



Java: Programmation Orientée Objet

Chapitre 2 : Classes et Objets

Novembre 11, 2025

Nouakchott

Classes et Objets

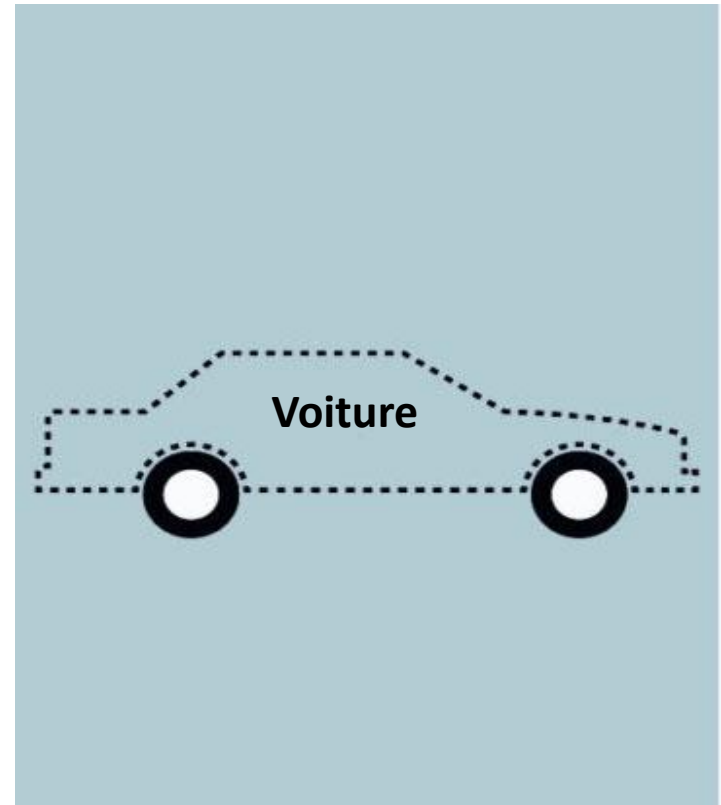
Classe

Une **classe** est une description abstraite d'un ensemble d'objets partageant les **mêmes attributs** (données) et les **mêmes méthodes** (comportements).

Exemple :

```
class Voiture {  
    String marque;  
    int vitesse;  
  
    void accelerer(int v){  
        vitesse += v;  
    }  
  
    // Getter pour marque  
    public String getMarque() {  
        return marque;  
    }  
  
    // Setter pour marque  
    public void setMarque(String marque) {  
        this.marque = marque;  
    }  
}
```

```
// Getter pour vitesse  
public int getVitesse() {  
    return vitesse;  
}  
  
// Setter pour vitesse  
public void setVitesse(int vitesse) {  
    this.vitesse = vitesse;  
}  
  
// Méthode non accesseur  
public void freiner() {  
    vitesse = 0; // la voiture s'arrête  
                // complètement  
}  
}
```



Classes et Objets

Classe

La classe se compose des éléments suivants :

- ❑ Les attributs **marque** et **vitesse** sont **privés**
cela protège les données (**encapsulation**).

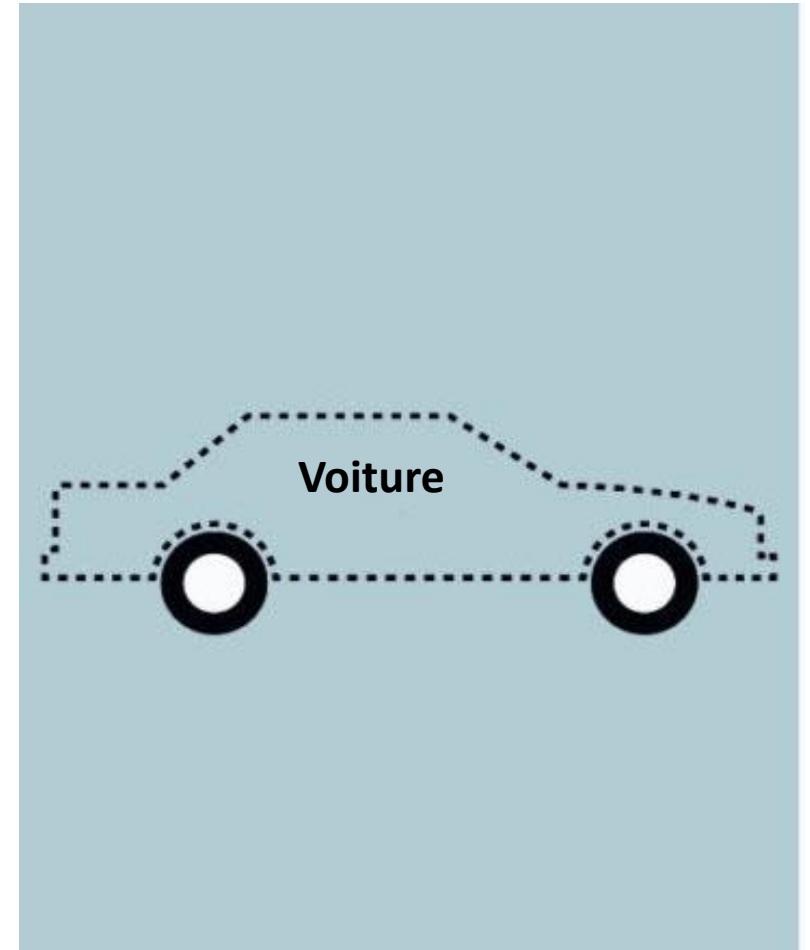
- ❑ Les getters (**getMarque()**, **getVitesse()**)
permettent de lire les valeurs.

