



Cours 1 : Introduction à Java

Mardi 25 Janvier 2022

Ulysse Coscoy (ucoscoy@excilys.com)

Mathéo Allard (mallard@excilys.com)

Alexander van Dalen (avan-dalen@excilys.com)

Guillaume Da Silva (gdasilva@excilys.com)

Excilys
Développeurs de passion

- Tous les mardi de 9h à 12h15
- 3 Cours + 3 TP
- Un mini projet
- Beaucoup de fun !

SOMMAIRE

- I. Présentation de Java
- II. Fonctionnement de Java
- III. Syntaxe de base de Java
- IV. Java, un langage orienté objet

Présentation de Java



1. Java est un langage de programmation...
 - généraliste (\neq dédié)
 - de haut niveau
 - créé en 1995 par Sun Microsystems (racheté par Oracle en 2009)
 - orienté objet
2. Sa version la plus récente est la version 17 (sortie en septembre 2021)
3. En TP, nous utiliserons la version 8 de Java

- Java dispose d'une communauté active (nombreux frameworks créés et maintenus par la communauté et pour la communauté)
- Il est basé sur le principe WORA (Write Once, Run Anywhere)
- C'est un langage...
 - interprété
 - portable
 - fortement typé !!
- Il gère la mémoire tout seul !



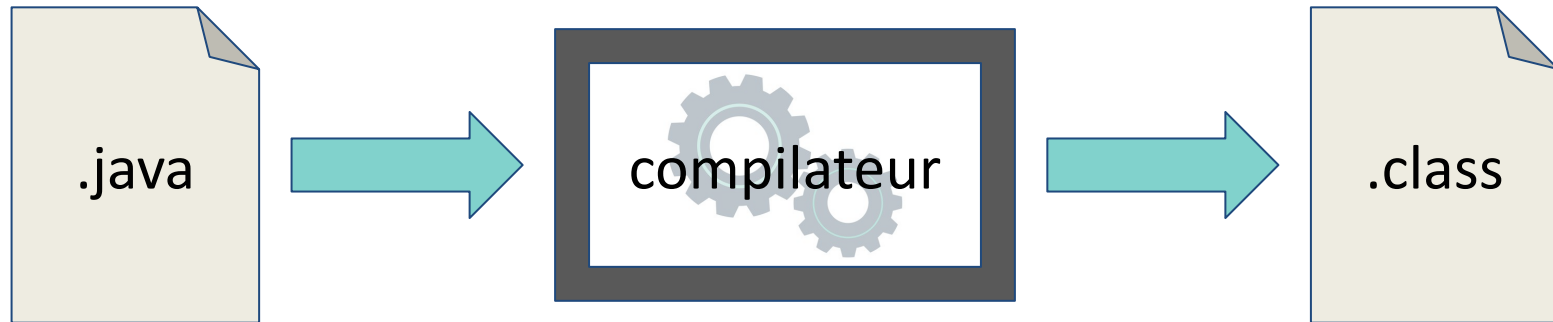
Fonctionnement de Java

Compilation du code source

- Java est un langage **interprété**, pourtant il y a bien une étape de **compilation** !
- Rappel : interprétation vs. compilation
 - Langage **compilé** \Rightarrow les fichiers sources sont convertis sous forme de fichiers de code binaire par un compilateur.
 - Langage **interprété** \Rightarrow les fichiers sources sont interprétés au fur et à mesure de l'exécution par un interpréteur.
- Les fichiers source peuvent être édités à l'aide de n'importe quel éditeur de texte (Bloc-Notes, VS Code, Atom, Vim...)

- En Java, on écrit le code source dans des fichiers ***.java**
- Ces fichiers doivent ensuite être compilés vers du bytecode Java
- La compilation produit autant de fichiers ***.class** qu'il y a de classes dans le code source.
Pour l'instant, on admet que : un fichier *.java = une classe = un fichier *.class
- Ce sont les fichiers *.class qui seront interprétés ensuite

- Résumé en image



- En pratique, on compile ses fichiers source avec la commande `javac`, en précisant le (ou les) nom du fichier *.java que l'on souhaite compiler
 - Exemple : `javac HelloWorld.java`
- Par défaut, les fichiers *.class sont créés dans le répertoire contenant le fichier *.java correspondant
 - On peut spécifier un dossier de sortie différent avec l'option `-d`
`javac -d /tmp/mesClasses HelloWorld.java`
- Un doute sur la syntaxe ? \Rightarrow `man javac`

