Programmation orientée objets en Java

Salima Hassas

Transparents de Mickaël BARON (SERLI Informatique)

Déroulement du cours

- Structuration du cours
  - Présentation des concepts
  - Illustration avec de nombreux exemples
  - Des bulles d'aide tout au long du cours :

Ceci est une alerte

Ceci est une astuce

- Mise en place du cours
  - Ancien cours de Francis Jambon
  - Cours de Fabrice Depaulis
  - Livre : Programmer en Java 2ème édition – Claude Delannoy - Eyrolles
  - Internet : *www.developpez.com*

Organisation ...

- Partie 1 : Introduction au langage JAVA
- Partie 2 : Bases du langage
- Partie 3 : Classes et objets
- Partie 4 : Héritage
- Partie 5 : Héritage et polymorphisme
- Partie 7 : Les indispensables : package, collections et exception

Introduction au langage Java

Rapide historique de Java

- Origine
  - Créé par Sun Microsystems
  - Cible : les systèmes embarqués (véhicules, électroménager, etc) utilisant des langages dédiés incompatibles entre eux
- Dates clés
  - **1991** : Introduction du langage « Oak » par James Gosling
  - **1993** : Montée en puissance du Web grâce à Mosaic (l'idée d'adapter Java au Web fait son chemin)
  - **1995** : Réalisation du logiciel HotJava en Java permettant d'exécuter des *applets*
  - **1996** : Netscape™ Navigator 2 incorpore une machine virtuelle Java 1.0 en version « beta »
  - **1997** : Un premier pas vers une version industrielle Java 1.1
  - **1999** : Version industrielle de Java

Sun voit Java comme ...

- Références
  - WikiPedia : *fr.wikipedia.org/wiki/java_%28technologie%29*
  - White papers : *java.sun.com/docs/white/index.html*
- Sun définit le langage Java comme
  - Simple
  - Orienté objet
  - Réparti
  - Interprété
  - Robuste
  - Sûr
  - Portable
  - Performant
  - Multitâches
  - Dynamique ...

Principe de fonctionnement de Java

- Source Java
  - Fichier utilisé lors de la phase de programmation
  - Le seul fichier réellement intelligible par le programmeur!
- Byte-Code Java
  - Code objet destiné à être exécuté sur toute « Machine Virtuelle » Java
  - Provient de la compilation du code source
- Machine Virtuelle Java
  - Programme interprétant le Byte-Code Java et fonctionnant sur un système d'exploitation particulier
  - Conclusion : il suffit de disposer d'une « Machine Virtuelle » Java pour pouvoir exécuter tout programme Java même s 'il a été compilé avec un autre système d'exploitation

Machines Virtuelles Java ...

- Navigateurs Web, Stations de travail, Network Computers
- WebPhones
- Téléphones portables
- Cartes à puces
- ...

Principales étapes d'un développement

- Création du code source
  - A partir des spécifications (par exemple en UML)
  - Outil : éditeur de texte, IDE
- Compilation en Byte-Code
  - A partir du code source
  - Outil : compilateur Java
- Diffusion sur l'architecture cible
  - Transfert du Byte-Code seul
  - Outils : réseau, disque, etc
- Exécution sur la machine cible
  - Exécution du Byte-Code
  - Outil : Machine Virtuelle Java

Java et ses versions ...

- Différentes versions de la machine virtuelle
  - Java 2 Micro Edition (Java ME) qui cible les terminaux portables
  - Java 2 Standard Edition (Java SE) qui vise les postes clients
  - Java 2 Enterprise Edition (Java EE) qui définit le cadre d'un serveur d'application

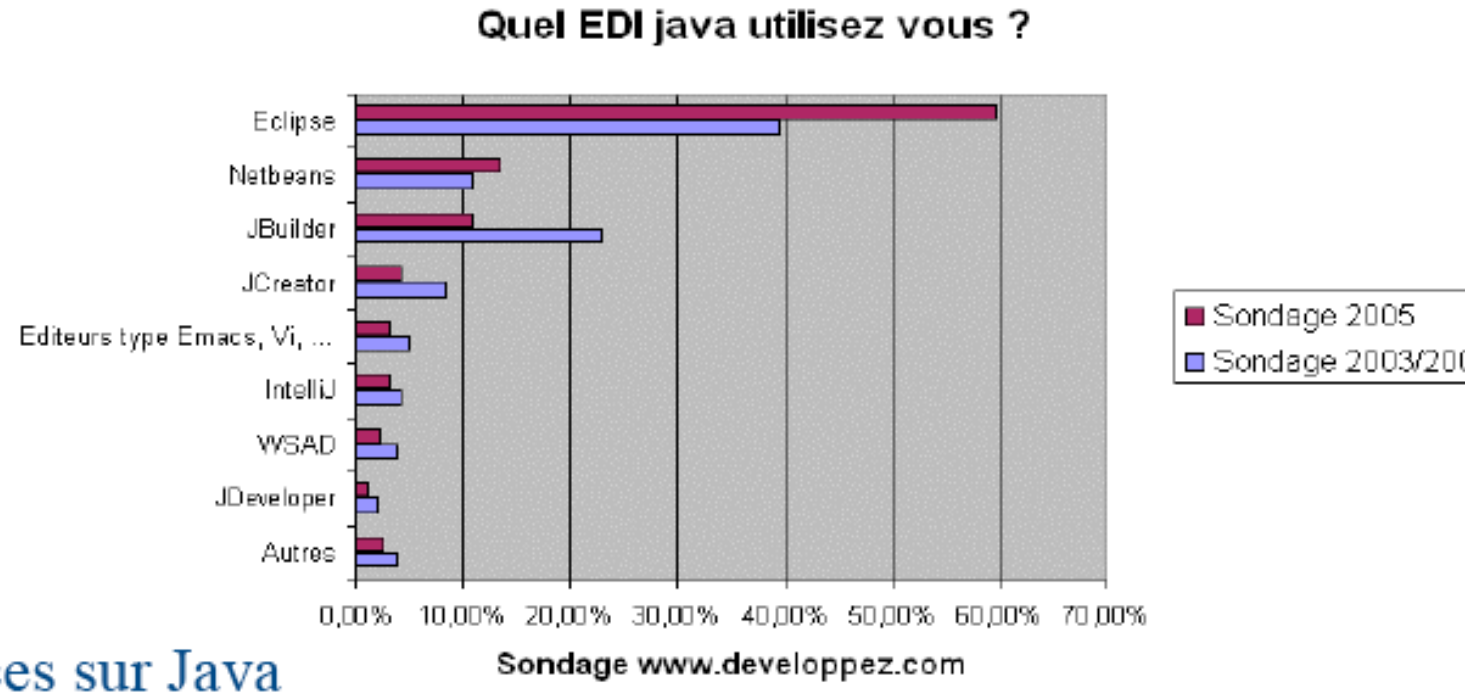
Dans la suite du cours, on va s'intéresser principalement aux API fournies par Java SE

- Différentes finalités
  - SDK (Software Development Kit) fournit un compilateur et une machine virtuelle
  - JRE (Java Runtime Environment) fournit uniquement une machine virtuelle. Idéal pour le déploiement de vos applications.

- Version actuelle de Java
  - Actuellement « Java SE 5.0 » ou encore appelée « JDK 5.0 »
  - Bientôt Java SE 6.0 (aperçue rapide en fin du cours de Java)

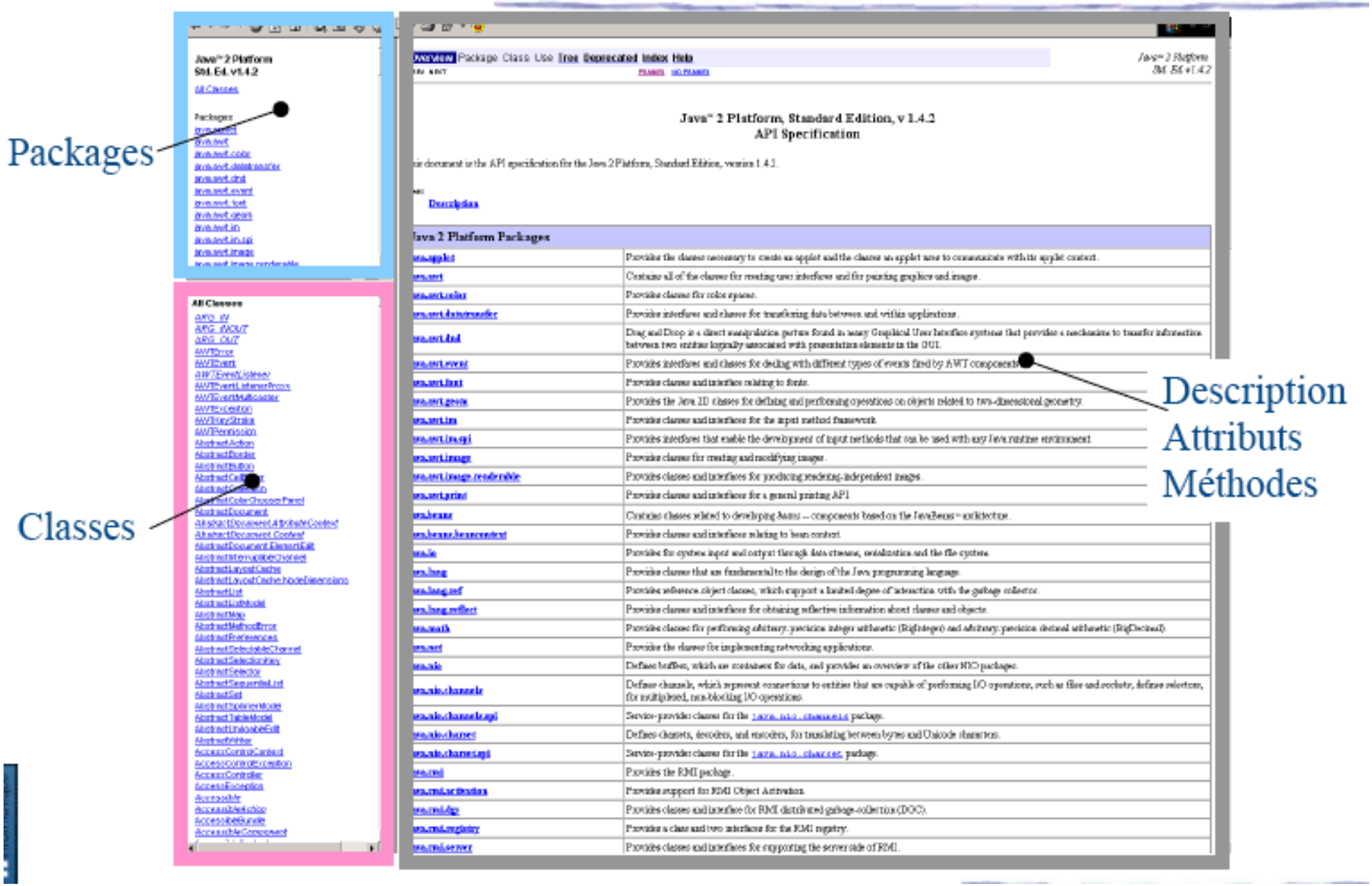
Les outils ...

- Simple éditeurs ou environnements de développement
  - Eclipse
  - NetBeans
  - JBuilder
  - …



- Les ressources sur Java
  - Site de Java chez Sun : *java.sun.com*
  - API (référence) : *java.sun.com/j2se/1.5.0*
  - Tutorial de Sun : *java.sun.com/doc/bookstutorial*
  - Cours et exemples : *java.developpez.com*
  - Forum : *fr.comp.lang.java*

L'API de Java


Packages
Classes
Description
Attributs
Méthodes
Java™ 2 Platform
Std. Ed. v1.4.2
All Classes
Packages
All Classes
Java™ 2 Platform, Standard Edition, v 1.4.2
API Specification
Java 2 Platform Packages

Ouvrages d'initiation

- Programmer en Java (2ème édition)
  - Auteur : Claude Delannoy
  - Éditeur : Eyrolles
  - Edition : 2002 - 661 pages - ISBN : 2212111193

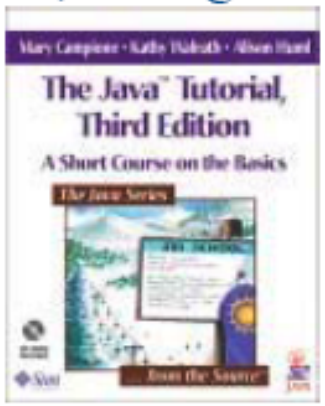
- Java en action
  - Auteur : Ian F. Darwin
  - Éditeur : O'Reilly
  - Edition : 2002 - 836 pages - ISBN : 2841772039

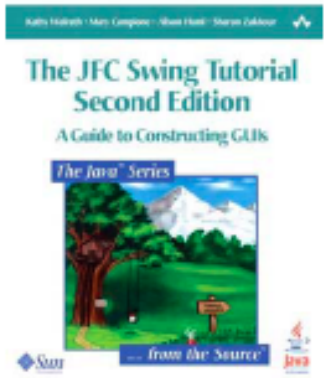
- Apprendre Java et C++ en parallèle
  - Auteur : Jean-Bernard Boichat
  - Éditeur : Eyrolles
  - Edition : 2003 - 742 pages - ISBN : 2212113277

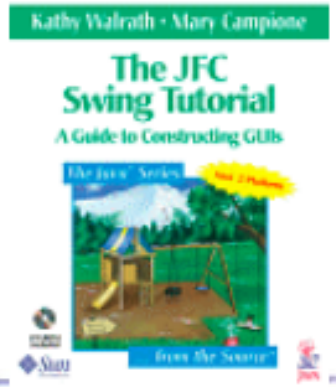
Ouvrages de référence

- Ouvrages thématiques aux éditions O'Reilly sur une sélection des Packages Java (certains traduits en Français)

- Ouvrages de référence de SUN aux éditions Paperback (en anglais uniquement)







Programmation Orientée Objet application au langage Java

Bases du langage

Premier exemple de programme en Java

```
public class PremierProg {

    public static void main (String[] argv) {
        System.out.println("Ola, mon Premier Programme");
    }
}
```

- *public class PremierProg*
  - Nom de la classe
- *public static void main*
  - La fonction principale équivalent à la fonction *main* du C/C++
- *String[] argv*
  - Permet de récupérer des arguments transmis au programme au moment de son lancement
- *System.out.println("Ola ... ")*
  - Méthode d'affichage dans la fenêtre console

Mise en œuvre

- Pas de séparation entre définition et codage des opérations
  - Un seul fichier « NomDeClasse.java »
  - Pas de fichier d'en tête comme C/C++

Nom de la classe = Nom du fichier java

- Compilation
  - *javac* NomDeClasse.java ou *javac* *.java quand plusieurs classes
  - Génération d'un fichier Byte-Code « NomDeClasse.class »
  - Pas d'édition de liens (seulement une vérification)

Ne pas mettre l'extension .class pour l'exécution

- Exécution
  - *java* NomDeClasse
  - Choisir la classe principale à exécuter

Types primitifs de Java

- Ne sont pas des objets !!!
- Occupent une place fixe en mémoire réservé à la déclaration
- Types primitifs :
  - Entiers : **byte** (1 octet) - **short** (2 octets) - **int** (4 octets) - **long** (8 octets)
  - Flottants (norme IEEE-754) : **float** (4 octets) - **double** (8 octets)
  - Booléens : **boolean** (true ou false)
  - Caractères : **char** (codage Unicode sur 16 bits)
- Chacun des types simples possède un alter-ego objet disposant de méthodes de conversion (à voir dans la partie Classes et Objets)
- L'autoboxing introduit depuis la version 5.0 convertit de manière transparente les types primitifs en référence

Initialisation et constantes

- Initialisation
  - Une variable peut recevoir une valeur initiale au moment de sa déclaration :

```
int n = 15;
boolean b = true
```

  - Cette instruction joue le même rôle :

```
int n;
n = 15;
boolean b;
b = true;
```

Penser à l'initialisation au risque d'une erreur de compilation

```
int n;
System.out.println(" n = " + n);
```

- Constantes
  - Ce sont des variables dont la valeur ne peut affectée qu'une fois
  - Elles ne peuvent plus être modifiées
  - Elles sont définies avec le mot clé **final**

```
final int n = 5;
final int t;
...
t = 8;
n = 10; // erreur : n est déclaré final
```

Structure de contrôle

- Choix
  - Si alors sinon : « **if** *condition* {…} **else** {…} »

Il n'y a pas de mot-clé « then » dans la structure Choix

- Itérations
  - Boucle : « **for** (*initialisation* ; *condition* ; *modification*) { … } »
  - Tant que : « **while** (*condition*) {…} »
  - Faire jusqu'à : « **do** {…} **while** (*condition*) »