- ❑ Les setters (**setMarque()**, **setVitesse()**)
permettent de modifier les valeurs.

- ❑ **Méthode supplémentaire** (non accesseur)

freiner() : met la vitesse à 0, ce qui correspond à l'arrêt total de la voiture.
permettent d'effectuer des actions.

les
accesseurs



Classes et Objets

Classe

■ Règles de nommage des classes en Java

Quelques règles pratiques à respecter lors de la création d'une classe :

Règles

- Le nom d'une classe doit toujours commencer par une lettre majuscule (ex : **Compte**).
- Lorsqu'un nom est composé de plusieurs mots, chaque mot commence par une majuscule : c'est la convention CamelCase (ex. : **CompteBancaire**).
- Chaque classe doit être enregistrée dans un fichier portant le même nom que la classe, suivi de l'extension .java (ex : **CompteBancaire.java**).

Classes et Objets

Objet

Un **objet** est une **instance d'une classe**. Il dispose de son **propre espace mémoire** et possède une **valeur spécifique pour chacun de ses attributs**.

```
class Voiture {  
    String marque;  
    int vitesse;  
  
    void accelerer(int v){  
        vitesse += v;  
    }  
  
    ...  
  
    public String getMarque() {  
        return marque;  
    }  
  
    public void setMarque(String marque) {  
        this.marque = marque;  
    }  
}
```

Objets



marque: Toyota
vitesse: 130
vitesse de la voiture Toyota: 80 km/h.



marque: Jeep
vitesse: 150
vitesse de la voiture Jeep: 80 km/h.



marque: BMW
vitesse: 160
vitesse de la voiture BMW:
80 km/h.

Classes et Objets

Objet

La création effective d'un objet

- ✓ La création d'un objet doit être demandée explicitement dans le programme en utilisant l'opérateur **new**.
- ✓ Cette opération réserve un espace mémoire pour stocker l'objet et initialise ses attributs.

La création d'un objet : syntaxe

new NomDeClasse(); => **Instanciation**

Mémoire

marque = **Toyota**
couleur = **Bleue**
Modèle = **2021**
Carburant = **Essence**
automatique = **true**

Un objet de
type voiture

Objet

La création effective d'un objet

Exemple: création de deux objets distincts de la classe Voiture

```
public static void main(String[] args) {  
  
    Voiture v1 = new Voiture();  
    Voiture v2 = new Voiture();  
  
    // Utilisation des setters pour attribuer les valeurs  
    v1.setMarque("Toyota");  
    v1.setVitesse(80);  
  
    v2.setMarque("Peugeot");  
    v2.setVitesse(60);  
  
    // Utilisation des getters pour lire les valeurs  
    System.out.println(v1.getMarque() + " roule à " + v1.getVitesse() + " km/h");  
    System.out.println(v2.getMarque() + " roule à " + v2.getVitesse() + " km/h");  
}
```

Classes et Objets

Objet

La déclaration: Syntaxe

- **NomDeClasse objetId ;**
- **NomDeClasse objetId1 , objetId2 , . . . ;**

Attention

- La déclaration d'une variable de type primitif réserve directement un emplacement mémoire pour stocker la valeur de cette variable.
- En revanche, la déclaration d'un objet ne réserve pas de place mémoire pour l'objet lui-même, mais seulement pour une **référence** qui pourra pointer vers cet objet plus tard.
- Ainsi, un objet simplement déclaré (non encore créé avec **new**) a pour valeur **null**.

Classes et Objets

L'affectation et comparaison d'objets

■ L'affectation sur des variables de type classe

Variables de type classe (v1 et v2), destinées à contenir des références sur des objets.

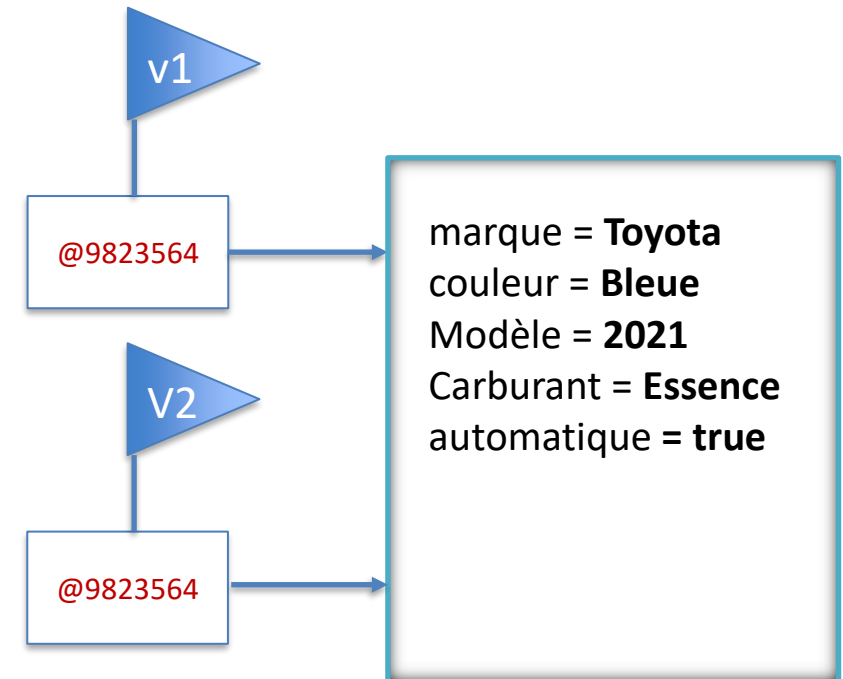
```
Voiture v1 = new Voiture();
```

```
Voiture v2 = v1;
```

