code et les commentaires



**EXPLIQUER n'est pas  
TRADUIRE**

```
Voiture ma_voiture;      // commentaire
...
Voiture ta_voiture;      /* commentaire
... */
```

# N. Dév. : Déclaration : Variables

- Ne pas mettre d'accent, ni de caractère spécial
- Une seule déclaration d'entité par ligne

```
String nom;           // Ok
String prenom;        // Ok
String nom, prenom;   // A éviter
```



- Aligner les types, identifiants et commentaires

```
String nom;           // nom de l'eleve
String prenom;        // prenom de l'eleve
int    carnet_notes[]; // notes de l'eleve
```

# N. Dév. : Déclaration : Variables

✓ Rassembler toutes les déclarations d'un bloc au début de ce bloc (sauf boucle for)

```
for (int i = 0 ; i < 9 ; i++) {  
    ...  
}
```

```
int taille;  
...  
void maMethode() {  
    int taille;  
    ...  
}
```

✓ Eviter de déclarer une variable masquant la variable d'un bloc englobant

# N. Dév. : Déclaration : méthodes

- Accolade ouvrante de début de bloc : fin de la ligne de déclaration
- Accolade fermante : ligne séparée, indentation de la déclaration
- Déclaration de méthode précédée d'une ligne blanche

```
class MaClasse extends MaClasseMere {  
    String nom;  
    String prenom;  
  
    MaClasse(String nom, String prenom) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
  
    void neRienFaire() {}    // Exception  
}
```

# N. Dév. : Déclaration : Constructeur

- Tjrs définir explicitement le constructeur par défaut (sans arg.)
- Tjrs initialiser les variables d'instance dans le constructeur
- Tjrs appeler explicitement le constructeur hérité lors de la redéfinition d'un constructeur dans une classe fille (avec super)

```
class MaClasse extends MaClasseMere {
    String nom;
    String prenom;
    int    age;
    MaClasse() {
        this.nom = n;
        this.prenom = p;
        this.age = this.calculeAge(d);
    }
    int calculeAge(Date d) {
        ...
    }
}
```

# N. Dév. : Règles de "Nommage"

- Rend les programmes plus lisibles
- Identification rapide de l'entité désignée (variable, méthode, constante, ...)

## Règles

- **Packages** : en minuscules
- **Classes** : première lettre de chaque mot en majuscule (pas de \_)
- **Méthodes** : verbe, première lettre en minuscule, première lettre des mots suivants en majuscule (pas de \_)
  - Accesseur : get + nom du champ
  - Modifieur : set + nom du champ
  - Conversion : to + nom de la classe renvoyée
- **Variables**
  - Première lettre en minuscule, première lettre des mots suivants en majuscule (pas de \_)
  - Eviter les variables avec un seul caractère (exception : variables de boucle)
- **Constantes** : en majuscules, mots séparés par \_



# N. D  v. : "Nommage"

```
package monpackage;    // un package
```

```
class MaClasse extends MaClasseMere {    // une classe  
    ...
```

```
public void augmenterVitesse(int v) {    // une m  thode  
    ...
```

```
public int getVitesse() {                // accesseur  
public void setVitesse(int v) {          // modifieur
```

```
/* Variables */  
int      maVariable;  
Voiture uneVariableDeLaClasseVoiture;
```

```
/* Constantes */  
static double PI = 3.1416;  
static int     HAUTEUR_MAX = 5;
```

# N. D  v. : S  parateurs

## But : Rendre le code moins dense

- **Indentation** (reconnaissance des blocs)
  - 4 espaces (pas de tabulations)
  - Eviter les lignes contenant plus de 80 caract  res
- **Lignes blanches** (s  parations logiques)
  - avant la d  claration d'une m  thode
  - entre la d  clarations des variables locales et la 1  re ligne de code
  - avant un commentaire d'une seule ligne
  - avant chaque section logique dans le code d'une m  thode
- **Espaces**
  - entre un mot cl   et une parenth  se
  - apr  s chaque virgule dans une liste d'arguments
  - de chaque c  t   d'un op  rateur binaire ex :  $a = (b + c) * d$
  - pas entre un nom de m  thode et sa parenth  se ouvrante
  - pas avant les op  rateurs unaires (ex ++)
- **Coupure de lignes** (lignes >    80 caract  res)
  - couper la ligne apr  s une virgule ou avant un op  rateur
  - aligner le d  but de la nouvelle ligne au d  but de l'expression coup  e