- Sélection bornée
  - Selon faire : « **switch** *identificateur* { **case** valeur0 : … **case** valeur1 : … **default**: …} »
  - Le mot clé **break** demande à sortir du bloc

Penser à vérifier si break est nécessaire dans chaque case

Structure de contrôle

Exemple : structure de contrôle

```
public class SwitchBreak {

    public static void main (String[] argv) {
        int n = ...;
        System.out.println("Valeur de n :" + n);
        switch(n) {
            case 0 : System.out.println("nul");
                     break;
            case 1 :
            case 2 : System.out.println("petit");
            case 3 :
            case 4 :
            case 5 : System.out.println("moyen");
                     break;
            default : System.out.println("grand");
        }
        System.out.println("Adios...");
    }
}
```

Faisons varier *n* :

```
Valeur de n : 0
nul
Adios...

Valeur de n : 1
petit
moyen
Adios...

Valeur de n : 6
grand
Adios...
```

Se demander si break est nécessaire

Opérateurs sur les types primitifs

- Opérateurs arithmétiques
  - Unaires : « +a, -b »
  - Binaires : « a+b, a-b, a*b, a%b »
  - Incrémentation et décrémentation : « a++, b-- »
  - Affectation élargie : « +=, -=, *=, /= »

Attention : erreur

```
boolean t = true;
if (t = true) {...}
```

Préférer :

```
boolean t = true;
if (t) {...}
```

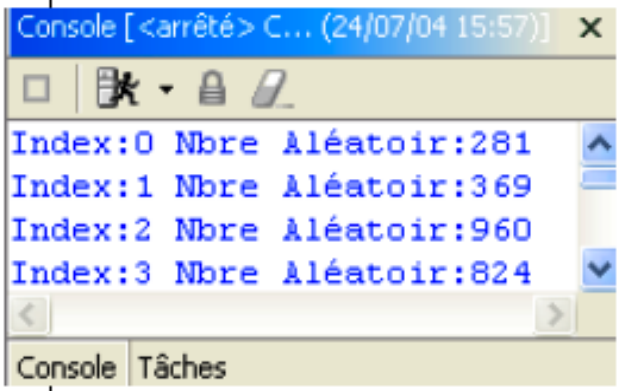
- Opérateurs comparaisons
  - « a==b, a!=b, a>b, a<b, a>=b, a<=b »
- Opérateurs logiques
  - Et : « a && b », « a & b »
  - Ou : « a || b », « a | b »
- Conversion de type explicite (cast)
  - « (NouveauType)variable »

Opérateurs sur les types primitifs

- Exemple du Loto
  - Pas optimiser mais montrer l'utilisation des concepts précédents

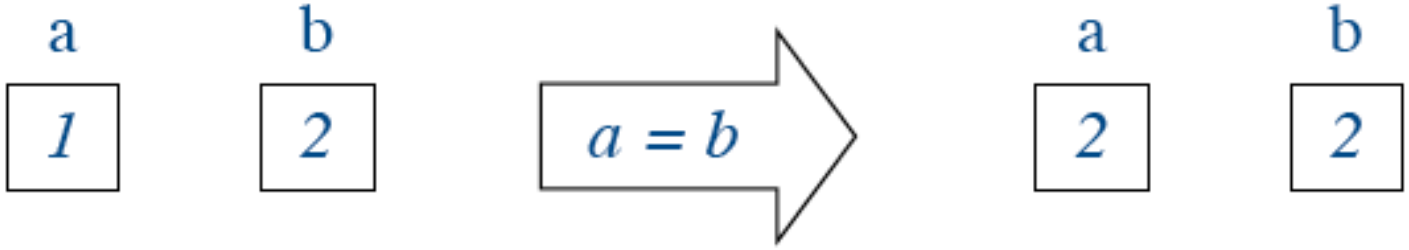
```
public class ExempleTypesPrimitifs {

    public static void main (String[] argv) {
        int compteur = 0;

        while(compteur != 100) {
            // Prend un nombre aléatoire
            double nbreAleatoir = Math.random() * 1000;

            // Etablie un index de 0 à 10
            int index = compteur % 10;

            // Construction de l'affichage
            System.out.println("Index:" + index +
                "Nbre Aléatoir:" + (int)nbreAleatoir);

            // Incrémentation de la boucle
            compteur+= 1;
        }
    }
}
```

A voir plus tard…



Affectation, recopie et comparaison

- Affecter et recopier un type primitif
  - « a=b » signifie a prend la valeur de b
  - a et b sont distincts
  - Toute modification de a n'entraîne pas celle de b
- Comparer un type primitif
  - « a == b » retourne « true » si les valeurs de a et b sont identiques



Les tableaux en Java

- Les tableaux sont considérés comme des **objets**
- Fournissent des collections ordonnées d'éléments
- Les éléments d'un tableau peuvent être :
  - Des variables d'un type primitif (int, boolean, double, char, …)
  - Des références sur des objets (à voir dans la partie Classes et Objets)
- Création d'un tableau
  - ① Déclaration = déterminer le type du tableau
  - ② Dimensionnement = déterminer la taille du tableau
  - ③ Initialisation = initialiser chaque case du tableau

Les tableaux en Java : Déclaration

① Déclaration

- La déclaration précise simplement le type des éléments du tableau

```
int[] monTableau;
```

monTableau | *null*

- Peut s'écrire également

```
int monTableau[];
```

Attention : une déclaration de tableau ne doit pas préciser de dimensions

```
int monTableau[5]; // Erreur
```

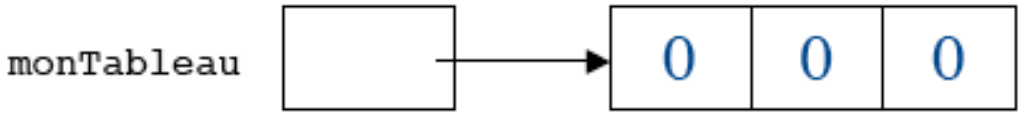
Les tableaux en Java : Dimensionnement

② Dimensionnement

- Le nombre d'éléments du tableau sera déterminé quand l'objet tableau sera effectivement créé en utilisant le mot clé **new**
- La taille déterminée à la création du tableau est fixe, elle ne pourra plus être modifiée par la suite
- Longueur d'un tableau : « monTableau.**length** »

```
int[] monTableau; // Déclaration
monTableau = new int[3]; // Dimensionnement
```

- La création d'un tableau par **new**
  - Alloue la mémoire en fonction du type de tableau et de la taille
  - Initialise le contenu du tableau à 0 pour les types simples



Les tableaux en Java : Initialisation

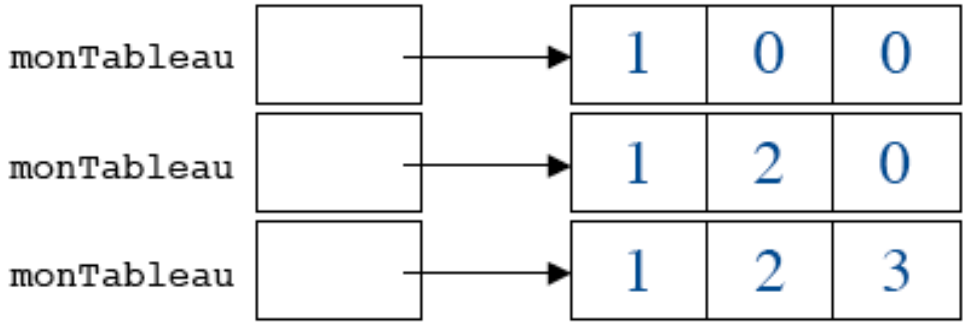
③ Initialisation

- comme en C/C++ les indices commencent à zéro
- l'accès à un élément d'un tableau s'effectue suivant cette forme

```
monTab[varInt]; // varInt >= 0 et <length
```

- Java vérifie automatiquement l'indice lors de l'accès (lève une exception)

```
monTab[0] = 1;
monTab[1] = 2;
monTab[2] = 3;
```



- Autre méthode : en donnant explicitement la liste de ses éléments entre {…}

```
int[] monTab = {1, 2, 3}
```

- est équivalent à

```
monTab = new int[3];
monTab[0] = 1; monTab[1] = 2; monTab[2] = 3;
```

Les tableaux en Java : Synthèse

① Déclaration

```
int[] monTableau;
```

② Dimensionnement

```
monTableau = new int[3];
```

③ Initialisation

```
monTableau[0] = 1;
monTableau[1] = 2;
monTableau[2] = 3;
```

Ou ①② et ③

```
int[] monTab = {1, 2, 3};
```

```
for (int i = 0; i < monTableau.length; i++) {
    System.out.println(monTableau[i]);
}
```

Les tableaux en Java : Multidimensionnels

- Tableaux dont les éléments sont eux mêmes des tableaux
- Déclaration

```
type[][] monTableau;
```

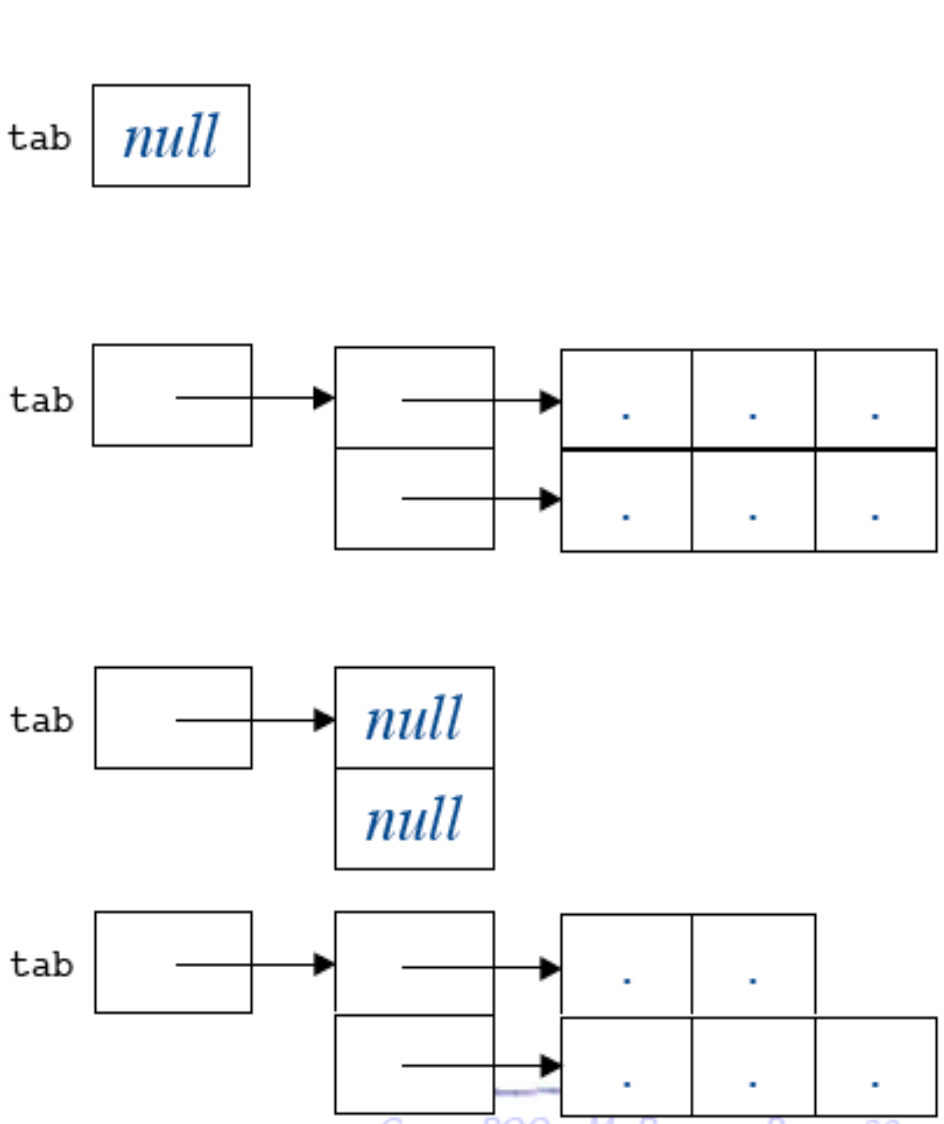
- Tableaux rectangulaires
  - Dimensionnement :

```
monTableau = new type[2][3]
```

- Tableaux non-rectangulaires
  - Dimensionnement :

```
monTableau = new type[2]
```

```
monTableau[0] = new type[2]
monTableau[1] = new type[3]
```



Petite précision du « System.out.println(…) »

- Usages : affichage à l'écran
  - « System.out.println(…) » : revient à la ligne
  - « System.out.print(…) » : ne revient pas à la ligne
- Différentes sorties possibles
  - « out » sortie standard
  - « err » sortie en cas d'erreur (non temporisée)
- Tout ce que l'on peut afficher…
  - Objets, nombres, booléens, caractères, …
- Tout ce que l 'on peut faire…
  - Concaténation sauvage entre types et objets avec le « + »

```
System.out.println("a=" + a + "donc a < 0 est " + a < 0);
```

Commentaires et mise en forme

- Documentation des codes sources :
  - Utilisation des commentaires

```
// Commentaire sur une ligne complète
int b = 34; // Commentaire après du code

/* Le début du commentaire
** Je peux continuer à écrire …
Jusqu'à ce que le compilateur trouve cela */
```

  - Utilisation de l'outil Javadoc (à voir dans la partie les Indispensables)

- Mise en forme
  - Facilite la relecture
  - Crédibilité assurée !!!!
  - Indentation à chaque niveau de bloc

```
if (b == 3) {
    if (cv == 5) {
        if (q) {
            ...
        } else {
            ...
        }
        ...
    }
    ...
}
```

Préférer

```
if (b == 3) {
if (cv == 5) {
if (q) {
...
} else {...}
...
}
...
}
```

Éviter

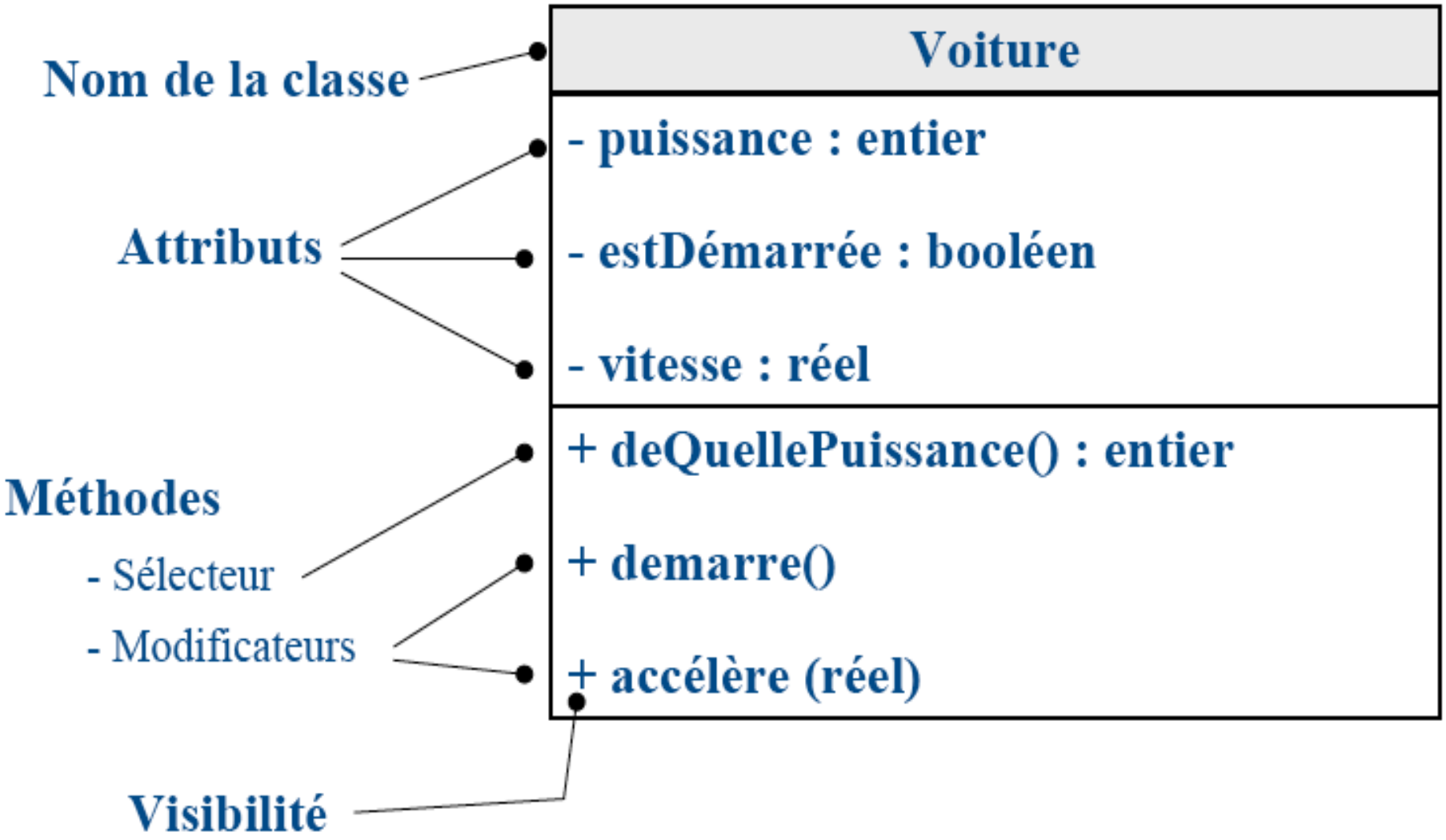
Programmation Orientée Objet application au langage Java