- Résumé en image



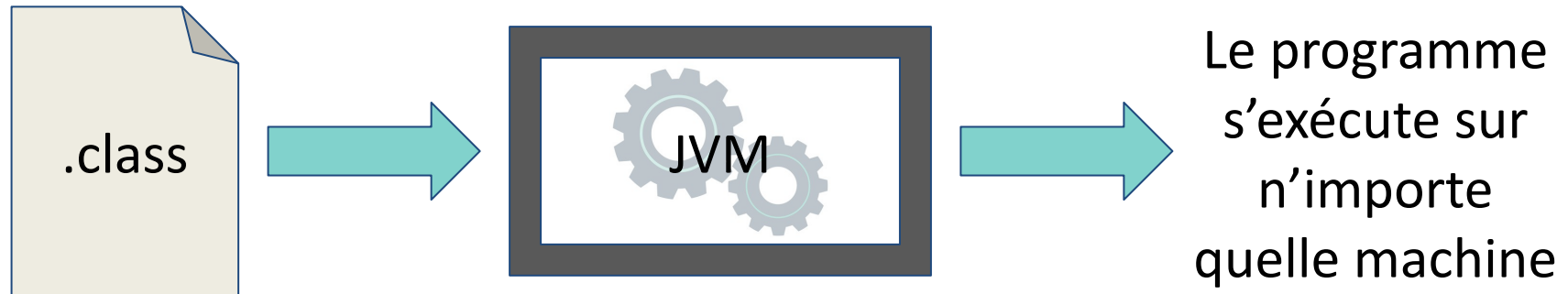
Fonctionnement de Java

Exécution du code par la JVM

- En Java, l'**interpréteur** utilisé pour exécuter les fichiers de bytecode est la **JVM** (Java Virtual Machine)
- Son rôle est de transformer le code Java compilé en code machine
- La JVM effectue aussi une tâche d'optimisation du code au cours de l'exécution

- Il existe différentes versions de la JVM, en fonction des systèmes d'exploitation. Mais toutes ces versions interprètent le bytecode Java de la même manière.
- Gros avantage de cela : le code **Java est portable** !

- Résumé en image



- En pratique, on exécute un fichier de bytecode Java avec la commande `java`, en précisant le (ou les) nom de la classe que l'on souhaite exécuter (= nom du fichier sans l'extension `.class`)
 - Exemple : `java HelloWorld`
- Un doute sur la syntaxe ? \Rightarrow `man java`

Syntaxe de base de Java

- Java est **sensible à la casse** !
- La syntaxe du langage Java est proche de celle du **langage C**
 - Une instruction se termine par un **point-virgule** « ; »
 - Les **commentaires** peuvent être sur une seule ligne ou sur plusieurs lignes

```
// commentaire inline
/* commentaire
multilignes */
```
 - Les commentaires permettent de faire de la **documentation** générée :

```
/** @param @return */
```

Syntaxe de base de Java

Construction des identifiants

- Identifiant = nom de variable, de fonction, de classe, ...
- Composé exclusivement des caractères suivants :
 - [A-Z] (lettres de l'alphabet en majuscules)
 - [a-z] (lettres de l'alphabet en minuscules)
 - _ (underscore)
 - \$ (dollar)
 - [0-9] (chiffres)
- Ne peut pas commencer par un chiffre
- Ne peut pas être un mot-clé du langage Java
- Par convention, on utilise le **camelCase**

- Mots-clés réservés en Java 8 :

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

- Ces identifiants sont-ils valides ou invalides ?
 - abc123
 - AzeRtY
 - 42uioP
 - aZe_rty\$uiOp
 - azerty-uiop
 - \$azerty
 - _azerty
 - _\$
 - case

- Ces identifiants sont-ils valides ou invalides ?
 - abc123
 - AzeRtY
 - 42uioP → l'identifiant ne peut pas commencer par un chiffre
 - aZe_rty\$uiOp
 - azerty-uiop → le tiret n'est pas un caractère autorisé
 - \$azerty
 - _azerty
 - _\$
 - case → c'est mot-clé réservé du langage Java

Syntaxe de base de Java

Les types de base

- Il existe 8 types de base, dits « types primitifs » :
 - boolean
 - byte
 - short
 - int
 - long
 - float
 - double
 - char
- Nombres entiers respectivement sur 1, 2, 4 ou 8 octets
- Nombres à virgule, respectivement sur 4 ou 8 octets
- Leur nom commence par une **minuscule**