# N. Dév. : Structures de contrôle

```
if (condition) {  
    traitements;  
} else if (condition) {  
    traitements;  
} else {  
    traitements;  
}
```

```
switch (condition) {  
    case ABC:    traitements;  
    case DEF:    traitements;  
    case XYZ:    traitements;  
    default:     traitements;  
}
```

```
for ( initialisation; condition; mise à jour) {  
    traitements;  
}
```

```
while (condition) {  
    traitements;  
}
```

```
try {  
    traitements;  
} catch (Exception1 e1) {  
    traitements;  
} catch (Exception2 e2) {  
    traitements;  
} finally {  
    traitements;  
}
```

# N. Dév. : Parenthèses

Il est préférable d'utiliser les parenthèses lors de l'usage de plusieurs opérateurs pour éviter des problèmes liés à la priorité des opérateurs.

<del>if (i == j &amp;&amp; m == n)</del>	// à éviter
if ( (i == j) && (m == n) )	// à utiliser

# N. Dév. : Choisir une langue

---

IMPERATIF : Ne pas mélanger les langues.

CONSEIL : Développer en anglais.

## *Java : Normes*

*Questions*



# Plan du cours



## ***JAVA***

1. Syntaxe de base
2. Classes et objets
3. E/S
4. Héritage/Polymorphisme
5. Normes de développement
6. **Exceptions**

**Définition** : une exception est un signal qui indique que quelque chose d'exceptionnel (comme une erreur) s'est produit. Elle interrompt le flot d'exécution normal du programme.

- **lancer** (*throws*) une exception consiste à signaler ce quelque chose
- **capturer** (*catch*) une exception permet d'exécuter les actions nécessaires pour traiter cette situation



# Exceptions : Intérêt

---

## Gérer les erreurs est indispensable

Mauvaise gestion peut avoir des conséquences catastrophiques

## Un mécanisme simple et lisible

- Regroupement du code réservé au traitement des erreurs (pas de "mélange" avec l'algorithme)
- Possibilité de "récupérer" une erreur à plusieurs niveaux d'une application (propagation dans la pile des appels de méthodes)

# Exceptions : Exemples

```
public class TestException {  
    public static void main(java.lang.String[] args) {  
        int i = 3;  
        int j = 0;  
        System.out.println("résultat = " + (i / j));  
    }  
}
```

Exception in thread "main" java.lang.ArithmeticException: / by zero at tests.TestException.main(TestException.java:23)