Remrques

- Les variables v1 et v2 contiennent la même référence en mémoire.
- Elles pointent donc toutes les deux vers le même objet.
- Par conséquent, toute modification effectuée via l'une des deux variables affecte le même objet.

Références
vers le même
objet



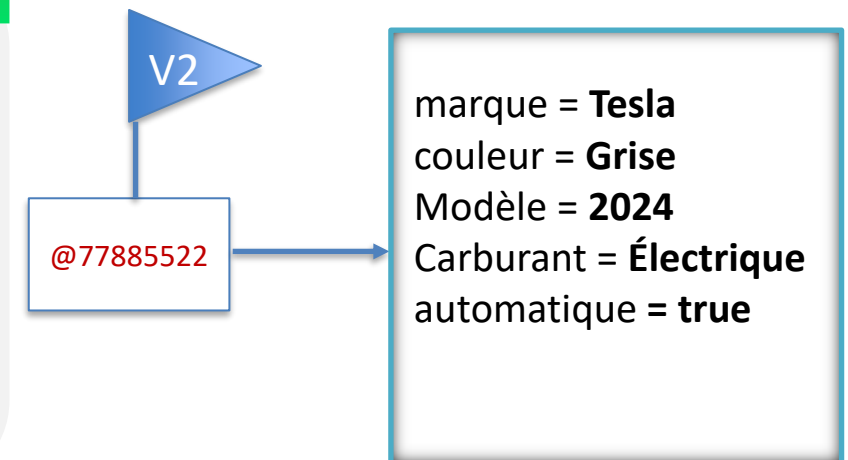
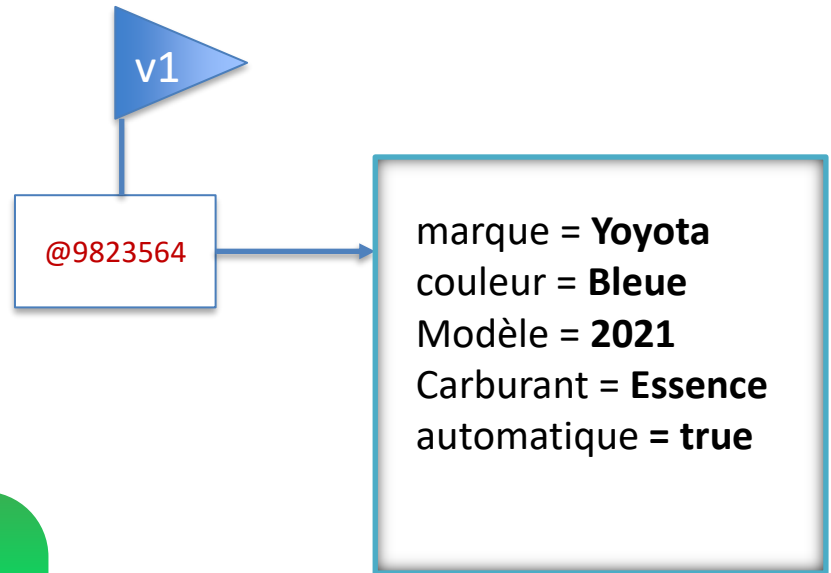
Classes et Objets

L'affectation et comparaison d'objets

■ La création des objets différents

```
Voiture v1 = new Voiture();
```

```
Voiture v2 = new Voiture();
```



Remrques

- Chaque appel à **new** Voiture() crée un nouvel objet dans la mémoire.
- Donc ici, deux objets différents de type Voiture sont créés.
- Les objets référencés par v1 et v2 sont indépendants : une modification sur l'un n'affecte pas l'autre.