Classes et Objets

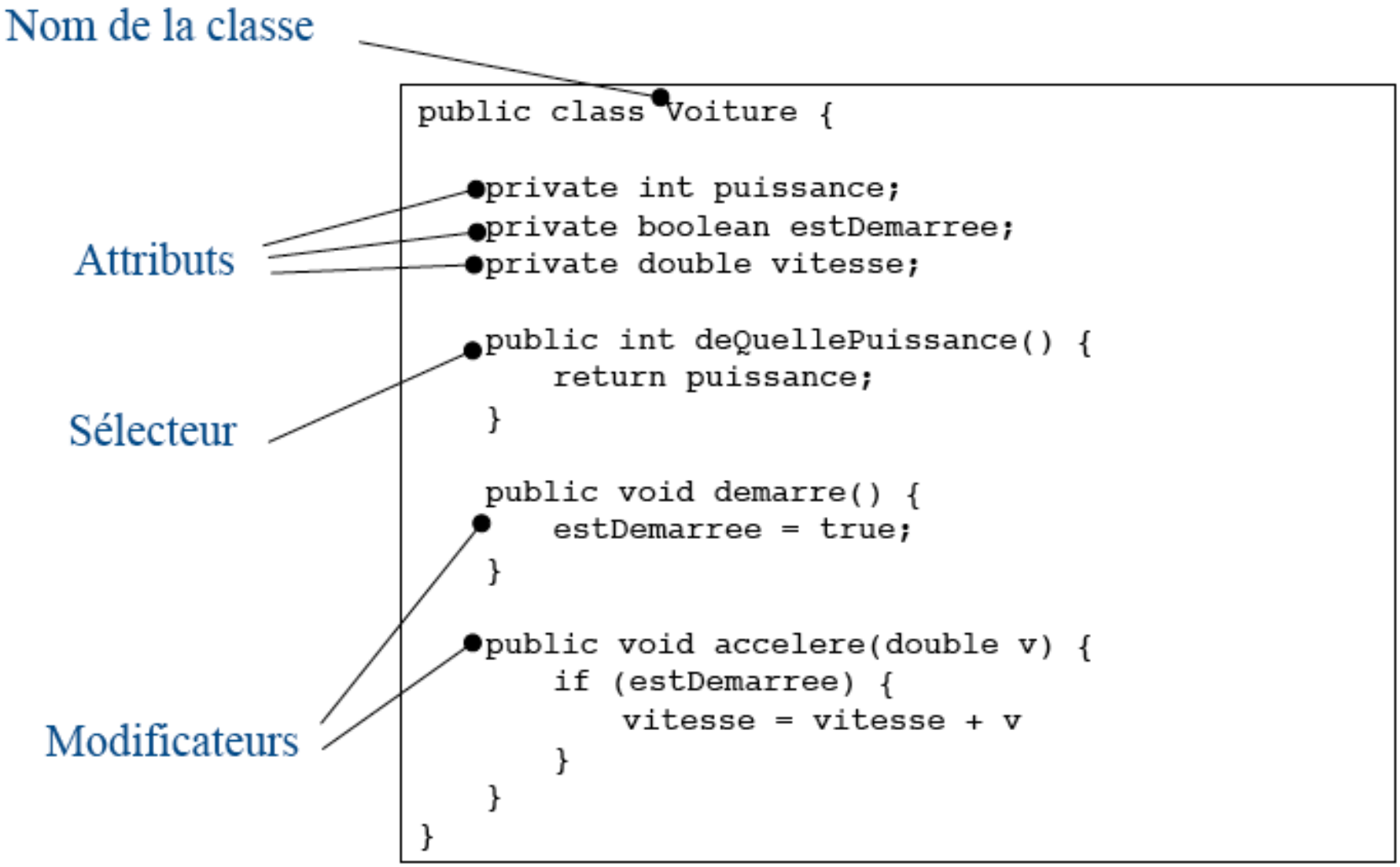
Classe et définition

- Une classe est constituée :
  - Données ce qu'on appelle des **attributs**
  - Procédures et/ou des fonctions ce qu'on appelle des **méthodes**
- Une classe est un modèle de définition pour des objets
  - Ayant même structure (même ensemble d'attributs)
  - Ayant même comportement (même méthodes)
  - Ayant une sémantique commune
- Les **objets** sont des représentations dynamiques (**instanciation**), du modèle défini pour eux au travers de la classe
  - Une classe permet d'**instancier** (créer) plusieurs objets
  - Chaque objet est instance d'une classe et une seule

Classe et notation UML



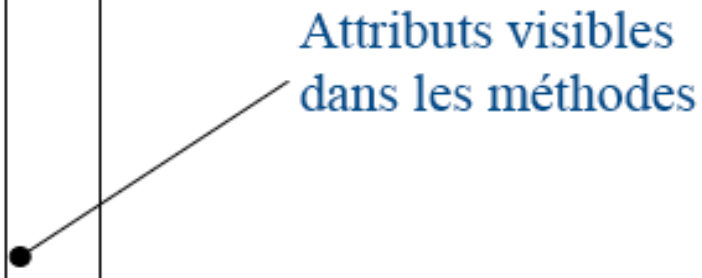
Codage de la classe « Voiture »



Classe et visibilité des attributs

- Caractéristique d'un attribut :
  - Variables « globales » de la classe
  - Accessibles dans toutes les méthodes de la classe

```
public class Voiture {

    private int puissance;
    private boolean estDemarree;
    private double vitesse;

    public int deQuellePuissance() {
        return puissance;
    }

    public void demarre() {
        estDemarree = true;
    }

    public void accelere(double v) {
        if (estDemarree) {
            vitesse = vitesse + v
        }
    }
}
```



Distinction entre attributs et variables

- Caractéristique d'une variable :
  - Visible à l'intérieur du bloc qui le définit

```
public class Voiture {

    private int puissance;
    private boolean estDemarree;
    private double vitesse;

    public int deQuellePuissance() {
        return puissance;
    }

    public void demarre() {
        estDemarree = true;
    }

    public void accelere(double v) {
        if (estDemarree) {
            double avecTolerance;
            avecTolerance = v + 25;
            vitesse = vitesse + avecTolerance
        }
    }
}
```

Variable visible uniquement dans cette méthode

Variable peut être définie n'importe où dans un bloc

Quelques conventions en Java : de la rigueur et de la classe ...

- Conventions de noms
  - CeciEstUneClasse
  - celaEstUneMethode(…)
  - jeSuisUneVariable
  - JE_SUIS_UNE_CONSTANTE

- Un fichier par classe, une classe par fichier
  - Classe « Voiture » décrite dans le fichier Voiture.java
  - Il peut exceptionnellement y avoir plusieurs classes par fichier (cas des *Inner classes*)

Respecter les minuscules et les majuscules des noms

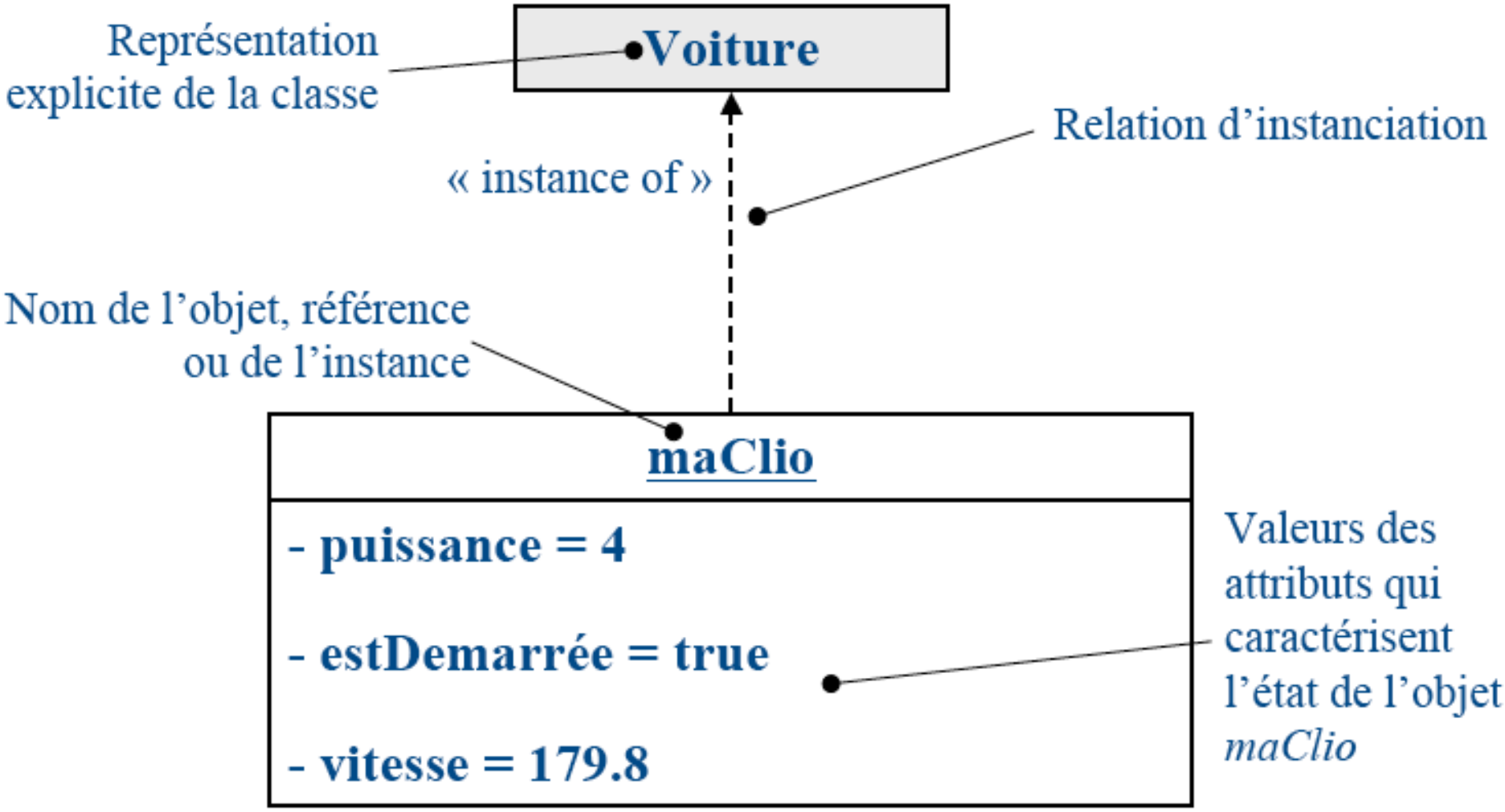
Objet et définition

- Un objet est **instance** d'une seule classe :
  - Se conforme à la description que celle-ci fournit
  - Admet une valeur propre à l'objet pour chaque attribut déclaré dans la classe
  - Les valeurs des attributs caractérisent l'**état** de l'objet
  - Possibilité de lui appliquer toute opération (**méthode**) définie dans la classe

- Tout objet est manipulé et identifié par sa référence
  - Utilisation de pointeur caché (plus accessible que le C++)
  - On parle indifféremment d'**instance**, de **référence** ou d'**objet**

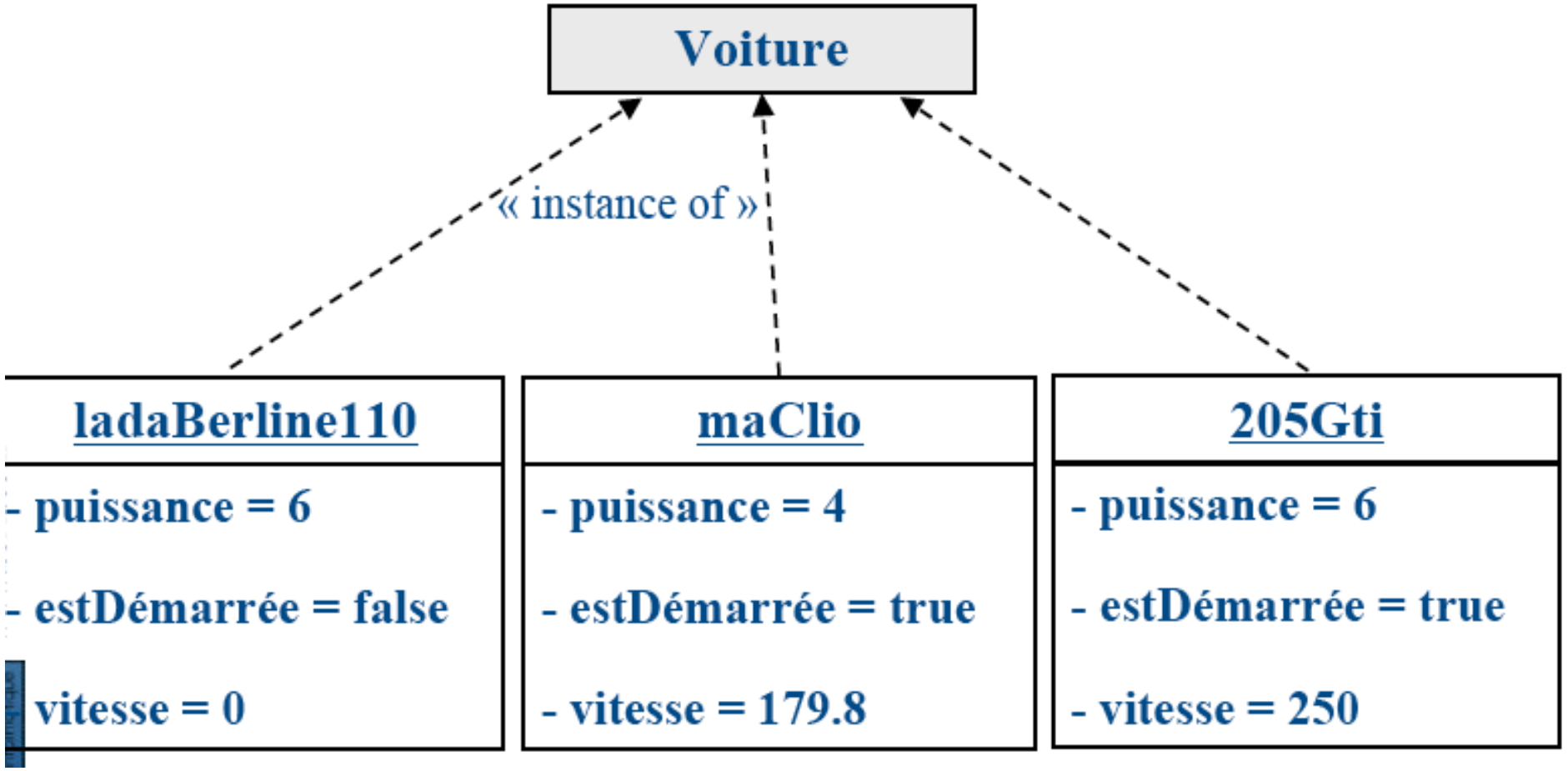
Objet et notation UML

- maClio est une instance de la classe *Voiture*



États des objets

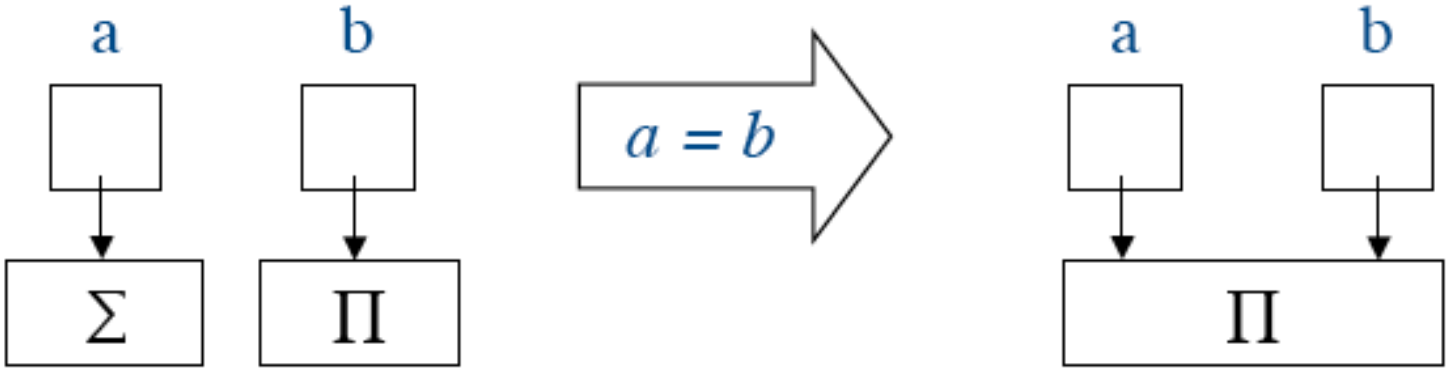
- Chaque objet qui est une instance de la classe *Voiture* possède ses propres valeurs d'attributs