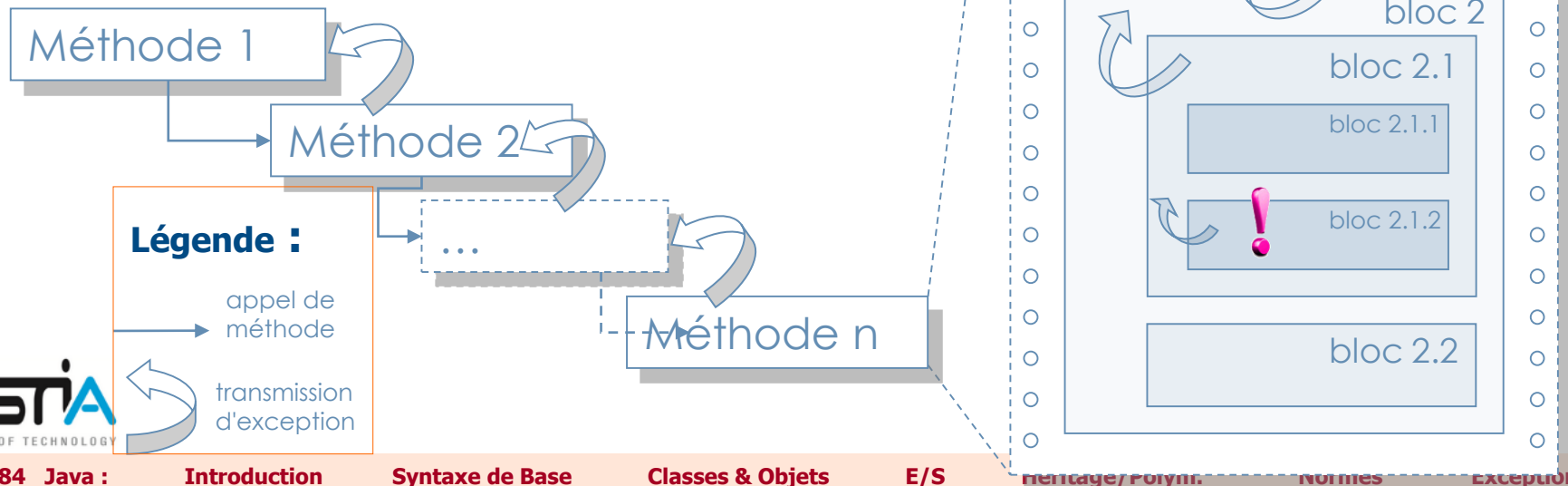
```
public class TestExceptionPoint {  
    public static void main(java.lang.String[] args) {  
        Voiture maVoiture;  
        maVoiture.accelere(30);  
    }  
}
```

Exception in thread "main" java.lang.NullPointerException: at tests.TestExceptionPoint.main(TestExceptionPoint.java:24)

# Exceptions : Principe

Lorsqu'une situation exceptionnelle est rencontrée, une exception est lancée

Si elle n'est pas traitée, elle est transmise au **bloc** englobant, ..., jusqu'à ce qu'elle soit traitée ou parvienne en haut de la pile d'appel. Elle stoppe alors l'application.

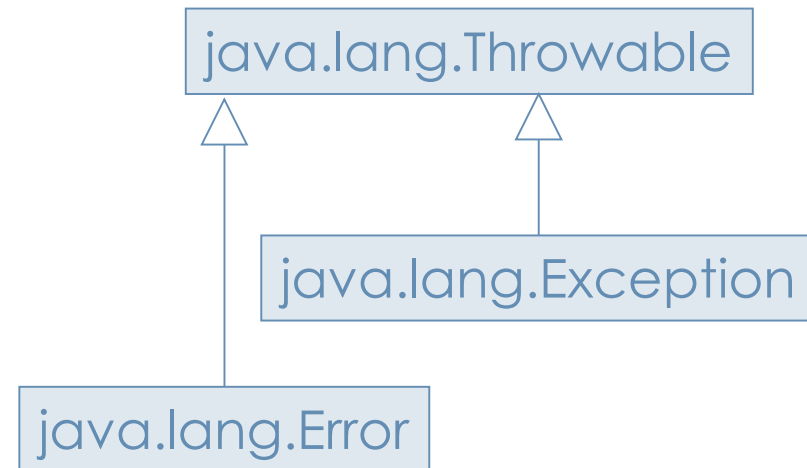


# Exceptions = Objet

## Une exception est une instance

- soit de la classe `java.lang.Error` : « erreurs systèmes » donc non récupérable
- soit de la classe `java.lang.Exception` : erreur récupérable

```
IOException  
NullPointerException  
OutOfMemoryException  
ArithmeticException  
ClassCastException  
ArrayIndexOutOfBoundsException  
NegativeArraySizeException  
EOFException  
...
```