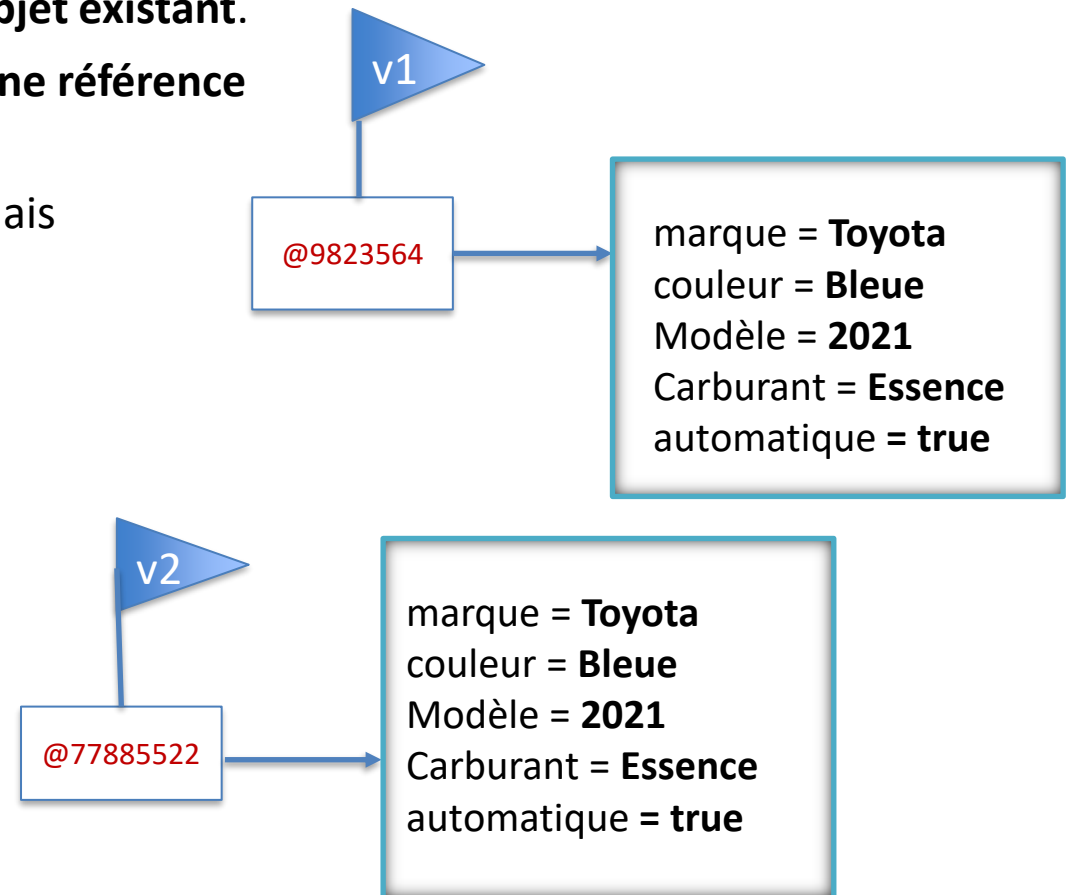
Classes et Objets

L'affectation et comparaison d'objets

■ Des objets différents en mémoire, mais identiques par leur contenu

- La méthode **clone()** permet de **créer une copie exacte d'un objet existant**.
- L'objet obtenu possède **les mêmes valeurs d'attributs**, mais **une référence mémoire différente**.
- Autrement dit, les deux objets sont **identiques en contenu**, mais **indépendants en mémoire**.

```
Voiture v1 = new Voiture();  
v1.setMarque("Toyota");  
v1.setCouleur("Bleue");  
  
⋮  
  
Voiture v2 = (Voiture) v1.clone();
```



Classes et Objets

L'affectation et comparaison d'objets

■ La comparaison d'objets en Java

En Java, il existe **deux façons de comparer des objets** :

- **Comparer leurs références** (leurs adresses en mémoire)
- **Comparer leurs contenus** (les valeurs de leurs attributs)

01

Comparaison des références : opérateur ==

- L'opérateur == compare les valeurs des variables de référence.
- Il vérifie si **deux variables pointent vers le même objet** en mémoire.

```
Voiture v1 = new Voiture();
```

```
Voiture v2 = v1;
```

```
System.out.println(v1 == v2); // true → même référence
```

Classe et Objet

L'affectation et comparaison d'objets

■ La comparaison d'objets en Java

02

Comparaison du contenu : méthode equals()

- Pour comparer les **valeurs internes** (attributs) de deux objets, on utilise la méthode equals().

N.B. Par défaut, equals() de la classe Object agit comme == (compare les références).
Il faut donc **redéfinir la méthode equals()** dans la classe pour comparer le **contenu réel**.

@Override

```
public boolean equals(Object obj) {  
    if (obj == null) return false;  
    if (!(obj instanceof Voiture)) return false;           // Vérifier le type  
    Voiture autre = (Voiture) obj;  
    return modele == autre.modele && marque==autre.marque && couleur==autre.couleur  
        && carburant == autre.carburant && automatique == autre.automatique;  
}
```

Classes et Objets

Le mot-clé null

Le mot-clé **null** est un littéral qui représente une référence nulle, c'est-à-dire une référence qui ne pointe vers aucun objet.

■ Utilisation

Le mot-clé **null** peut être attribué à n'importe quelle variable de type référence, comme les objets.

Il est couramment utilisé dans divers scénarios

- Pour initialiser des variables de type référence.
- Pour vérifier si un objet est instancié et gérer des exceptions.