- Il existe 8 types de base, dits « **types primitifs** » :
 - `boolean` → `false`
 - `byte` → `0`
 - `short` → `0`
 - `int` → `0`
 - `long` → `0`
 - `float` → `0.0`
 - `double` → `0.0`
 - `char` → `'\u0000'`
- Leur nom commence par une **minuscule**

- On ajoute aussi le type `String`
 - Ce n'est pas primitif, mais un type **Objet**
 - Son nom commence par une **majuscule**
 - `"azerty"`

Syntaxe de base de Java

Déclarer et utiliser des variables

- La déclaration d'une variable se fait selon le modèle suivant : `type identifiant;`
 - `int maVariable;`
- L'affectation se fait avec l'opérateur =
 - `maVariable = 12;`
- On peut compacter les deux étapes précédentes de la façon suivante :
 - `int maVariable = 12;`

- L'utilisation d'une variable se fait en utilisant simplement son identifiant dans le code
 - `maVariable + 1;`
- On peut définir plusieurs variables dans une même instruction en les séparant par une virgule
 - `boolean a, b, c;`
- On peut également y ajouter l'affectation
 - `double d1 = 3.1415, d2 = 42.0, d3;`

- Par défaut en Java, tout nombre entier est de type `int`. Pour créer expressément une valeur de type `long`, il faut ajouter un «`l`» à la fin du nombre.
 - `long marignan = 15151l;`
- Par défaut en Java, tout nombre à virgule est de type `double`. Pour créer expressément une valeur de type `float`, il faut ajouter un «`f`» à la fin du nombre.
 - `float pi = 3.141592f;`

- Il n'y a presque **pas de conversion implicite** en Java. Il faut le faire explicitement à l'aide d'un « cast »

```
int i;
double d = -12.0;
i = (int) d; // fonctionne
i = d;       // ne fonctionne pas
```

- Il y en a une dans le cas où un primitif est de taille inférieure au primitif recherché.

```
d = i; // fonctionne
```

Syntaxe de base de Java

Écrire dans la console

- Pour écrire sur la console, on peut utiliser la fonction `System.out.println()` fournie par le JDK
- `println()` accepte tous les types primitifs ainsi que les types `String`, tableaux et `Object`

```
int i = -15;

System.out.println(i);
System.out.println("La variable i vaut " + i);
```

Syntaxe de base de Java

Les opérateurs

- Nous avons déjà vu

- Affectation : =

```
int vingtSept;  
vingtSept = 27;
```

- Conversion (cast) : (type) valeur

```
byte b = 2;  
short s;  
s = (short) b;
```

- Opérations arithmétiques
 - Addition : +
 - Soustraction : -
 - Multiplication : *
 - Division : /
 - Modulo : %
 - Ces opérateurs peuvent être combinés à l'affectation :
+=, -=, *=, /=, %=

- Opérations arithmétiques
 - Incrémentation : ++
 - Incrémentation préfixe : ++i
(incrémente i et retourne sa valeur **après** incrémentation)
 - Incrémentation postfixe : i++
(incrémente i et retourne sa valeur **avant** incrémentation)
 - Décrémentation : --
 - Décrémentation préfixe : --i
(décrémente i et retourne sa valeur **après** décrémentation)
 - Décrémentation postfixe : i--
(décrémente i et retourne sa valeur **avant** décrémentation)

- Opérations logiques
 - Comparaisons
 - Inférieur : <
 - Supérieur : >
 - Inférieur ou égal : <=
 - Supérieur ou égal : >=
 - Égalité : ==
 - Inégalité : !=
 - Négation : !
 - Et : & &
 - Ou : | |

- On peut utiliser l'opérateur + pour **concaténer** des chaînes de caractères
 - `"Hello" + "World"` produit la chaîne `"HelloWorld"`

Syntaxe de base de Java

Les structures conditionnelles

- **if ... else**

- La structure minimale contient l'instruction `if`

```
if (booléen) {
    // faire quelque chose
}
```

- On peut ajouter d'autres cas avec l'instruction `else if`

```
else if (booléen) {
    // faire autre chose
}
```

- On peut ajouter une règle pour tous les autres cas avec `else`

```
else {
    // faire autre chose
}
```

- switch

- ```
switch (expression) {
 case constante1: // faire quelque chose
 case constante2: // faire quelque chose
}
```

- On peut ajouter une règle pour tous les autres cas avec default

```
switch (expression) {
 case constante1: // faire quelque chose
 case constante2: // faire quelque chose
 default: // faire autre chose
}
```