Affectation et comparaison

- Affecter un objet
  - « a = b » signifie a devient identique à b
  - Les deux objets a et b sont identiques et toute modification de a entraîne celle de b

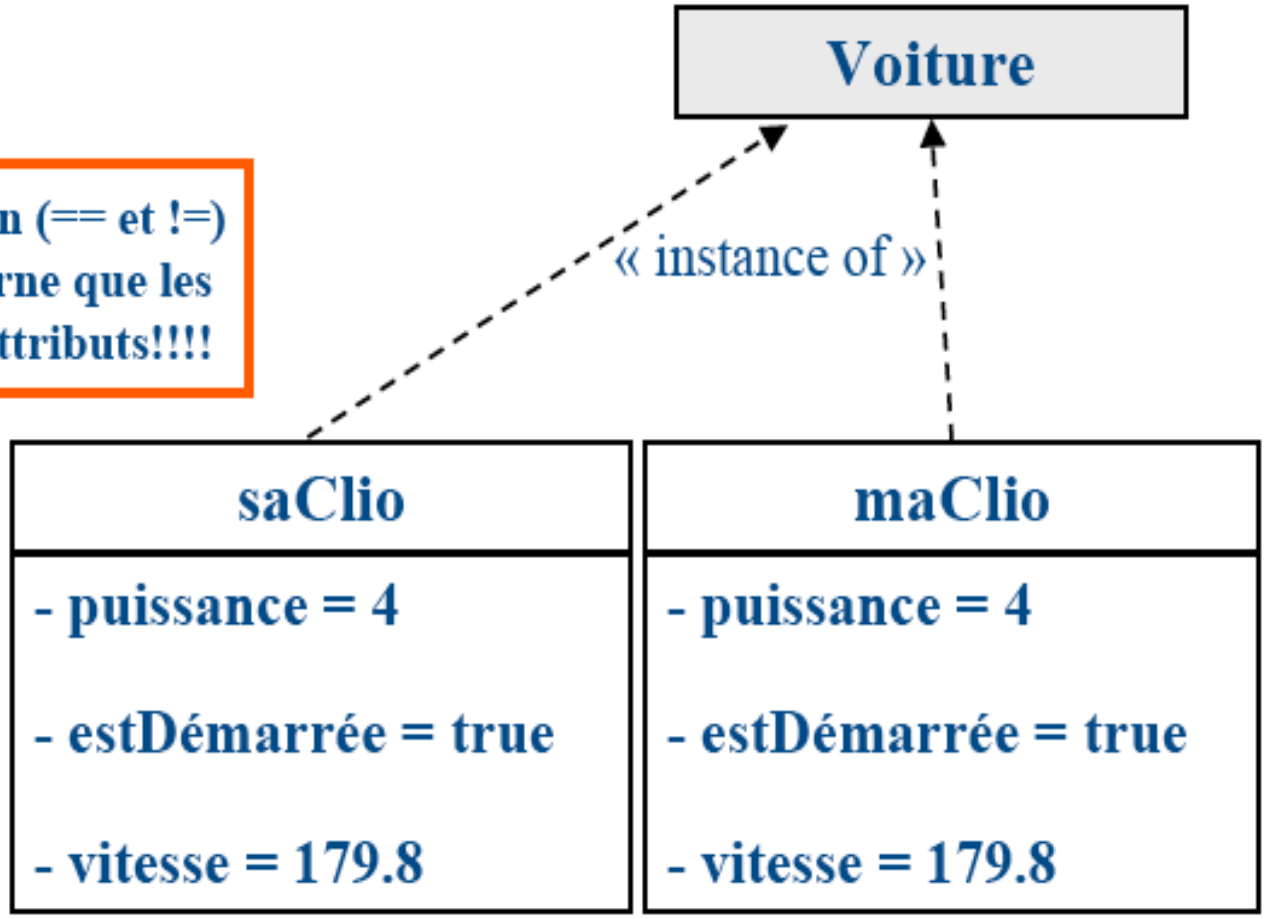
- Comparer deux objets
  - « a == b » retourne « true » si les deux objets sont identiques
  - C'est-à-dire si les références sont les mêmes, cela ne compare pas les attributs



Affectation et comparaison

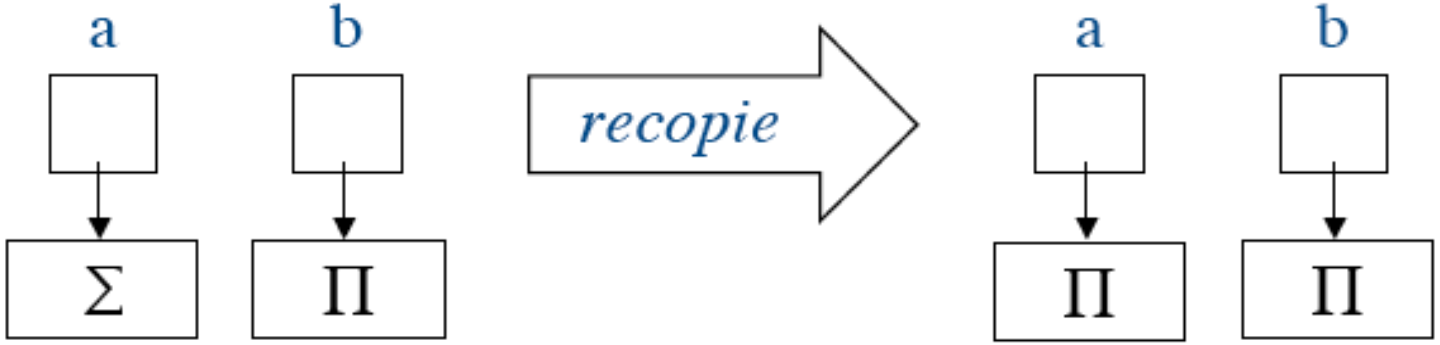
- L'objet **maClio** et **saClio** ont les mêmes attributs (états identiques) mais ont des références différentes
  - **maClio != saClio**

Le test de comparaison (== et !=) entre objets ne concerne que les référence et non les attributs!!!!



Affectation et comparaison

- Recopier les attributs d'un objet « clone() »
  - Les deux objets a et b sont distincts
  - Toute modification de a n'entraîne pas celle de b



- Comparer le contenu des objets : « equals(Object o) »
  - Renvoyer « true » si les objets a et b peuvent être considérés comme identique au vu de leurs attributs

Recopie et comparaison dans les parties suivantes

Structure des objets

- Un objet est constitué d'une partie « **statique** » et d'une partie « **dynamique** »

- Partie « **statique** »
  - Ne varie pas d'une instance de classe à une autre
  - Permet d'activer l'objet
  - Constituée des méthodes de la classe

- Partie « **dynamique** »
  - Varie d'une instance de classe à une autre
  - Varie durant la vie d'un objet
  - Constituée d'un exemplaire de chaque attribut de la classe

Cycle de vie d'un objet

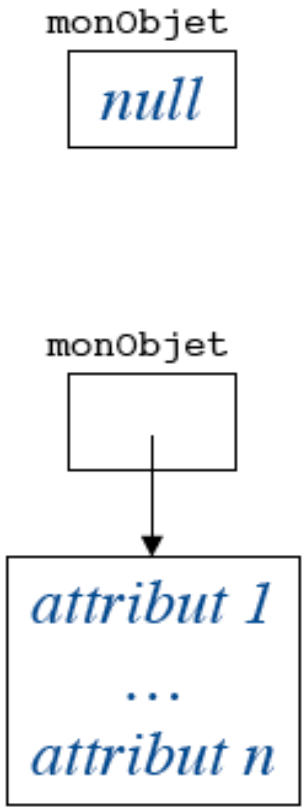
- Création
  - Usage d'un Constructeur
  - L'objet est créé en mémoire et les attributs de l'objet sont initialisés
- Utilisation
  - Usage des Méthodes et des Attributs (non recommandé)
  - Les attributs de l'objet peuvent être modifiés
  - Les attributs (ou leurs dérivés) peuvent être consultés

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

- Destruction et libération de la mémoire lorsque :
  - Usage (éventuel) d'un *Pseudo-Destructeur*
  - L'objet n'est plus référencé, la place mémoire qu'il occupait est récupérée

Création d'objets : déroulement

- La création d'un objet à partir d'une classe est appelée une **instanciation**.
- L'objet créé est une **instance** de la classe
- Déclaration
  - Définit le nom et le type de l 'objet
  - Un objet seulement déclaré vaut « **null** » (mot réservé du langage)
- Création et allocation de la mémoire
  - Appelle de méthodes particulières : les constructeurs
  - La création réservent la mémoire et initialise les attributs
- Renvoi d'une référence sur l'objet maintenant créé
  - monObjet != **null**



Création d'objets : réalisation

- La création d'un objet est réalisée par **new** Constructeur(paramètres)
  - Il existe un constructeur par défaut qui ne possèdent pas de paramètre (si aucun autre constructeur avec paramètre n'existe)

Les constructeurs portent le même nom que la classe

Déclaration

Création et allocation mémoire

```
public class TestMaVoiture {

    public static void main (String[] argv) {

        // Déclaration puis création
        Voiture maVoiture;
        maVoiture = new Voiture();

        // Déclaration et création en une seule ligne
        Voiture maSecondeVoiture = new Voiture();
    }
}
```

Création d'objets : réalisation

- Exemple : construction d'objets

```
public class TestMaVoiture {

    public static void main (String[] argv) {

        // Déclaration puis création
        Voiture maVoiture;
        maVoiture = new Voiture();

        // Déclaration d'une deuxième voiture
        Voiture maVoitureCopie;
        // Attention!! pour l'instant maVoitureCopie vaut null


        // Test sur les références.
        if (maVoitureCopie == null) {

            // Création par affectation d'une autre référence
            maVoitureCopie = maVoiture
            // maVoitureCopie possède la même référence que maVoiture
        }
        ...
    }}
```

Déclaration

Affectation par référence

Le constructeur de « Voiture »

- Actuellement
  - On a utilisé le constructeur par défaut sans paramètre
  - On ne sait pas comment se construit la « Voiture »
  - Les valeurs des attributs au départ sont indéfinies et identiques pour chaque objet (puissance, etc.)

Les constructeurs portent le même nom que la classe et n'ont pas de valeur de retour

- Rôle du constructeur en Java
  - Effectuer certaines initialisations nécessaire pour le nouvel objet créé

- Toute classe Java possède au moins un constructeur
  - Si une classe ne définit pas explicitement de constructeur, un constructeur par défaut sans arguments et qui n'effectue aucune initialisation particulière est invoquée

Le constructeur de « Voiture »

- Le constructeur de « Voiture »
  - Initialise « vitesse » à zéro
  - Initialise « estDémaree » à faux
  - Initialise la « puissance » à la valeur passée en paramètre du constructeur

Constructeur avec un paramètre

```
public class Voiture {

    private int puissance;

    private boolean estDemarree;

    private double vitesse;

    public Voiture(int p) {
        puissance = p;
        estDemaree = false;
        vitesse = 0;
    }
    ...
}
```

Construire une « Voiture » de 7 CV

- Création de la « Voiture » :
  - Déclaration de la variable « maVoiture »
  - Création de l'objet avec la valeur 7 en paramètre du constructeur

Déclaration

Création et allocation mémoire avec Voiture(int)

```
public class TestMaVoiture {

    public static void main(String[] argv)  {

        // Déclaration puis création
        Voiture maVoiture;

        maVoiture = new Voiture(7);

        Voiture maSecVoiture;
        // Sous entendu qu'il existe
        // explicitement un constructeur : Voiture(int)

        maSecVoiture = new Voiture(); // Erreur
    }
}
```

Constructeur sans arguments

- Utilité :
  - Lorsque l'on doit créer un objet sans pouvoir décider des valeurs de ses attributs au moment de la création
  - Il remplace le constructeur par défaut qui est devenu inutilisable dès qu'un constructeur (avec paramètres) a été défini dans la classe

```
public class Voiture {

    private int puissance;
    private boolean estDemarree;
    private double vitesse;

    public Voiture() {
        puissance = 4;
        estDemaree = false;
        vitesse = 0;
    }

    public Voiture(int p) {
        puissance = p;
        estDemaree = false;
        vitesse = 0;
    }...
}
```

```
public class TestMaVoiture {

    public static void main (String[] argv) {

        // Déclaration puis création
        Voiture maVoiture;
        maVoiture = new Voiture(7);
        Voiture maSecVoiture;
        maSecVoiture = new Voiture(); // OK
    }
}
```

Constructeurs multiples

- Intérêts
  - Possibilité d'initialiser un objet de plusieurs manières différentes
  - On parle alors de **surchage** (overloaded)
  - Le compilateur distingue les constructeurs en fonction :
    - de la position des arguments
    - du nombre
    - du type

Chaque constructeur possède le même nom (le nom de la classe)

```
public class Voiture {
    ...
    public Voiture() {
        puissance = 4; ...
    }

    public Voiture(int p) {
        puissance = p; ...
    }

    public Voiture(int p, boolean estDemaree) {
        ...
    }
}
```

Accès aux attributs

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

```
identificationObjet.nomAttribut
```

```
public class TestMaVoiture {

    public static void main (String[] argv) {

        // Déclaration puis création
        Voiture v1 = new Voiture();
        Voiture v2 = new Voiture();

        // Accès aux attributs en écriture
        v1.puissance = 110;

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

Il n'est pas recommandé d'accéder directement aux attributs d'un objet

Envoi de messages : appel de méthodes

- Pour « demander » à un objet d'effectuer un traitement il faut lui **envoyer un message**

- Un message est composé de trois parties
  - Une référence permettant de désigner l'objet à qui le message est envoyé
  - Le nom de la méthode ou de l'attribut à exécuter
  - Les éventuels paramètres de la méthode

```
identificationObjet.nomDeMethode( « Paramètres éventuels » )
```

- Envoi de message similaire à un appel de fonction
  - Le code défini dans la méthode est exécuté
  - Le contrôle est retourné au programme appelant

Envoi de messages : appel de méthodes

Ne pas oublier les parenthèses pour les appels aux méthodes

Voiture
- …
+ deQuellePuissance() : entier + demarre() + accélère (réel) + …

```
public class TestMaVoiture {

    public static void main (String[] argv) {

        // Déclaration puis création
        Voiture maVoiture = new Voiture();

        // La voiture démarre
        maVoiture.demarre();

        if (maVoiture.deQuellePuissance() == 4) {
            System.out.println("Pas très Rapide…");
        }

        // La voiture accélère
        maVoiture.accélère(123.5);
    }
```

Envoi d'un message à l'objet *maVoiture*

Appel d'un modificateur

Envoi d'un message à l'objet *maVoiture*

Appel d'un sélecteur

Envoi de messages : passage de paramètres

- Un paramètre d'une méthode peut être
  - Une variable de type simple
  - Une référence d'un objet typée par n'importe quelle classe
- En Java tout est passé par valeur
  - Les paramètres effectifs d'une méthode
  - La valeur de retour d'une méthode (si différente de « void »)
- Les types simples
  - Leur valeur est recopiée
  - Leur modification dans la méthode n'entraîne pas celle de l'original
- Les objets
  - Leur référence est recopiée et non pas les attributs
  - Leur modification dans la méthode entraîne celle de l'original!!!

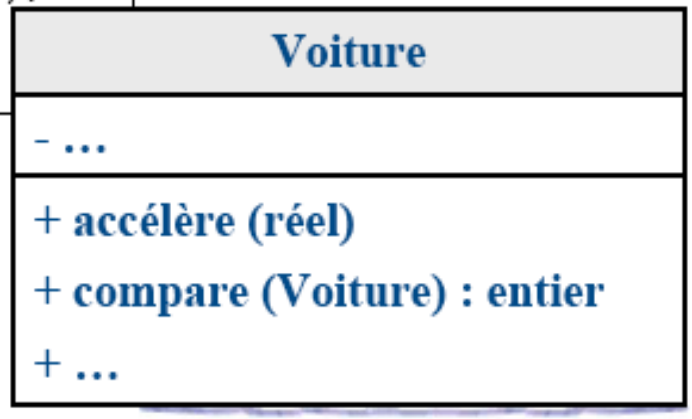
Envoi de messages : passage de paramètres

- Exemple

```
public class TestMaVoiture {

    public static void main (String[] argv) {

        // Déclaration puis création 1
        Voiture maVoiture = new Voiture();

        // Déclaration puis création 2
        Voiture maSecondeVoiture = new Voiture();

        // Appel de la méthode compare(voiture) qui
        // retourne le nombre de différence
        int p = maVoiture.compare(maSecondeVoiture);

        System.out.println("Nbre différence :" + p);
    }
}
```

Référence comme paramètre

Appel d'un sélecteur avec passage d'une référence



L'objet « courant »

- L'objet « courant » est désigné par le mot clé **this**
  - Permet de désigner l'objet dans lequel on se trouve
  - **self** ou **current** dans d'autres langages
  - Désigne une référence particulière qui est un membre caché

Ne pas tenter d'affecter une nouvelle valeur à this !!!!