# Exceptions : Gestion

## En Java : 3 blocs distincts

- **bloc try** : instructions susceptibles de déclencher une (des) exception(s) pour la(les)quelle(s) une gestion est mise en œuvre
- **blocs catch** : capturent les exceptions dont le type est spécifié et exécuter des instructions spécifiques
- **bloc finally** (optionnel) : instructions de "nettoyage", exécuté quelle que soit le résultat du bloc try (i.e. qu'il ait déclenché une exception ou non)

# Exceptions : Gestion

```
try {  
    operation_risqueel;  
    operation_risqueel2;  
} catch (ExceptionInteressante e) {  
    /* bloc de traitement interessant */  
} catch (ExceptionParticuliere e) {  
    /* bloc de traitement particulier */  
} catch (Exception e) {  
    /* bloc de traitement general */  
} finally {  
    /* bloc de traitement final */  
}
```

# Exceptions : Exemple

```
public class TestException {  
    public static void main(String[] args) {  
        int i = 3;  
        int j = 0;  
  
        try {  
            System.out.println("résultat = " + (i / j));  
        } catch (ArithmeticException e) {  
            System.out.println("Attention, division par 0");  
        } finally {  
            System.out.println("On passe à la suite ...");  
        }  
        .../...  
    }  
}
```

# Exceptions : Gestion

**catch**

```
catch (Throwable e) { }
```

Chaque clause catch doit être déclarée avec un argument de type Throwable ou une sous-classe de Throwable

Dans le cas où plusieurs clauses catch s'enchaînent, seule la première correspondante à l'exception levée est traitée

```
public class TestException {  
    public static void main(String[] args) {  
        int i = 3;  
        int j = 0;  
        try {  
            System.out.println("résultat = " + (i / j));  
        } catch (Exception e) {  
            ...  
        } catch (ArithmeticException e) {  
            ...  
        }  
    }  
}
```



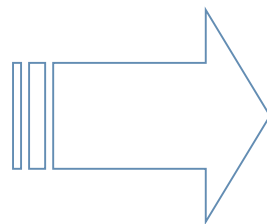
# Exceptions : Gestion

## finally

- Permet de spécifier du code dont l'exécution est garantie quoi qu'il arrive
- Intérêt double
  - Rassembler dans un seul bloc un ensemble d'instructions qui autrement auraient du être dupliquées
  - Effectuer des traitements après le bloc try, même si une exception a été levée et non attrapée par les blocs catch

# Il est possible de créer des exceptions

- Utiliser l'héritage pour définir de nouveaux types d'exceptions (`java.lang.Exception`)
- Puisqu'elles sont des objets les exceptions peuvent contenir :
  - des attributs particuliers
  - des méthodes



Il est possible d'enrichir  
soi-même le format  
des exceptions

# Créer ses propres exceptions

```
public class MonException extends Exception {  
    public MonException(String text) {  
        super(text);  
    }  
    public String toString() {  
        ...  
    }  
    ...  
}
```

Définition,  
hérite de  
Exception

Utilisation

```
throws new MonException("blabla");
```

# Exceptions : Déclaration

---

Toute **méthode** susceptible de lever une exception\* doit soit :

- La capturer (bloc try/catch)
- La « laisser passer » en indiquant son type dans sa signature par la clause **throws**

*\* ou bien parce qu'elle utilise "throws" ou bien parce qu'elle utilise une méthode qui utilise throws, ...*