- switch

- Utiliser l'instruction `break`;

```
switch (expression) {
 case constante1:
 // faire quelque chose
 break;
 case constante2:
 // faire quelque chose
 break;
 default: // faire autre chose
}
```

- Sans cette instruction, Java exécute toutes les instructions du switch qui suivent le cas correspondant à l'expression

- Les ternaires

- Structure d'une ternaire :

```
booléen ? actionSiVrai : actionSiFaux;
```

- On peut s'en servir dans une affectation ou non

```
int i = (12 < 27) ? 0 : 32;

int j = (i++ == 0) ? ++i : --i;
```

- Quelle est la valeur de j ?

# Syntaxe de base de Java

## Les boucles

- Il existe trois types de boucles :

- Les boucles conditionnelles

```
for (variable ; condition d'arrêt ; action) { }
```

- Aucun des trois champs du for n'est obligatoire.
- Dans le premier champ, on peut définir et/ou initialiser une ou plusieurs variables.

- Les boucles inconditionnelles « tant que ... faire »

```
while (condition) { }
```

- Les boucles inconditionnelles « faire ... tant que »

```
do {
 // actions à réaliser
} while (booléen);
```



# Java, un langage orienté objet

Présentation de la programmation orientée objet

- La programmation orientée objet
  - Paradigme de programmation informatique
  - Inventée dans les années 1960-70
  - Repose sur la notion d'objet (*merci Sherlock !*)
  - L'idée est de rassembler dans une même entité (l'objet) les données et les traitements qui s'y appliquent

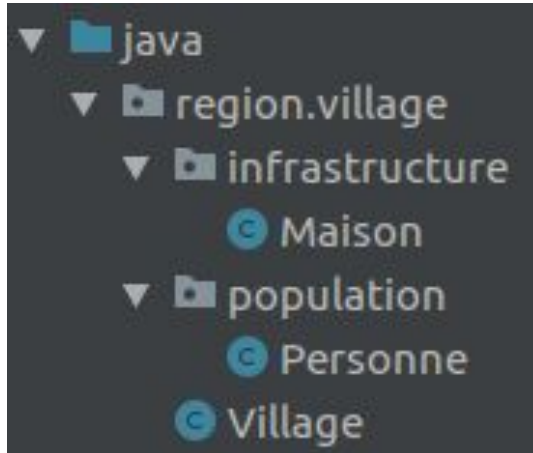
- **Avantages:**
  - Code plus clair, plus structuré, plus facile à comprendre et à maintenir
  - Conventions quasiment universelles
  - Code réutilisable
  - Très adaptée pour utiliser le modèle MVC
- **Inconvénients:**
  - Lourde à mettre en place pour de petits projets
  - Peut ne pas être adaptée dans certains cas

# Java, un langage orienté objet

La programmation orientée objet dans Java

- La base de la programmation orientée objet en Java est la notion de classe
- Les classes sont les squelettes des objets
- On peut considérer les classes comme les types des objets que l'on va manipuler
- Une variable ayant pour type une classe est une référence vers une instance de cette classe

- Un package peut être vu comme un dossier
- Les packages permettent d'organiser les fichiers de code de façon cohérente
- Chaque classe appartient à un package



```
package region.village.infrastructure;

public class Maison {
}
```

# Java, un langage orienté objet

## Concept de classes

- Un **attribut** est une variable appartenant à une classe
- Il peut être de **type** primitif ou objet
- On doit définir son **accessibilité** parmi les options :
  - `Public / Protected / Default Package / Private`
- On peut définir d'autres paramètres : `static`, `final` ...
- On doit nommer l'attribut avec un identifiant

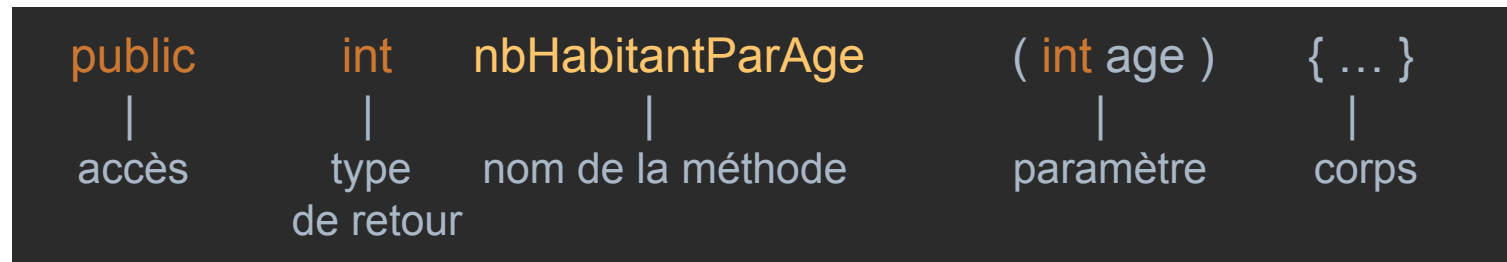
```
public Village village;
protected String adresse;
int nbPiece;
private Personne[] habitants;
```