```
this = ... ; // Ne pas y penser
```

- Utilité de l'objet « courant »
  - Rendre explicite l'accès aux propres attributs et méthodes définies dans la classe
  - Passer en paramètre d'une méthode la référence de l'objet courant

L'objet « courant » : attributs et méthodes

- Désigne des variables ou des méthodes définies dans une classe

```
public class Voiture {

    ...
    private boolean estDemarree;
    private double vitesse;

    public int deQuellePuissance() {
        ...
    }

    public void accelere(double vitesse) {
        if (estDemarree) {
            this.vitesse = this.vitesse + vitesse;
        }
    }
}
```

Désigne la variable *vitesse*

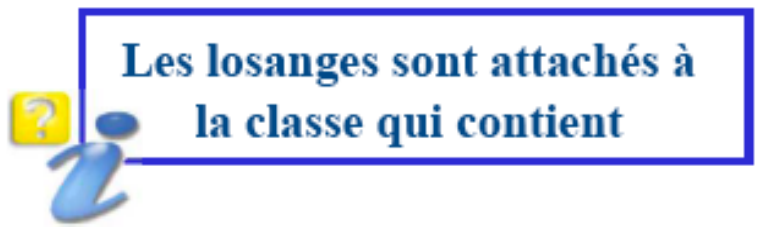
Désigne l'attribut *vitesse*

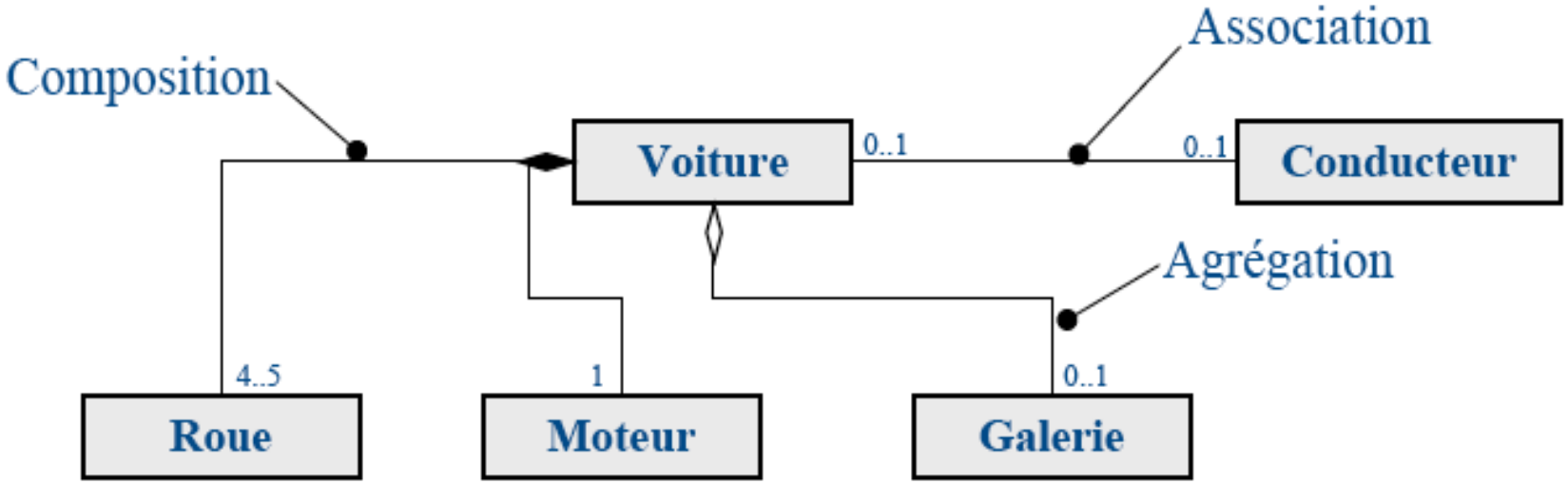
Désigne l'attribut *demarree*

this n'est pas nécessaire lorsque les identificateurs de variables ne présentent aucun équivoque

Le retour d'UML…

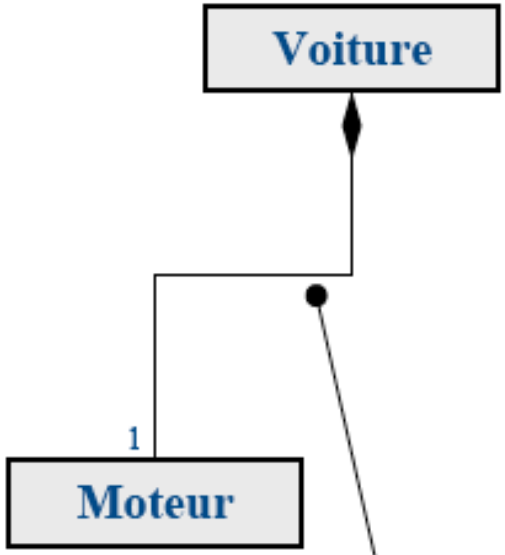
- Association : les objets sont sémantiquement liés —
  - Exemple : une Voiture est conduite par un Conducteur
- Agrégation : cycle de vie indépendant ◊
  - Exemple : une Voiture et une Galerie
- Composition : cycle de vie identiques ◆
  - Exemple : une Voiture possède un Moteur qui dure la vie de la Voiture





Codage de l'association : composition

- L'objet de classe *Voiture* peut envoyer des messages à l'objet de classe Moteur : Solution 1



A discuter : si le moteur d'une voiture est « mort », il y a la possibilité de le changer

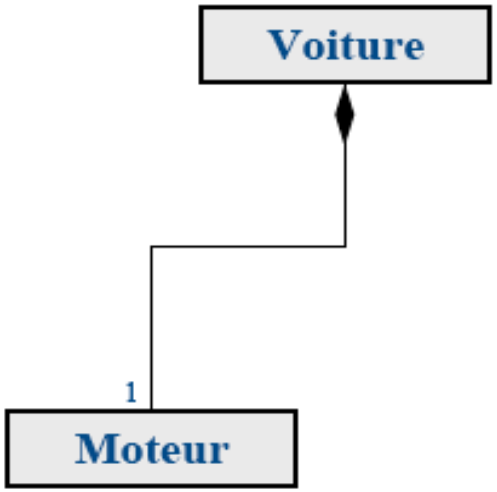
Attribut qui stocke la référence du moteur

```
public class Voiture {

    private Moteur leMoteur;
    ...

    public Voiture(int p) {
        leMoteur = new Moteur(p);
        ...
    }
    ...
}
```

Création de l'objet Moteur

Codage de l'association : composition

- L'objet de classe *Moteur* n'envoie pas de message à l'objet de classe Voiture : Solution 1



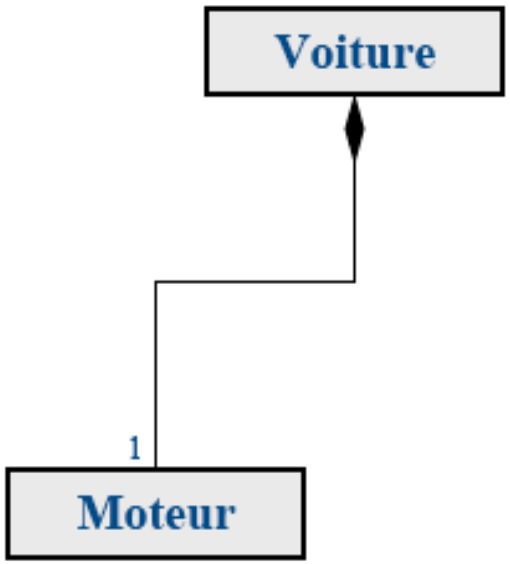
Attribut qui stocke la puissance

```
public class Moteur {

    private int puissance;
    ...

    public Moteur(int p) {
        puissance = p;
        ...
    }
    ...
}
```

La puissance est donnée lors de la construction

Codage de l'association : composition

- Il peut être nécessaire que les deux objets en composition s'échangent des messages : Solution 2
  - L'objet *Voiture* « voit » l'objet *Moteur*



Attribut qui stocke la référence du Moteur

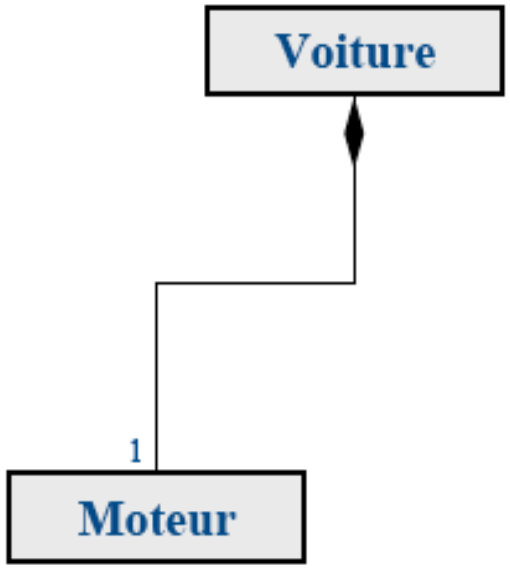
```
public class Voiture {

    private Moteur leMoteur;
    ...

    public Voiture(int p) {
        leMoteur = new Moteur(p,this);
        ...
    }
    ...
}
```

Création de l'objet Moteur

Transmission de la référence de l'objet courant

Codage de l'association : composition

- Il peut être nécessaire que les deux objets en composition s'échangent des messages : Solution 2
  - L'objet *Moteur* « voit » l'objet *Voiture*



Attribut qui stocke la puissance

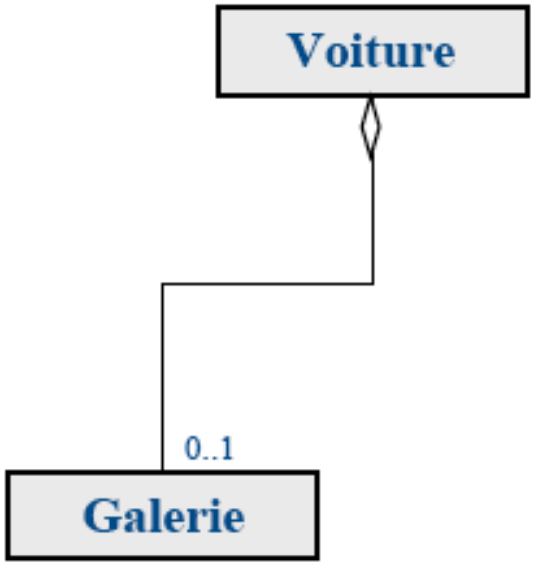
```
public class Moteur {

    private int puissance;
    private Voiture laVoiture;
    ...

    public Moteur(int p, Voiture v) {
        puissance = p;
        laVoiture = v;
        ...
    }
    ...
}
```

La puissance est donnée lors de la construction

Référence d'un objet Voiture en paramètre

Codage de l'association : agrégation

- L'objet de classe *Galerie* n'envoie pas de message à l'objet de classe *Voiture*



Attribut qui stocke une référence de Galerie

```
public class Voiture {

    private Galerie laGalerie;
    ...

    public Voiture(Galerie g) {
        laGalerie = g;
        ...
    }
    ...
}
```

Un objet Galerie est transmis au moment de la construction de Voiture

Destruction et ramasse-miettes

- La destruction des objets se fait de manière implicite
- Le ramasse-miettes ou Garbage Collector se met en route
  - Automatiquement :
    - Si plus aucune variable ne référence l'objet
    - Si le bloc dans lequel il était défini se termine
    - Si l'objet a été affecté à « null »
  - Manuellement :
    - Sur demande explicite du programmeur par l'instruction « System.gc() »
- Un pseudo-destructeur « *protected void finalize()* » peut être défini explicitement par le programmeur
  - Il est appelé juste avant la libération de la mémoire par la machine virtuelle, mais on ne sait pas quand. Conclusion : pas très sûr!!!!

Préférer définir une méthode et de l'invoquer avant que tout objet ne soit plus référencé : *detruit()*

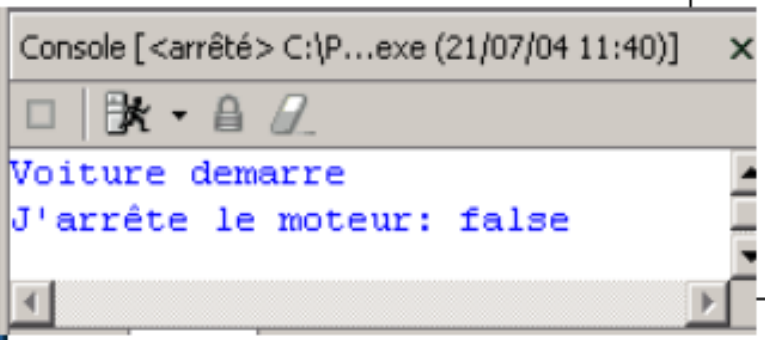
Destruction et ramasse-miettes

```
public class Voiture {

    private boolean estDemarree;
    ...

    protected void finalize() {
        estDemarree = false;
        System.out.println("Moteur arrêté");
    }
    ...
}
```

Pour être sûr que finalize s'exécute il faut absolument appeler explicitement *System.gc()*

```
public class TestMaVoiture {

    public static void main (String[] argv) {
        // Déclaration puis création de maVoiture
        Voiture maVoiture = new Voiture();
        maVoiture.demarre();
        // maVoiture ne sert plus à rien
        maVoiture = null;

        // Appel explicite du garbage collector
        System.gc();

        // Fin du programme
        System.exit(0);
        System.out.println("Message non visible");
    }
```

Force le programme à se terminer



Gestion des objets

- Afficher son type et son emplacement mémoire : *System.out.println()*

```
System.out.println(maVoiture) // Voiture@119c082
```

- Récupérer son type : méthode « *Class getClass()* »

```
maVoiture.getClass(); // Retourne un objet de type Class
Class classVoiture = maVoiture.getClass(); // Class est une classe!!!
```

- Tester son type : opérateur « *instanceof* » ou mot clé « *class* »

```
if (maVoiture instanceof Voiture) {...} // C'est vrai
```

ou

```
if (maVoiture.getClass() == Voiture.class) {...} // C 'est vrai également
```

Surcharge

- La surcharge (*overloading*) n'est pas limitée aux constructeurs, elle est possible également pour n'importe quelle méthode

- Possibilité de définir des méthodes possédant le même nom mais dont les arguments diffèrent

- Quand une méthode surchargée est invoquée le compilateur sélectionne automatiquement la méthode dont le nombre est le type des arguments correspondent au nombre et au type des paramètres passés dans l'appel de la méthode

Des méthodes surchargées peuvent avoir des types de retour différents à condition qu'elles aient des arguments différents

Surcharge

Exemple

```
public class Voiture {
    private double vitesse;
    ...

    public void accelere(double vitesse){
        if (estDemarree) {
            this.vitesse = this.vitesse + vitesse;
        }
    }
    public void acelere(int vitesse) {
        if (estDemaree) {
            this.vitesse = this.vitesse +
                (double)vitesse;
        }
    }
...}
```

```
public class TestMaVoiture {

    public static void main (String[] argv) {
        // Déclaration puis création de maVoiture
        Voiture maVoiture = new Voiture();

        // Accélération 1 avec un double
        maVoiture.accelere(123.5);
        // Accélération 2 avec un entier
        maVoiture.accelere(124);
}}
```

Constructeurs multiples : le retour

- Appel explicite d'un constructeur de la classe à l'intérieur d'un autre constructeur
  - Doit se faire comme première instruction du constructeur
  - Utilise le mot-clé « this(paramètres effectifs) »

- Exemple
  - Implantation du constructeur sans paramètre de « Voiture » à partir du constructeur avec paramètres

```
public class Voiture {
    ...

    public Voiture() {
        this(7, new Galerie());
    }

    public Voiture(int p) {
        this(p, new Galerie());
    }

    public Voiture(int p, Galerie g) {
        puissance = p;
        moteur = new Moteur(puissance);
        galerie = g;
        ...
    }
...
```

Encapsulation

- Possibilité d'accéder aux attributs d'une classe Java mais pas recommandé car contraire au principe d'encapsulation
  - Les données (attributs) doivent être protégés et accessibles pour l'extérieur par des sélecteurs
- Possibilité d 'agir sur la visibilité des membres (attributs et méthodes) d'une classe vis à vis des autres classes
- Plusieurs niveaux de visibilité peuvent être définis en précédant d'un modificateur la déclaration d'un *attribut*, *méthode* ou *constructeur*
  - private
  - public
  - protected

A revoir dans la partie suivante

Encapsulation : visibilité des membres d'une classe

	+ public	- private
classe	La classe peut être utilisée par n'importe quelle classe	Utilisable uniquement par les classes définies à l'intérieur d'une autre classe. Une classe privée n'est utilisable que par sa classe englobante
attribut	Attribut accessible partout où sa classe est accessible. N'est pas recommandé du point de vue encapsulation	Attribut restreint à la classe où est faite la déclaration
méthode	Méthode accessible partout où sa classe est accessible.	Méthode accessible à l'intérieur de la définition de la classe

Encapsulation

Exemple

```
public class Voiture {

    private int puissance;
    ...

    public void demarre() {
        ...
    }

    private void makeCombustion() {
        ...
    }
}
```

Une méthode privée ne peut plus être invoquée en dehors du code de la classe où elle est définie

```
public class TestMaVoiture {

    public static void main (String[] argv) {
        // Déclaration puis création de maVoiture
        Voiture maVoiture = new Voiture();

        // Démarrage de maVoiture
        maVoiture.demarre();

        maVoiture.makeCombustion(); // Erreur
    }
}
```

Les chaînes de caractères « String »

- Ce sont des objets traités comme des types simples...
- Initialisation

```
String maChaine = "Bonjour!"; // Cela ressemble à un type simple
```

- Longueur

```
maChaine.length(); // Avec les parenthèses car c'est une méthode
```

- Comparaison

```
maChaine.equals("Bonjour!"); // Renvoi vrai
```

- Concaténation

```
String essai = "ess" + "ai";
String essai = "ess".concact("ai");
```

Faites attention à la comparaison de chaînes de caractères.