Classes et Objets

Le mot-clé null

- **Exemple 1:** Initialisation d'une variable de référence

```
public class NullExample {  
    public static void main(String[] args) {  
        String str = null;  
        if (str == null) {  
            System.out.println("The string is null.");  
        }  
    }  
}
```

- **Exemple 2:** Vérification de la nullité avant l'appel de la méthode

```
public class NullCheckExample {  
    public static void main(String[] args) {  
        String str = null;  
        if (str != null) {  
            System.out.println(str.length());  
        } else {  
            System.out.println("The string is null, cannot call methods on it.");  
        }  
    }  
}
```

```
public static void main  
(String args[])  
{  
  
    Voiture v ;  
    v.setMarque("Tesla") ;
```

```
// erreur de compilation :  
// p n'a pas encore reçu de  
// valeur  
.....  
}
```

Le mot-clé this

- ✓ Dans une méthode d'une classe, **this** représente l'objet qui exécute la méthode.

```
public class Voiture {  
    private String marque;  
    private int vitesse;  
    public setMarque(String marque) {  
        this.marque = marque; // this.marque' désigne l'attribut de l'objet  
    }  
    public setVitesse(int vitesse) {  
        this.vitesse = vitesse; //le paramètre 'couleur' est affecté à  
                                l'attribut de même nom  
    }  
}
```

Classes et Objets

Les packages

Le mot-clé **package** est utilisé pour **regrouper des classes, interfaces et sous-packages** qui couvrent un même domaine.



Pour organiser le code

Pour éviter les conflits de noms

Aide à assurer la protection d'accès

Classes et Objets

Les packages

■ Utilisation et syntaxe

La déclaration du package se place **au tout début** du fichier source Java.

Syntaxe

```
package nomdupackage;
```

Remarques

- ✓ package doit être **la première ligne** du fichier.
- ✓ Ici, la classe MyClass fait partie du package **com.example.myapp**.

Exemple

```
package com.example.myapp;  
  
public class MyClass {  
    public void display() {  
        System.out.println("Hello from  
MyClass");  
    }  
}
```

Classes et Objets

Les packages

■ Importer un package

Syntaxe

```
import nomdupackage;
```

- ✓ Le mot-clé **import** permet d'utiliser des classes d'un autre package.
- ✓ MyClass du package com.example.myapp est importée dans la classe main.

Exemple

```
import com.example.myapp.MyClass;  
  
public class main {  
    public static void main(String[] args) {  
        MyClass myClass = new MyClass();  
        myClass.display();  
    }  
}
```

Il y a deux manière d'utiliser les imports :

```
import nomPackage.*;
```

toutes les classes du package sont importées

```
import nomPackage.nomClasse;
```

appel à une seule classe : l'avantage de cette notation est de réduire le temps de compilation

Classes et Objets

Hiérarchie de packages

En Java, les **packages** sont organisés de **manière hiérarchique**, c'est-à-dire qu'il est **possible d'imbriquer des packages** les uns dans les autres.



package com.example.monapp;

Ici, **com** contient **example**, qui contient **monapp**.

■ Correspondance avec les dossiers

- ✓ Java impose que la **structure des dossiers sur le disque** corresponde **exactement** à la hiérarchie des packages.

```
src/  
└─ com/  
    └─ example/  
        └─ monapp/  
            └─ MaClasse.java
```

Classes et Objets

Les modificateurs d'accès

Les modificateurs d'accès définissent **le niveau de visibilité** des **classes, attributs, méthodes** et **constructeurs** dans un programme Java.

Ils permettent de **protéger les données** et de **contrôler l'accès** aux membres d'une classe.

Modificateur	Role
public	Une variable, méthode ou classe déclarée public est visible par tous les autres objets.
par défaut : package-private	Il n'existe pas de mot clé pour définir ce niveau, qui est le niveau par défaut lorsqu'aucun modificateur n'est précisé. Cette déclaration permet à une entité (classe, méthode ou variable) d'être visible par toutes les classes se trouvant dans le même package
protected	Si une méthode ou une variable est déclarée protected, seules les méthodes présentes dans le même package que cette classe ou ses sous-classes pourront y accéder. On ne peut pas qualifier une classe avec protected.
private	C'est le niveau de protection le plus fort. Les composants ne sont visibles qu'à l'intérieur de la classe : ils ne peuvent être modifiés que par des méthodes définies dans la classe et prévues à cet effet.

Classes et Objets

Méthodes

les méthodes de classe sont des **fonctions définies dans une classe qui décrivent les comportements ou les actions** que les objets de la classe peuvent effectuer.

Syntaxe

```

modificateurs type_retourné nom_méthode ( arg1, arg1, ... )
{
    // le corps de la méthode
}
  
```

liste de paramètres

✓ Le type retourné peut être primitif ou correspondre à un objet. Si la méthode ne retourne rien, alors on utilise **void**.

Signature d'une méthode

La **signature d'une méthode** est la combinaison de son nom et de la liste de ses paramètres (nombre, types et ordre).

Exemple

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

Nom de la méthode: additionner

Liste des paramètres: int a, int b

Type de retour: int

Modificateurs: public

Corps: **return** a+b;

Classes et Objets

La surcharge de méthodes (Overloading)

La **surcharge de méthodes** consiste à **déclarer plusieurs méthodes portant le même nom**, mais ayant **des signatures différentes** (types, nombre ou ordre des paramètres).