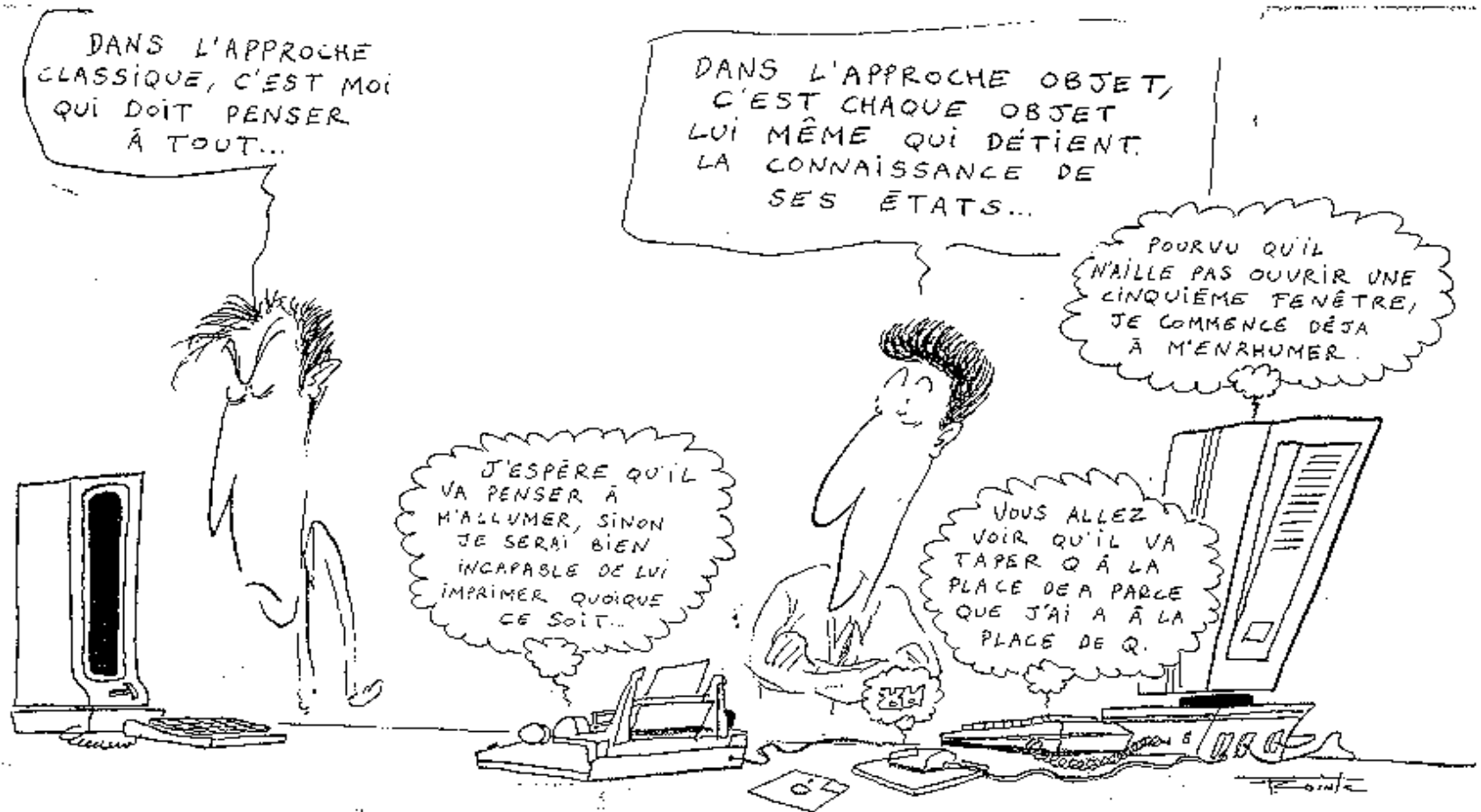
# Exceptions : Exemple

```
public class MaClasse {  
    public void uneMethode() throws MonException {  
        /* ... Traitements */  
        MonException e = new MonException();  
        throw e;  
        /* Autres traitements ... */  
    }  
  
    /* Cette méthode déclenche une erreur de compilation */  
    public void autreMethode() {  
        /* ... Traitements */  
        this.uneMethode();  
        /* Autres traitements ... */  
    }  
  
    /* Celle-là non ... */  
    public void encoreUneAutreMethode() throws MonException{  
        /* ... Traitements */  
        this.uneMethode();  
        /* Autres traitements ... */  
    }  
}
```