```
maChaine == "toto";
```

Comparaison sur les références !!

Les Chaînes modifiables « StringBuffer »

- Elles sont modifiables par insertion, ajouts, conversions, etc

- On obtient une « StringBuffer » avec ses constructeurs

```
StringBuffer mCM = new StringBuffer(int length);
StringBuffer mCM = new StringBuffer(String str);
```

- On peut les transformer en chaînes normales « String » par :

```
String s = mCM.toString();
```

- On peut leur ajouter n 'importe (surcharge) quoi par :

```
mCM.append(...); // String, int, long, float, double
```

- On peut leur insérer n 'importe (surcharge) quoi par :

```
mCM.insert(int offset, ...); // String, int, long, float, double
```

Les chaînes décomposables « StringTokenizer »

- Elles permettent la décomposition en mots ou éléments suivant un délimiteur

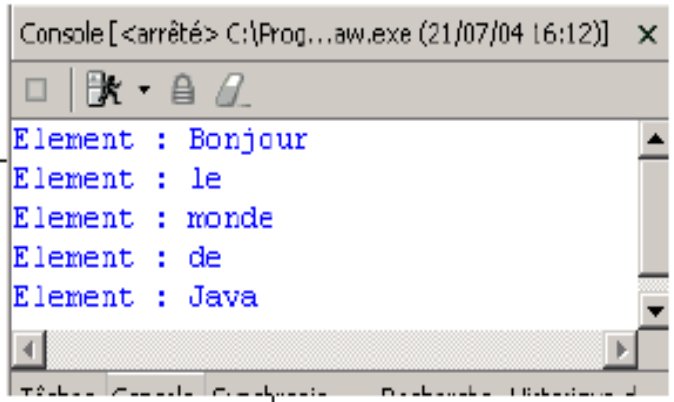
```
this is a test => this
                  is
                  a
                  test
```

- On obtient une « StringTokenizer » avec ses constructeurs

```
StringTokenizer mCM = new StringTokenize(String str); // Délimiteur = blanc
StringTokenizer rMCM = new StringTokenizer(String str, String delim);
```

- Un exemple :

```
StringTokenizer st =
    new StringTokenizer("Bonjour,
        le monde|de|Java",", |");

while(st.hasMoreElements())
    System.out.println("Element : " + st.nextElement());
```



Variables de classe

- Il peut être utile de définir pour une classe des attributs indépendamment des instances : nombre de Voitures créées

- Utilisation des Variables de classe comparables aux « variables globales »

- Usage des **variables de classe**
  - Variables dont il n'existe qu'un seul exemplaire associé à sa classe de définition
  - Variables existent indépendamment du nombre d'instances de la classe qui ont été créés
  - Variables utilisables même si aucune instance de la classe n'existe

Variables de classe

- Elles sont définies comme les attributs mais avec le mot-clé **static**

```
public static int nbVoitureCreees;
```

Attention à l'encapsulation. Il est dangereux de laisser cette variable de classe en public.

- Pour y accéder, il faut utiliser non pas un identificateur mais le nom de la classe

```
Voiture.nbVoitureCreees = 3;
```

Il n'est pas interdit d'utiliser une variable de classe comme un attribut (au moyen d 'un identificateur) mais fortement déconseillé

Constantes de classe

- Usage
  - Ce sont des constantes liées à une classe
  - Elles sont écrites en MAJUSCULES

Une constante de classe est généralement toujours visible

- Elles sont définies (en plus) avec le mot-clé final

```
public class Galerie {
    public static final int MASSE_MAX = 150;
}
```

- Pour y accéder, il faut utiliser non pas un identificateur d 'objet mais le nom de la classe (idem variables de classe)

```
if (maVoiture.getWeightLimite() <= Galerie.MASSE_MAX) {...}
```

Variables et Constantes de classe : exemple

- Exemple

```
public class Voiture {

    public static final int PTAC_MAX = 3500;
    private int poids;
    public static int nbVoitureCreees;
    ...

    public Voiture(int poids, ...) {
        this.poids = poids;
        ...
    }
}
```

Dangereux car possibilité de modification extérieure...

```
public class TestMaVoiture {

    public static void main (String[] argv) {
        // Déclaration puis création de maVoiture
        Voiture maVoiture = new Voiture(2500);
        ...

        System.out.println("Poids maxi:" +
            Voiture.PTAC_MAX);
        System.out.println(Voiture.nbVoitureCreees);
        ...
    }
}
```

Utilisation de Variables et Constantes de classe par le nom de la classe Voiture

Méthodes de classe

- Usage
  - Ce sont des méthodes qui ne s'intéressent pas à un objet particulier
  - Utiles pour des calculs intermédiaires internes à une classe
  - Utiles également pour retourner la valeur d'une variable de classe en visibilité *private*

- Elles sont définies comme les méthodes d'instances, mais avec le mot clé **static**

```
public static double vitesseMaxToleree() {
    return vitesseMaxAutorisee*1.10;
}
```

- Pour y accéder, il faut utiliser non pas un identificateur d'objet mais le nom de la classe (idem variables de classe)

```
Voiture.vitesseMaxToleree()
```

Méthodes de classe

- Exemple

```
public class Voiture {

    private static int nbVoitureCreees;
    ...

    public static int getNbVoitureCreees(){
        return Voiture.nbVoitureCreees;
    }
}
```

Déclaration d'une variable de classe privée. Respect des principes d'encapsulation.

Déclaration d'une méthode de classe pour accéder à la valeur de la variable de classe.

```
public class TestMaVoiture {

    public static void main (String[] argv) {
        // Déclaration puis création de maVoiture
        Voiture maVoiture = new Voiture(2500);
        ...

        System.out.println("Nbre Instance :" +
                    Voiture.getNbVoitureCreees());
    }
}
```

Méthodes de classe : erreur classique

- Exemple

```
public class Voiture {

    private Galerie laGalerie;
    ...

    public Voiture(Galerie g) {
        laGalerie = g;
        ...
    }

    public static boolean isGalerieInstall() {
        return (laGalerie != null)
    }
}
```

Déclaration d'un objet Galerie non statique

Erreur : Utilisation d'un attribut non statique dans une zone statique

On ne peut pas utiliser de variables d'instance dans une méthode de classe!!!!

Méthodes de classe

- Rappel : chacun des types simples (int, double, boolean, char) possède un alter-ego objet disposant de méthodes de conversion
- Par exemple la classe ***Integer*** « encapsule » le type **int**
  - Constructeur à partir d'un int ou d'une chaîne de caractères

```
public Integer(int value);
public Integer(String s);
```

  - Disponibilité de méthodes qui permettent la conversion en type simple

```
Integer valueObjet = new Integer(123);
int valuePrimitif = valueObjet.intValue();
    Ou
int valuePrimitif = valueObjet; (AutoBoxing)
```

  - Des méthodes de classe très utiles qui permettent à partir d'une chaîne de caractères de transformer en type simple ou type object

```
String maValueChaine = new String("12313");
int maValuePrimitif = Integer.parseInt(maValueChaine);
```

Attention aux erreurs de conversion. Retour d'une exception. Voir dans la dernière partie du cours

Les tableaux en Java : application Objets

① Déclaration

```
Voiture[] monTableau;
```

② Dimensionnement

```
monTableau = new Voiture[3];
```

③ Initialisation

```
monTableau[0] = new Voiture(5);
monTableau[1] = new Voiture(7);
monTableau[2] = new Voiture(8);
```

Ou ①② et ③

```
Voiture[] monTab = {
    new Voiture(5),
    new Voiture(7),
    new Voiture(8)
};
```

```
for (int i = 0; i < monTableau.length; i++) {
    System.out.println(monTableau[i].demarre());
}
```

Une colle

- La méthode **`main()`** est nécessairement **`static`**. Pourquoi ?

Une colle

- La méthode **`main()`** est nécessairement **`static`**. Pourquoi ?

- $\Rightarrow$ La méthode **`main()`** est exécutée au début du programme. Aucune instance n'est donc déjà créée lorsque la méthode **`main()`** commence son exécution. Ça ne peut donc pas être une méthode d'instance.

Blocs d'initialisation `static`

- Les blocs **`static`** permettent d'initialiser les variables **`static`** trop complexes à initialiser dans leur déclaration :