Exemple

```
public class Calculatrice {  
    // Méthode 1  
    public int additionner(int a, int b) {  
        return a + b;  
    }  
    // Méthode 2 (signature différente)  
    public double additionner(double a, double b)  
    {  
        return a + b;  
    }  
    // Méthode 3 (nombre de paramètres différent)  
    public int additionner(int a, int b, int c) {  
        return a + b + c;  
    }  
}
```

Remarque

- ◆ La surcharge permet d'utiliser un même nom de méthode pour plusieurs comportements.
- ◆ Le compilateur choisit la bonne version selon les paramètres passés.

Polymorphisme (statique)

- ✓ Surcharge est une forme de polymorphisme statique.
- ✓ Le compilateur choisit **quelle méthode exécuter** en fonction des **paramètres utilisés à la compilation**.

Classes et Objets

La surcharge de méthodes: cas d'ambiguïté

Une **ambiguïté** se produit quand le compilateur ne peut pas déterminer **quelle méthode surchargée** doit être appelée.

Exemple 1

```
public class TestAmbigu {
    void afficher(int a, double b) {
        System.out.println("int - double");
    }

    void afficher(double a, int b) {
        System.out.println("double - int");
    }

    public static void main(String[] args) {
        TestAmbigu t = new TestAmbigu();
        t.afficher(5, 5); // Erreur : appel ambigu!
    }
}
```

Exemple 2

```
public class Point {
    private int x, y;

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

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

    public static void main(String[] args) {
        Point a = new Point();
        int n; byte b;
        a.deplace(n, b); // Appel de deplace(int, byte)
        a.deplace(b, n); // Appel de deplace(byte, int)
        a.deplace(b, b); // Erreur : ambiguïté
    }
}
```

Classes et Objets

La surcharge de méthodes: règles générales

■ Compatibilité par affectation

Un **type A** est compatible par affectation avec un type B, si une valeur de type B peut être affectée à une variable de type A sans erreur de compilation.

```
A variable = valeurDeTypeB;
```

Java permet certaines **conversions implicites** entre types numériques :

byte → **short** → **int** → **long** → **float** → **double**

Si une classe hérite d'une autre, alors une instance de la sous-classe peut être affectée à une variable de la superclasse.

String -> **Objet**

- ✓ Une méthode est dite **moins compatible** lorsqu'elle peut accepter un argument grâce à une conversion implicite (ou parce que son paramètre est d'un type plus général).

```
byte b = 10;  
int i = b;      //compatible (byte → int)  
double d = i;  // compatible (int → double)
```

```
String s = "Bonjour";  
Object o = s;  
// compatible : String → Object
```

```
void afficher(Object o) {}  
void afficher(String s) {}  
afficher("Bonjour!");
```

Classes et Objets

La surcharge de méthodes: règles générales

■ Principe d'élimination successive

Quand plusieurs méthodes surchargées peuvent correspondre à un même appel, le compilateur recherche toutes **les méthodes acceptables** et **il choisit la meilleure si elle existe**.

Recherche des méthodes acceptables

Une méthode est considérée comme acceptable si :

Elle a le bon nombre d'arguments,

Les types des arguments sont compatibles avec ceux de la méthode,

Elle est accessible (ex. : non private si l'appel est externe).

Étapes de sélection

- **Aucune méthode acceptable**
→ Erreur de compilation.
- **Une seule méthode acceptable**
→ Elle est utilisée.
- **Plusieurs méthodes acceptables**
→ Le compilateur compare les signatures :
 - Il élimine les méthodes "moins compatibles".
 - Si une seule reste
→ utilisée.
 - Si plusieurs restent
→ Erreur d'ambiguïté.

Classes et Objets

Les constructeurs

Les constructeurs sont des méthodes spéciales utilisées **pour initialiser les objets**. Contrairement aux méthodes classiques, les constructeurs sont **invoqués lors de la création d'une instance** d'une classe.

Le constructeur est
une méthode qui :

n'a pas de type de retour
(pas même void) .

porte le même nom que la classe

est automatiquement appelée
lors de la création d'une
instance.

Exemple

```
public class Voiture {  
    private String marque;  
    private String couleur;  
    // Constructeur  
    public Voiture(String marque, String couleur) {  
        this.marque = marque; // initialise l'attribut marque  
        this.couleur = couleur; // initialise l'attribut couleur  
    }  
    public void afficher() {  
        System.out.println("Marque : " + marque + ", Couleur : " + couleur);  
        System.out.println();  
    }  
    public static void main(String[] args) {  
        Voiture v1 = new Voiture("Toyota", "Bleue"); // appel automatique  
        v1.afficher(); // du constructeur  
    }  
}
```

Classes et Objets

Types de constructeurs

01

Constructeur par défaut

Un constructeur par défaut est automatiquement fourni par le compilateur Java si aucun constructeur n'est explicitement défini dans la classe. Il initialise l'objet avec des valeurs par défaut.

02

Constructeur sans argument

Un constructeur sans argument est explicitement défini par le programmeur et ne prend aucun paramètre. Il est similaire au constructeur par défaut mais peut inclure un code d'initialisation personnalisé.

03

Constructeur paramétré

Un constructeur paramétré accepte des arguments pour initialiser un objet avec des valeurs spécifiques. Cela permet une initialisation plus souple et plus contrôlée des attributs de l'objet.

04

Constructeur de copie

Un constructeur de copie est utilisé pour créer un nouvel objet en tant que copie d'un objet existant. Java ne fournit pas de constructeur de copie par défaut, mais il est possible de l'implémenter manuellement.