# POO: Conclusion

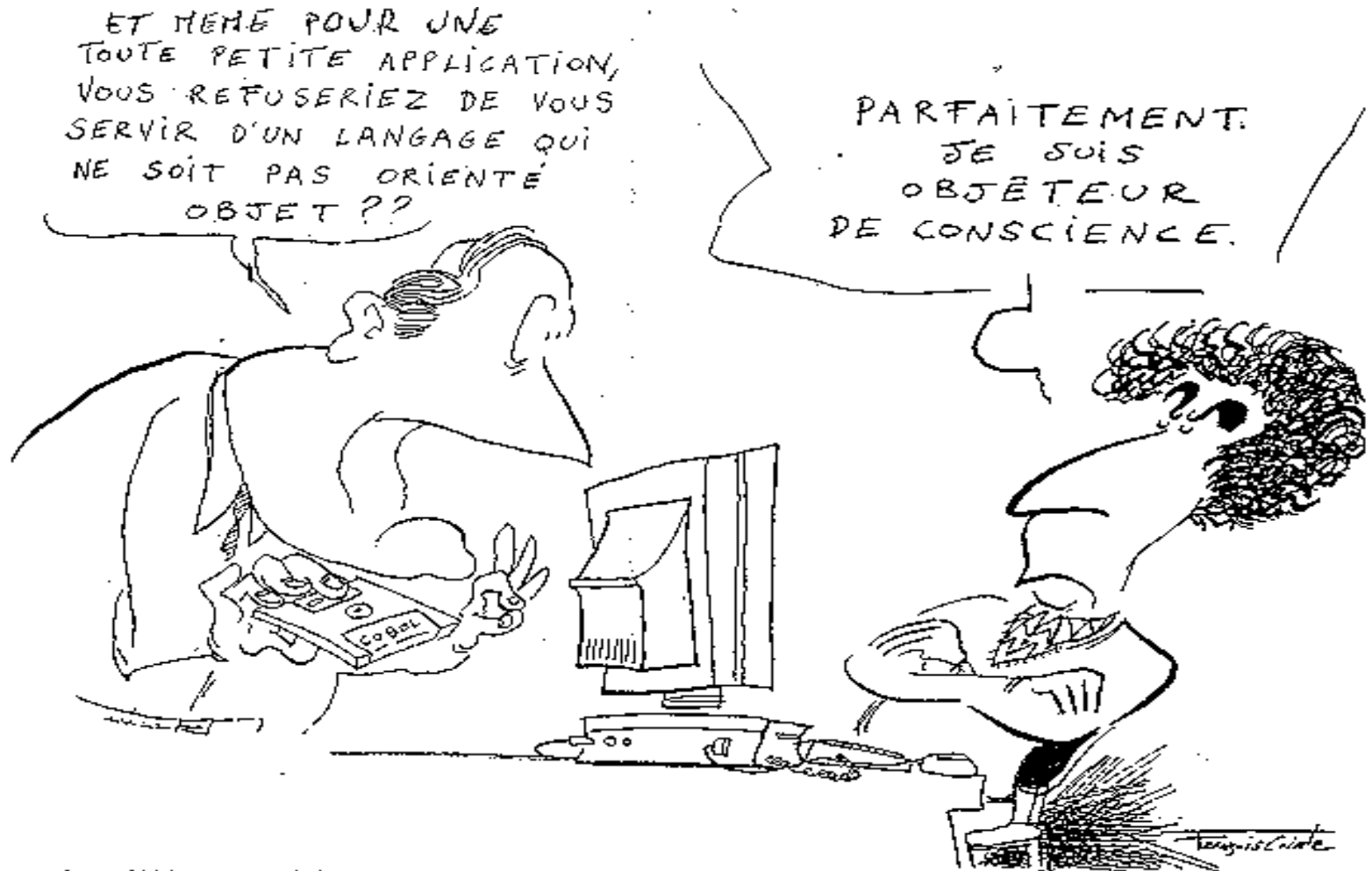
*C'est terminé !*

*Clin d'œil !*



# POO: Conclusion

## Clin d'œil !



## *Java : exceptions*

*Questions*





# FIN



*Questions*