```
class UneClasse {
  static int[] unTableau = new int[25];
  static {
    for (int i = 0; i<25; i++) {
      . . . // initialisation de unTableau
    }
  } // fin du bloc static
  . . .
```

- Ils sont exécutés une seule fois, quand la classe est chargée en mémoire

Programmation Orientée Objet application au langage Java

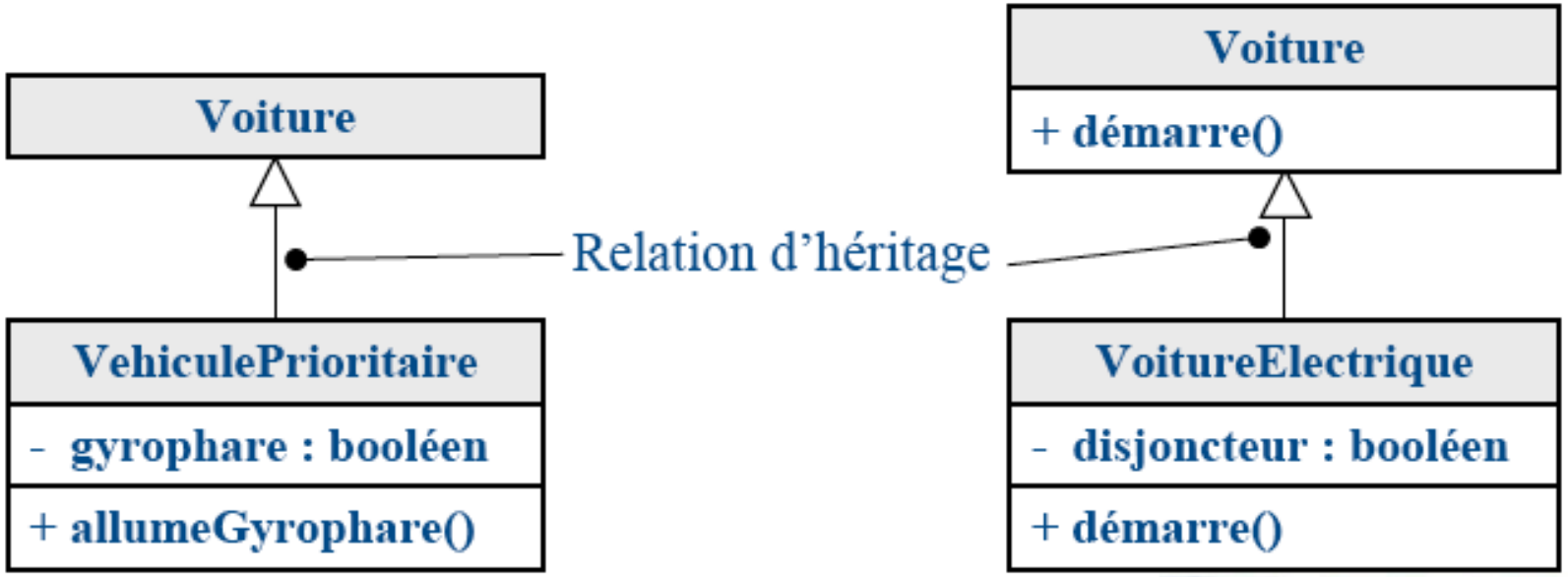
Héritage

Définition et intérêts

- Héritage
  - Technique offerte par les langages de programmation pour construire une classe à partir d'une (ou plusieurs) autre classe en partageant ses attributs et opérations
- Intérêts
  - **Spécialisation, enrichissement** : une nouvelle classe réutilise les attributs et les opérations d 'une classe en y ajoutant et/ou des opérations particulières à la nouvelle classe
  - **Redéfinition** : une nouvelle classe redéfinit les attributs et opérations d'une classe de manière à en changer le sens et/ou le comportement pour le cas particulier défini par la nouvelle classe
  - **Réutilisation** : évite de réécrire du code existant et parfais on ne possède pas les sources de la classe à hériter

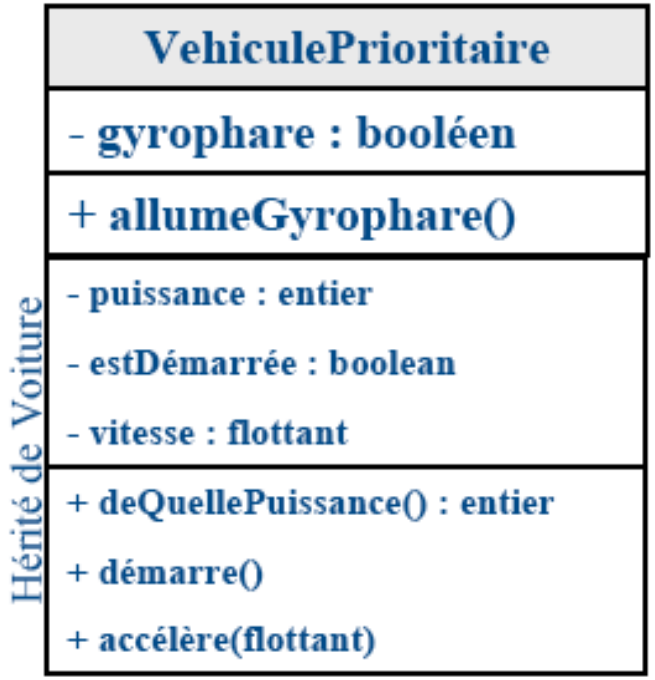
Spécialisation de la classe « Voiture »

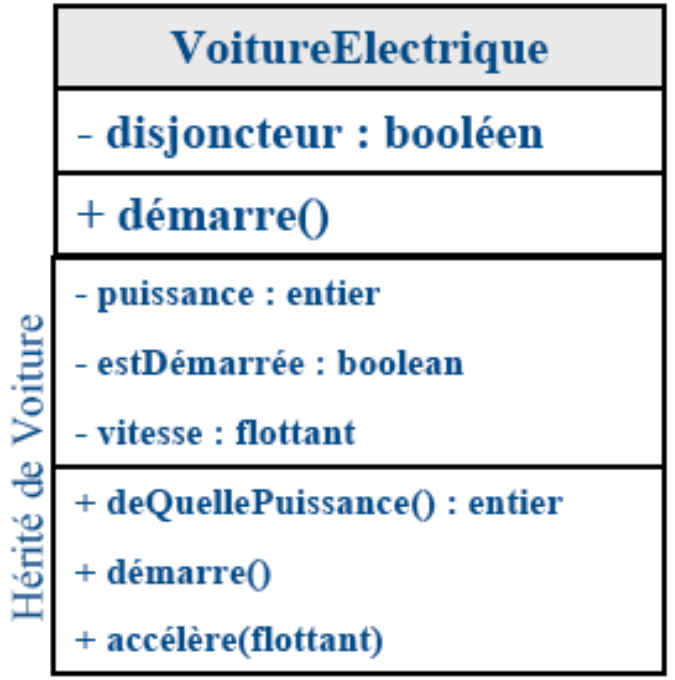
- Un véhicule prioritaire est une voiture avec un gyrophare
  - Un véhicule prioritaire répond aux mêmes messages que la Voiture
  - On peut allumer le gyrophare d'un véhicule prioritaire
- Une voiture électrique est une voiture dont l'opération de démarrage est différente
  - Une voiture électrique répond aux même messages que la Voiture
  - On démarre une voiture électrique en activant un disjoncteur



Classes et sous-classes

- Un objet de la classe *VehiculePrioritaire* ou *VoitureElectrique* est aussi un objet de la classe *Voiture* donc il dispose de tous les attrib et opérations de la classe *Voiture*



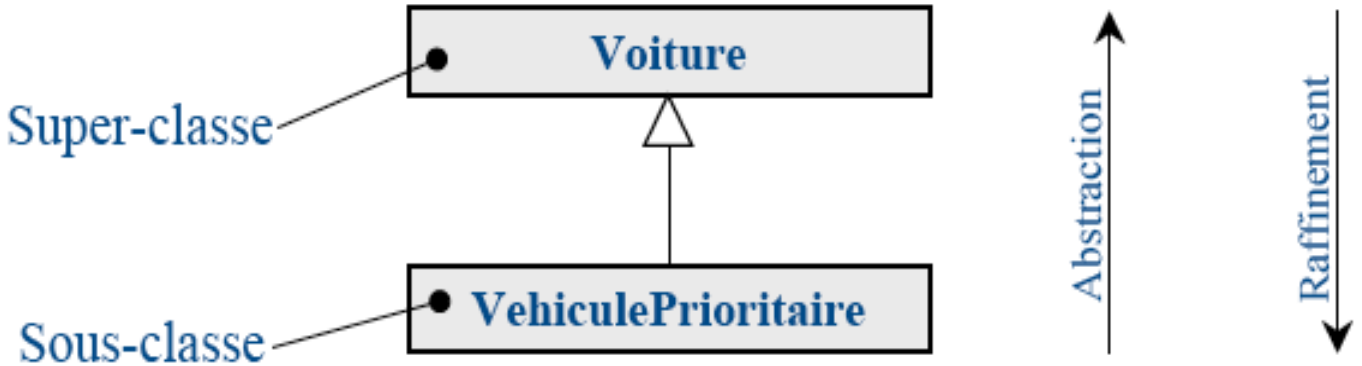


Classes et sous-classes : terminologie

- Définitions
  - La classe *VehiculePrioritaire* **hérite** de la classe *Voiture*
  - *Voiture* est la **classe mère** et *VehiculePrioritaire* la **classe fille**
  - *Voiture* est la **super-classe** de la classe *VehiculePrioritaire*
  - *VehiculePrioritaire* est une **sous-classe** de *Voiture*
- Attention 
  - Un objet de la classe *VehiculePrioritaire* ou *VoitureElectrique* est forcément un objet de la classe *Voiture*
  - Un objet de la classe *Voiture* n'est pas forcément un objet de la classe *VehiculePrioritaire* ou *VoitureElectrique*

Généralisation et Spécialisation

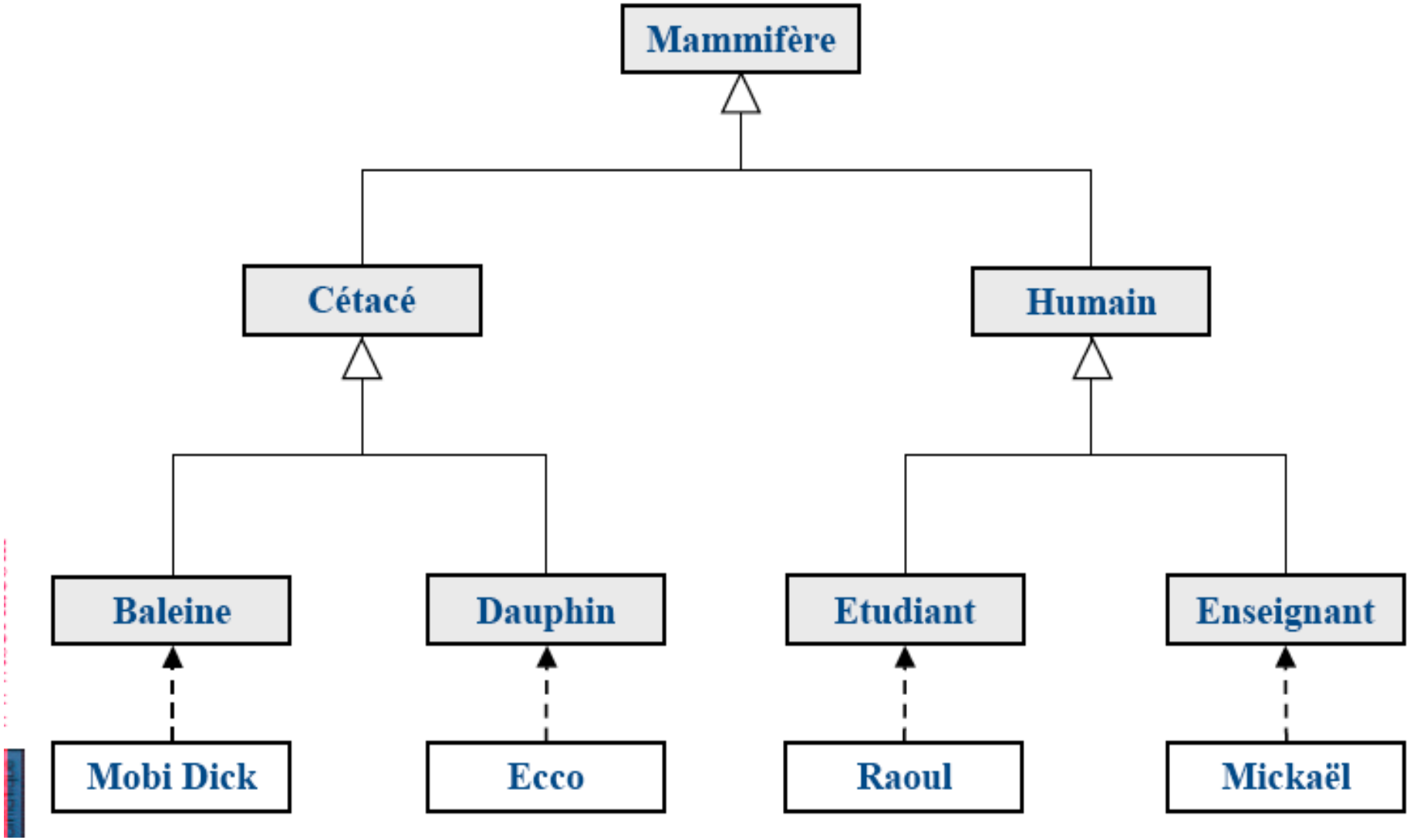
- La généralisation exprime une relation « **est-un** » entre une classe et sa super-classe



- L'héritage permet
  - de **généraliser** dans le sens abstraction
  - de **spécialiser** dans le sens raffinement

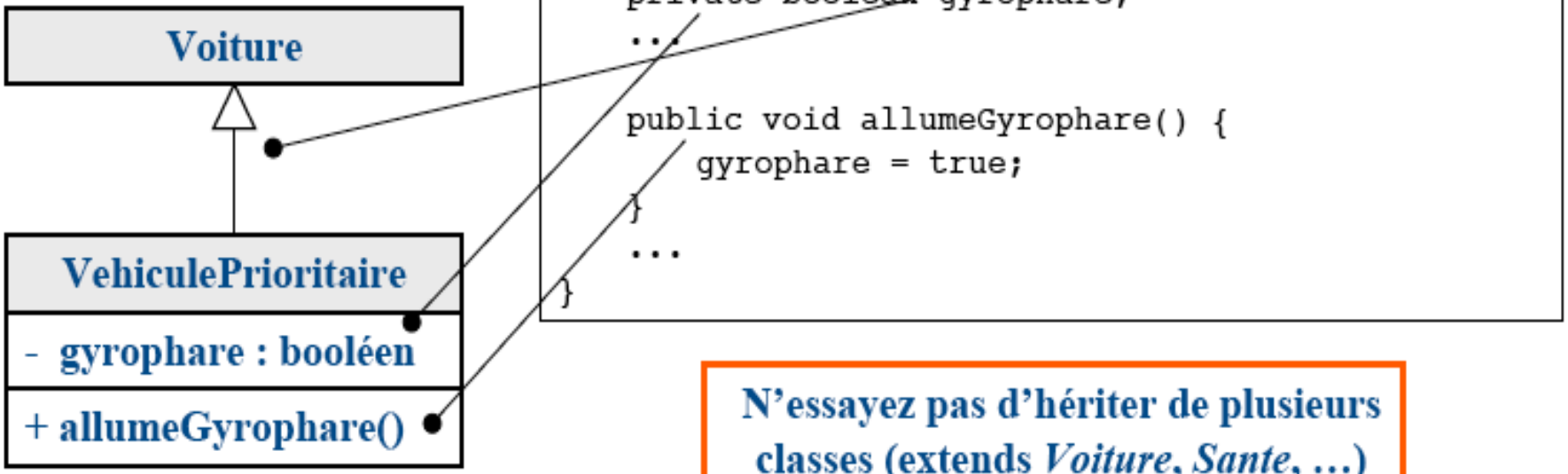
Exemple d'héritage

- Exemple



Héritage et Java

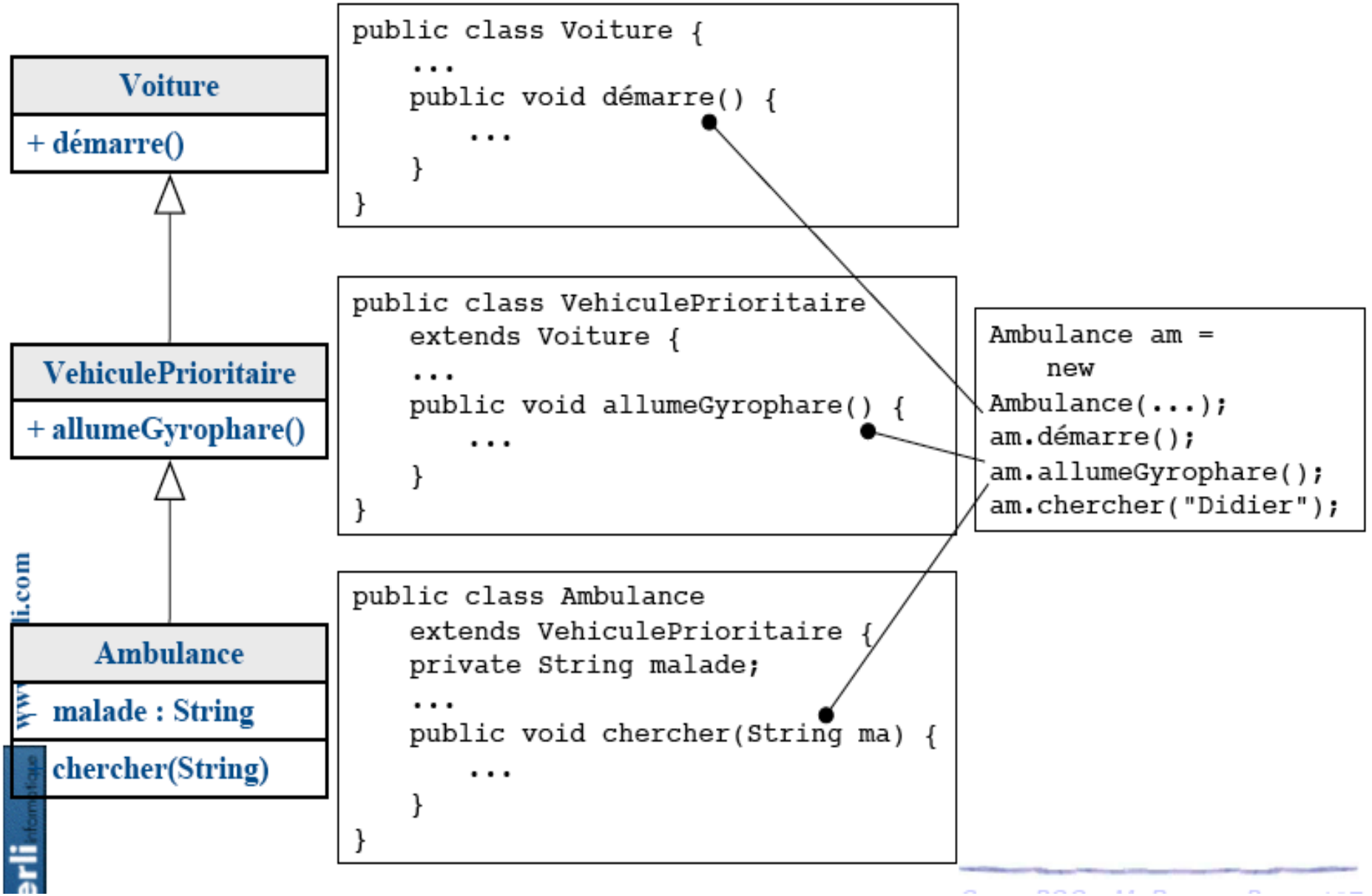
- Héritage simple
  - Une classe ne peut hériter que d'une seule autre classe
  - Dans certains autres langages (ex : C++) possibilité d'héritage multiple
  - Utilisation du mot-clé **extends** après le nom de la classe




```
public class VehiculePrioritaire extends Voiture {

    private boolean gyrophare;
    ...

    public void allumeGyrophare() {
        gyrophare = true;
    }
    ...
}
```

N'essayez pas d'hériter de plusieurs classes (extends *Voiture*, *Sante*, …) ça ne fonctionne pas

Héritage à plusieurs niveaux



Surcharge et redéfinition

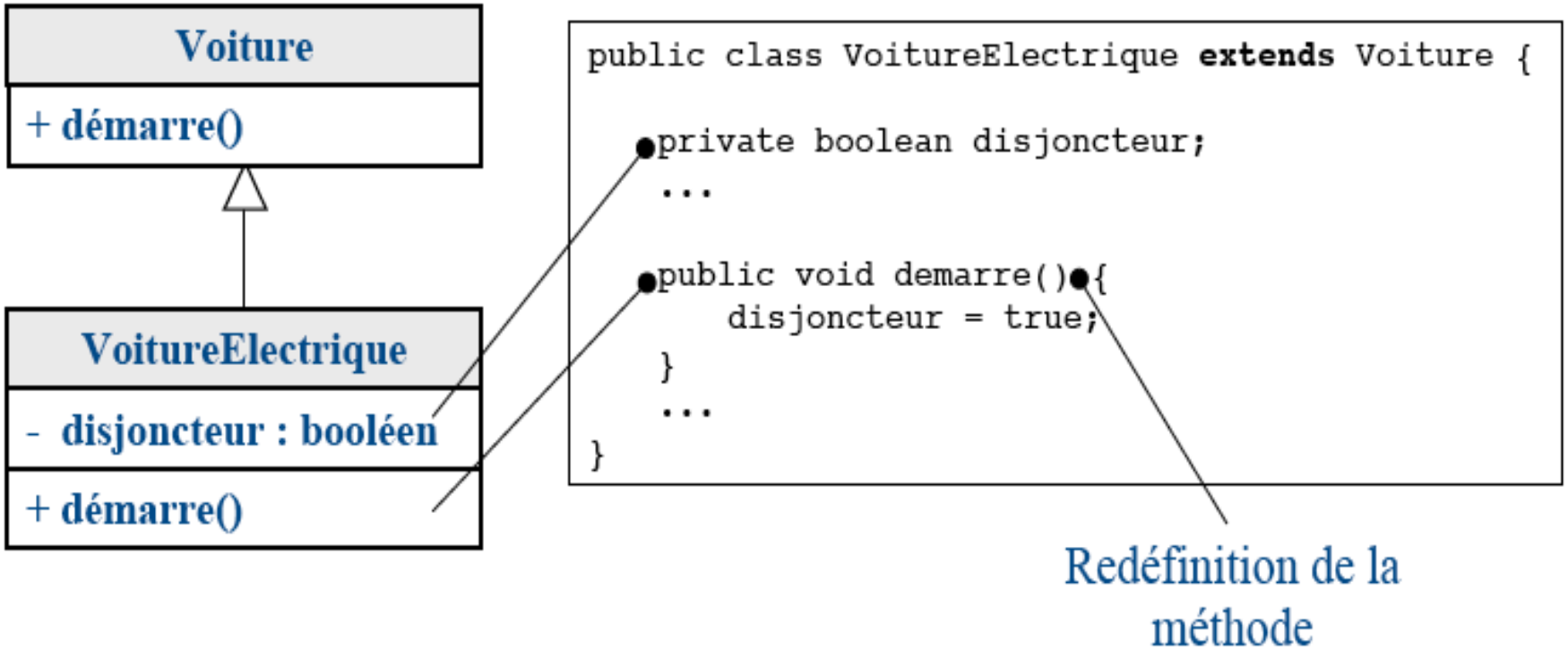
- L'héritage
  - Une sous-classe peut ajouter des nouveaux attributs et/ou méthodes à ceux qu'elle hérite (surcharge en fait partie)
  - Une sous-classe peut redéfinir (redéfinition) les méthodes dont elle hérite et fournir des implémentations spécifiques pour celles-ci
- Rappel de la *surcharge* : possibilité de définir des méthodes possédant le même nom mais dont les arguments (paramètres et valeur de retour) diffèrent

Des méthodes surchargées peuvent avoir des types de retour différents à condition qu'elles aient des arguments différents

- *Redéfinition* (overriding) : lorsque la sous-classe définit une méthode dont le nom, les paramètres et le type de retour sont identiques

Surcharge et redéfinition

- Une voiture électrique est une voiture dont l'opération de démarrage est différente
  - Une voiture électrique répond aux même messages que la *Voiture*
  - On démarre une voiture électrique en activant un disjoncteur



Surcharge et redéfinition