Classes et Objets

Surcharge des constructeurs

Comme les méthodes, les constructeurs peuvent être **surchargés**.
La surcharge s'effectue en **changeant le nombre, le type ou l'ordre des paramètres**.

Exemple 1

```
public class Voiture {
    private String marque;
    private String couleur;
    // Constructeur 1 : sans paramètre
    public Voiture() {
        marque = "Inconnue";
        couleur = "Blanche";
    }
    // Constructeur 2 : avec un paramètre
    public Voiture(String marque) {
        this.marque = marque;
        couleur = "Noire";
    }
    // Constructeur 3 : avec deux paramètres
    public Voiture(String marque, String couleur) {
        this.marque = marque;
        this.couleur = couleur;
    }
}
```

Exemple 2 (Constructeur par copie)

```
public class Voiture {
    private String marque;
    private String couleur;
    // Constructeur normal
    public Voiture(String marque, String couleur) {
        this.marque = marque;
        this.couleur = couleur;
    }
    // Constructeur par copie
    public Voiture(Voiture autre) {
        this.marque = autre.marque;
        this.couleur = autre.couleur;
    }

    public static void main(String[] args) {
        Voiture v1 = new Voiture("Toyota", "Bleue"); // Création //normale
        Voiture v2 = new Voiture(v1); // Copie de v1
    }
}
```

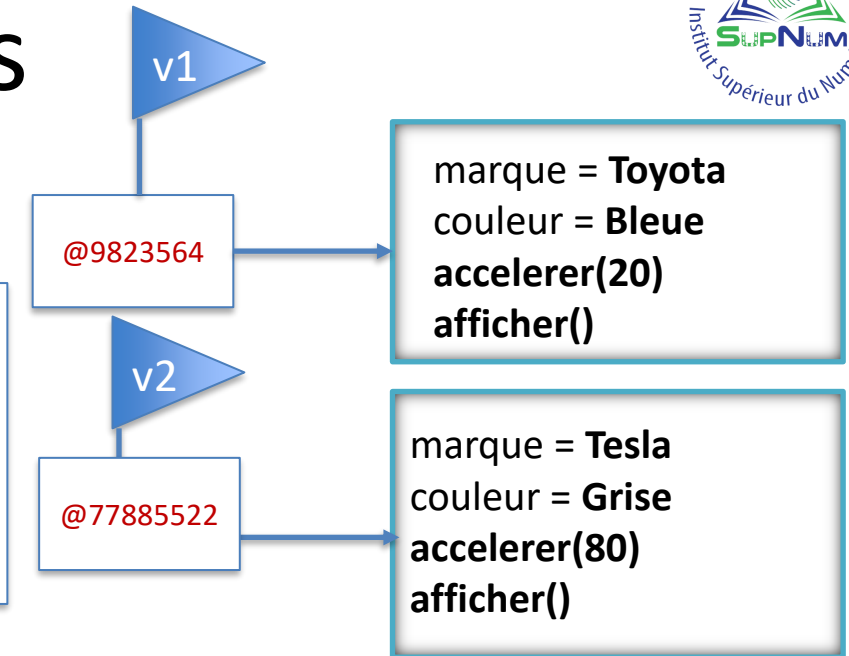
Classes et Objets

Champs et méthodes de classe

Les champs d'instance et les méthodes d'instance sont des éléments propres à chaque objet créé à partir d'une classe.

Champs d'instance : variables représentant l'état d'un objet.

Méthodes d'instance : fonctions décrivant le comportement d'un objet.



Chaque objet de type Voiture possède sa propre marque et sa propre couleur.

```
public class Voiture {
    String marque; // Champs d'instance
    String couleur; // Champs d'instance
    int vitesse; // Champs d'instance
    public void accelerer(int ajout) { // Méthode d'instance
        vitesse += ajout;
    }
    public void afficher() { // Méthode d'instance
        System.out.println("Marque : " + marque + ", Vitesse : "
            + vitesse);
    }
}
```