```
public class Voiture {
    ...
    public void demarre() {
        ...
    }
}
```

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

Redéfinition

Surcharge

```
public class VoitureElectrique
    extends Voiture {
    ...
    public void demarre() {
        ...
    }
}
```

VoitureElectrique possède « au plus » une méthode de moins que *VehiculePrioritaire*

```
public class VehiculePrioritaire
    extends Voiture {
    ...
    public void demarre(int code) {
        ...
    }
}
```

VehiculePrioritaire possède « au plus » une méthode de plus que *VoitureElectrique*

Redéfinition avec réutilisation

- Intérêt
  - La redéfinition d'une méthode « écrase » le code de la méthode héritée
  - Possibilité de réutiliser le code de la méthode hérité par le mot-clé **super**
  - **super** permet ainsi la désignation explicite de l 'instance d 'une classe dont le type est celui de la classe mère
  - Accès aux attributs et méthodes redéfinies par la classe courante mais que l'on désire utiliser

```
super.nomSuperClasseMethodeAppelee(...);
```

- Exemple de la Voiture : les limites à résoudre
  - L'appel à la méthode *demarre* de *VoitureElectrique* ne modifie que l'attribut *disjoncteur*
  - ...

Redéfinition avec réutilisation

Exemple

```
public class Voiture {

    private boolean estDemarree;
    ...

    public void demarre() {
        estDemarree = true;
    }
}
```

Mise à jour de l'attribut estDemarree

```
public class VoitureElectrique extends Voiture {

    private boolean disjoncteur;
    ...

    public void demarre() {
        disjoncteur = true;
        super.demarre();
    }
    ...
}
```

La position de super n'a ici aucune importance

```
public class TestMaVoiture {

    public static void main (String[] argv) {
        // Déclaration puis création
        VehiculeElectrique laRochelle =
            new VehiculeElectrique(...);
        laRochelle.demarre();
    }
}
```

Envoi d'un message par appel de *demarre*

Usage des constructeurs : suite

- Possibilité comme les méthodes de réutiliser le code des constructeurs de la super-classe

- Appel explicite d'un constructeur de la classe mère à l'intérieur d'un constructeur de la classe fille
  - Utilise le mot-clé **super**

L'appel au constructeur de la super-classe doit se faire absolument en première instruction

```
super(paramètres du constructeur);
```

- Appel implicite d'un constructeur de la classe mère est effectué quand il n'existe pas d'appel explicite. Java insère implicitement l'appel **super()**

Usage des constructeurs : suite

Exemple

```
public class Voiture {
    ...

    public Voiture() {
        this(7, new Galerie());
    }

    public Voiture(int p) {
        this(p, new Galerie());
    }

    public Voiture(int p, Galerie g) {
        puissance = p;
        moteur = new Moteur(puissance);
        galerie = g;
        ...
    }
    ...
}
```

L'appel au constructeur de la super-classe doit se faire absolument un première instruction

Implantation du constructeur de *VoiturePrioritaire* à partir de *Voiture*

```
public class VoiturePrioritaire
    extends Voiture {

    private boolean gyrophare;

    public VoiturePrioritaire(int p, Galerie g) {
        super(p, null);
        this.gyrophare = false;
}}
```

Usage des constructeurs : suite

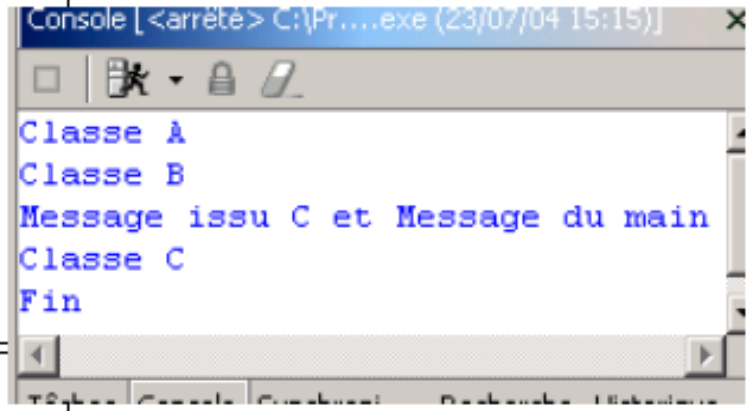
Exemple : chaînage des constructeurs

```
public class A {
    public A() {
        System.out.println("Classe A");
    }
}
```

```
public class B extends A {
    private String message;
    public B(String message) {
        super(); // Appel implicite
        System.out.println("Classe B");
        System.out.println(message);
    }
}
```

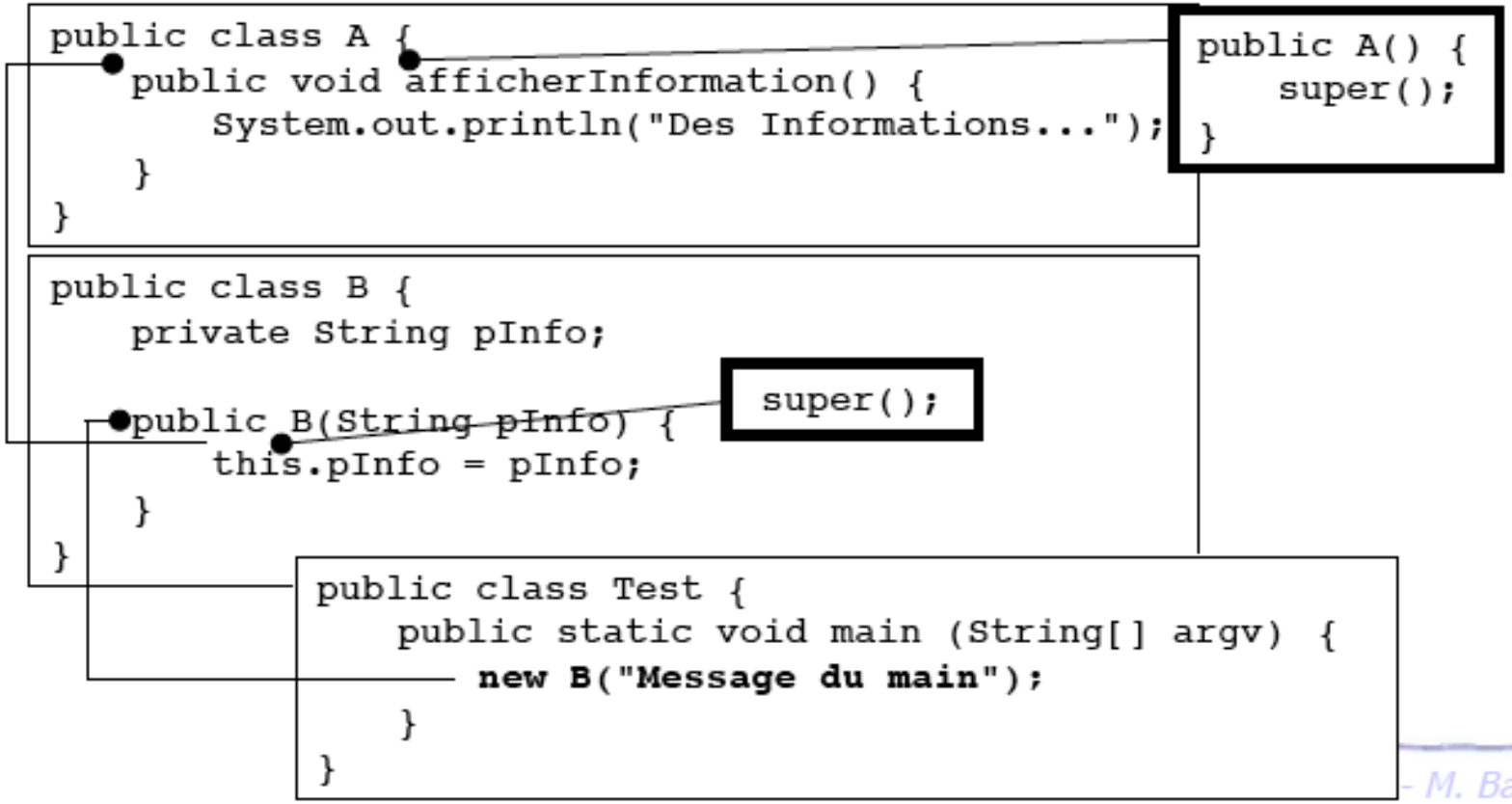
```
public class C extends B {
    public C(String debut) {
        super("Message issu C" + debut);
        System.out.println("Classe C");
        System.out.println("Fin");
    }
}
```

```
public class Test {
    public static void main (String[] argv) {
        new C("Message du main");
    }
}
```



Usage des constructeurs : suite

- Rappel : si une classe ne définit pas explicitement de constructeur, elle possède alors un constructeur par défaut
  - Sans paramètre
  - Qui ne fait rien
  - Inutile si un autre constructeur est défini explicitement



Usage des constructeurs : suite

Exemple

```
public class Voiture {
    ...
    public Voiture(int p) {
        this(p, new Galerie());
    }

    public Voiture(int p, Galerie g) {
        puissance = p;
        moteur = new Moteur(puissance);
        galerie = g;
        ...
    }
    ...
}
```

Constructeurs explicites désactivation du constructeur par défaut

Erreur : il n'existe pas dans *Voiture* de constructeur sans paramètre

```
public class VoiturePrioritaire
    extends Voiture {

    private boolean gyrophare;

    public VoiturePrioritaire(int p, Galerie g) {
        super();
        this.gyrophare = false;
    }
}
```

La classe Object : le mystère résolu

- La classe **Object** est la classe de plus haut niveau dans la hiérarchie d'héritage
  - Toute classe autre que **Object** possède une super-classe
  - Toute classe hérite directement ou indirectement de la classe **Object**
  - Une classe qui ne définit pas de clause **extends** hérite de la classe **Object**

```
public class Voiture extends Object {
    ...

    public Voiture(int p, Galerie g) {
        puissance = p;
        moteur = new Moteur(puissance);
        galerie = g;
        ...
    }
    ...
}
```

Object
+ Class getClass()
+ String toString()
+ boolean equals(Object)
+ int hashCode()
...

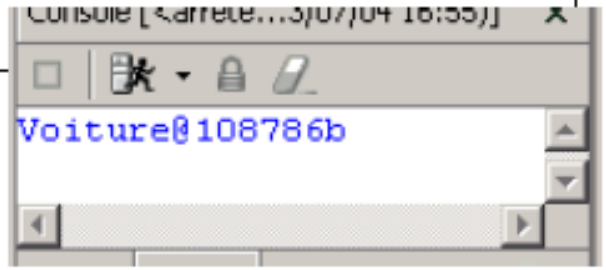
Il n'est pas nécessaire d'écrire explicitement extends Object

La classe Object : le mystère résolu

```
public class Voiture {
    ...
    public Voiture(int p) {
        this(p, new Galerie());
    }
}
```

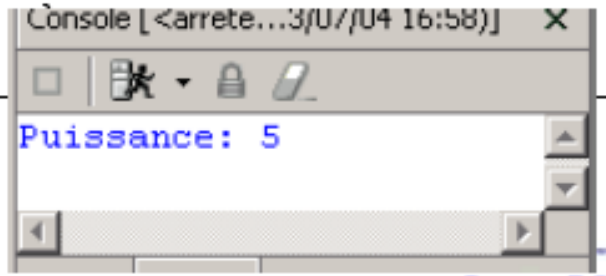
```
public class Test {
    public static void main (String[] argv) {
        Voiture maVoiture = new Voiture(3);
        System.out.println(maVoiture);
    }
```

```
public Sring toString() {
    return (this.getClass().getName() +
        "@" + this.hashCode());
}
```




```
public class Voiture {
    ...
    public Voiture(int p) {
        this(p, new Galerie());
    }

    public String toString() {
        return("Puissance:" + p);
    }
}
```

```
.ln(maVoiture.toString());
public class Test {
    public static void main (String[] argv) {
        Voiture maVoiture = new Voiture(3);
        System.out.println(maVoiture);
    }
```

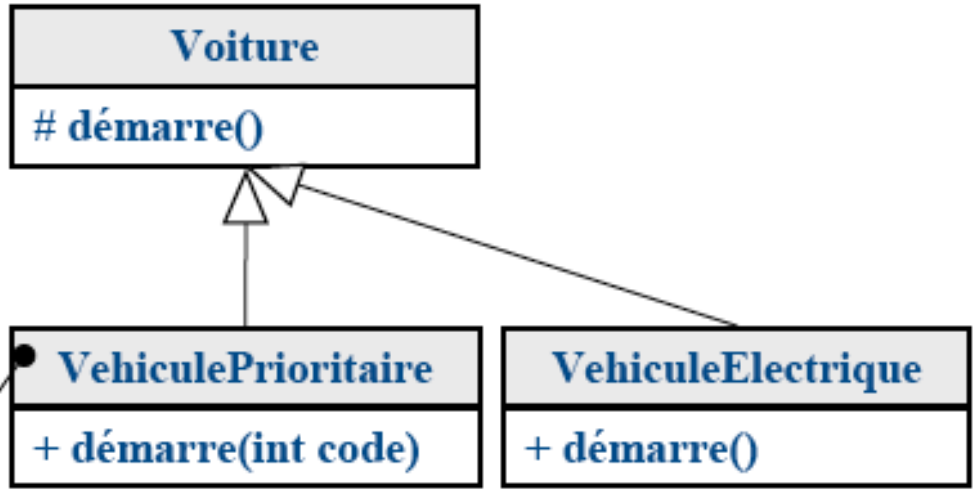


Redéfinition de la méthode
String toString()

Droits d'accès aux attributs et méthodes

- Exemple de la Voiture : les limites à résoudre
  - La méthode *démarre()* est disponible dans la classe *VehiculePrioritaire*
    C'est-à-dire que l'on peut démarrer sans donner le code !!!
  - Solution : protéger la méthode démarre() de la classe Voiture
- Réalisation
  - Utilisation du mot-clé **protected** devant la définition des méthodes et/ou attributs
  - Les membres sont accessibles dans la classe où il est défini, dans toutes ses sous-classes

démarre() n'est pas accessible « publiquement » dans un objet *VehiculePrioritaire*



Droits d'accès aux attributs et méthodes

Exemple

```
public class Voiture {

    private boolean estDemarree;
    ...

    protected void demarre() {
        estDemarree = true;
    }
}
```

```
public class VoiturePrioritaire
    extends Voiture {

    private int codeVoiture;

    public void demarre(int code) {
        if (codeVoiture == code) {
            super.demarre();
        };
    }
}
```

```
public class TestMaVoiture {

    public static void main (String[] argv) {
        // Déclaration puis création de maVoiture
        VehiculeElectrique laRochelle = new VehiculeElectrique(...);
        larochelle.demarre(); // Appel le demarre de VehiculeElectrique

        VehiculePrioritaire pompier = new VehiculePrioritaire(...);
        pompier.demarre(1234); // Appel le demarre VoiturePrioritaire
        pompier.demarre(); // Erreur puisque demarre n'est pas public
    }}
```

Méthodes et classes finales

- Définition
  - Utilisation du mot-clé **final**
  - Méthode : interdire une éventuelle redéfinition d'une méthode

```
public final void demarre();
```

  - Classe : interdire toute spécialisation ou héritage de la classe concernée

```
public final class VoitureElectrique extends Voiture {
    ...
}
```

La classe *String* par exemple est finale