Utilisation

```
Voiture v1 = new Voiture();
v1.marque = "Toyota";
v1.accelerer(20);
v1.afficher();

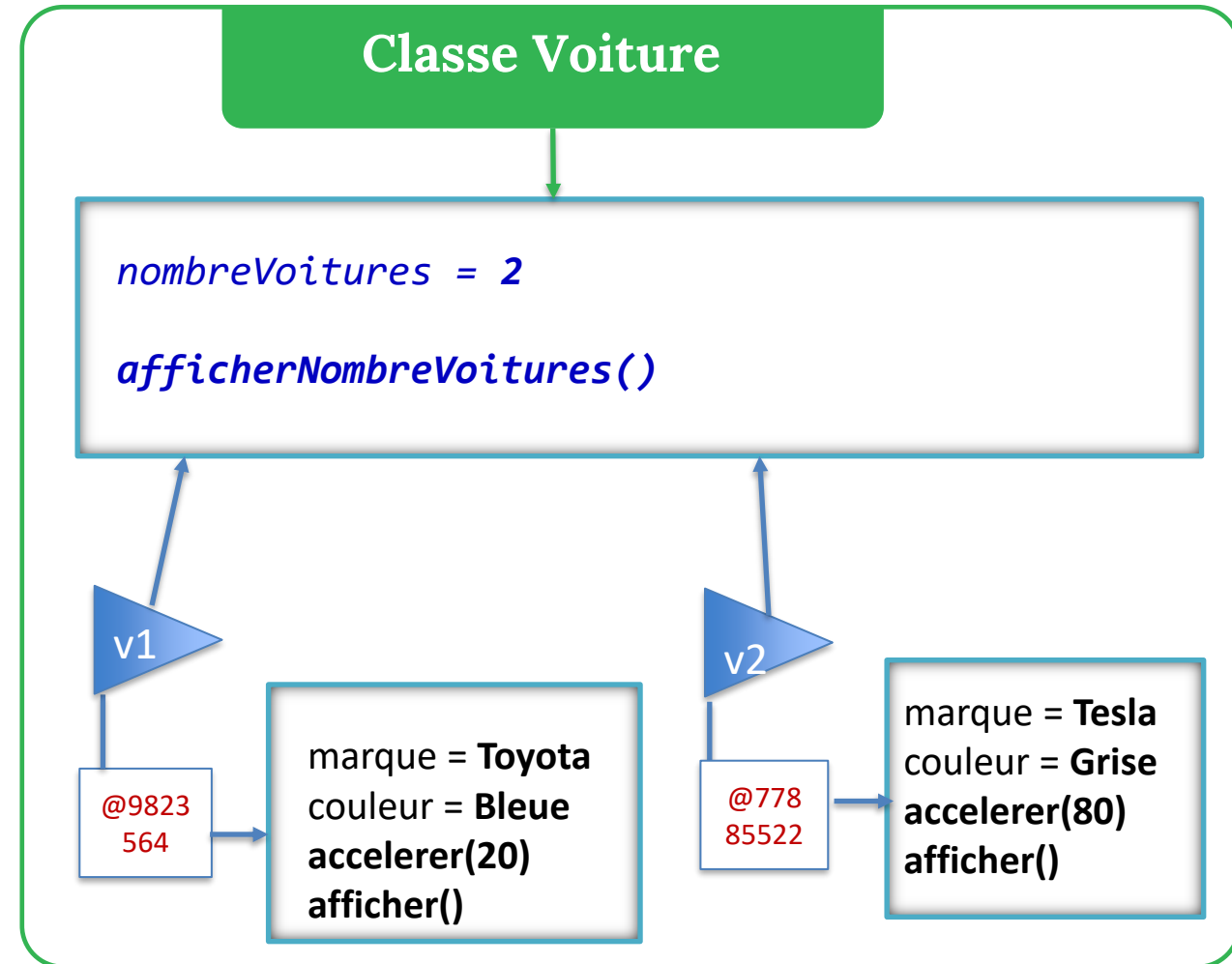
Voiture v2 = new Voiture();
v2.marque = "Tesla";
v2.accelerer(80);
v2.afficher();
```

Classes et Objets

Champs et méthodes de classe

Le mot-clé **static** indique qu'un **membre (variable ou méthode)** appartient à la **classe elle-même**, et non à une instance particulière.

- ✓ Une variable d'instance → **propre à chaque objet.**
- ✓ Une variable **static** → **partagée par tous les objets de la classe.**



Classes et Objets

Champs et méthodes de classe

```
public class Voiture {  
    private String marque;  
    private String couleur;  
    private int vitesse;  
    static int nombreVoitures = 0; // Variable de classe  
  
    public Voiture(String marque, String couleur) {  
        this.marque = marque;  
        this.couleur = couleur;  
        nombreVoitures++; // Incrémenter le compteur commun  
    }  
    // Méthode d'instance  
    public void accélérer(int ajout) {  
        vitesse += ajout;  
    }  
    // Méthode d'instance  
    public void afficher() {  
        System.out.println("Marque : " + marque + ", Couleur : " + couleur);  
    }  
    // Méthode de classe  
    public static void afficherNombreVoitures() {  
        System.out.println("Nombre total de voitures : " + nombreVoitures);  
    }  
}
```

Classes et Objets

Attribut final

Un attribut **final** est une variable dont la valeur **ne peut être attribuée qu'une seule fois**.

Toute tentative de **modification ultérieure provoque une erreur de compilation**.

Il doit être initialisé explicitement :

- soit dans la déclaration,
- soit dans le constructeur (mais pas plus tard).

Exemple 1 – Initialisation directe :

```
private final int n = 20;
```

Tous les objets
auront la même
valeur pour n.

Exemple 2 – Initialisation dans le constructeur :

```
Class A{
  private final int n;
  public A() { n = 10; }}
```

Exemple 3 – Valeur différente par objet :

```
private final
int n;
public A(int nn)
{ n = nn; }
```

Règles importantes

- Un champ final doit être initialisé (pas d'initialisation par défaut).
- Il ne peut pas être modifié après affectation.
- L'initialisation doit avoir lieu au plus tard dans le constructeur.

Remarques

- Un champ final peut être public ou privé.
- Le mot-clé final peut aussi s'appliquer à :
 - ✓ une méthode → empêche la redéfinition (override),
 - ✓ une classe → empêche l'héritage.

Chaque objet peut avoir une
valeur différente pour n.

Fin du Chapitre