- **L'accessibilité** d'un élément définit d'où il est accessible
  - **Public** : Accessible depuis n'importe quelle classe
  - **Protected** : Accessible dans les classes situées dans le même package et celles héritant de la classe courante
  - **Default Package** : Accessible seulement depuis les classes situées dans le même package
  - **Private** : Accessible seulement dans la classe

```
public Village village;
protected String adresse;
int nbPiece; // Default Package = laisser vide
private Personne[] habitants;
```

- Une **méthode** est une fonction appartenant à une classe
- Structure:
  - Accessibilité
  - Type de retour
  - Nom de la méthode
  - Paramètre(s)



- **Signature :**

```
public int nbHabitantParAge (int age) { ... }
```

- Deux méthodes ne peuvent pas avoir la même signature
- Deux méthodes ne peuvent pas différer seulement par leur type de retour

```
public int nbHabitantParAge(int age) {...}
```

```
public int nbHabitantParAge(int age1, int age2) {...}
```

```
public int nbHabitantParAge(int age, String name) {...}
```

```
public int nbHabitantParAge(int age2, int age1) {...}
```

```
public int nbHabitantParAge(String name, int age) {...}
```

```
public String nbHabitantParAge(int age1, int age2) {...}
```

- **Signature :**

```
public int nbHabitantParAge (int age) { ... }
```

- Deux méthodes ne peuvent pas avoir la même signature
- Deux méthodes ne peuvent pas différer seulement par leur type de retour

```
public int nbHabitantParAge(int age) {...}
```

```
public int nbHabitantParAge(int age, String name) {...}
```

```
public int nbHabitantParAge(String name, int age) {...}
```

```
public int nbHabitantParAge(int age1, int age2) {...}
```

```
public int nbHabitantParAge(int age2, int age1) {...}
```

```
public String nbHabitantParAge(int age1, int age2) {...}
```

- **Surcharger** une méthode
  - Même type de retour et même nom
  - Type, ordre et/ou nombre de paramètres différents

```
public int nbHabitantParAge(int age) {...}
```

```
public int nbHabitantParAge(int age, String name) {...}
```

```
public int nbHabitantParAge(Village village, int age) {...}
```

- Pour créer un objet et initialiser ses attributs, on utilise un **constructeur**
- Il peut exister plusieurs constructeurs pour une même classe (grâce à la surcharge)
- Si aucun constructeur n'est créé, un constructeur vide est créé par défaut

```
public Personne() {}

public Personne(int age, String nom) {
 this.age = age;
 this.nom = nom;
}

public Personne(String nom) {
 this(0, nom);
}
```

- Les constructeurs n'ont **pas de type de retour**
- On peut appeler un constructeur dans un autre avec le mot-clé `this()`

```
public Personne() {}

public Personne(int age, String nom) {
 this.age = age;
 this.nom = nom;
}

public Personne(String nom) {
 this(0, nom);
}
```

- **L'encapsulation** est un concept dont le but est...
  - de contrôler l'utilisation d'un objet
  - de rendre plus lisible le code associé
- Comment faire de l'encapsulation ?
  - Mettre en privé tous les attributs de la classe
  - Les rendre accessible/modifiable par des méthodes publiques (getter/setter)

```
private int age;
private String nom;
```

```
public String getNom() {
 return nom;
}

public void setNom(String nom) {
 this.nom = nom;
}
```



- Pourquoi utiliser les setters ?
  - Cela permet de vérifier la valeur fournie en paramètre et/ou d'y apporter un traitement supplémentaire

```
private int age;
private String nom;
```

```
public int getAge() {
 return nom;
}

public void setAge(int age) {
 if (age < 18) {
 this.age = 18;
 } else {
 this.age = age;
 }
}
```

- La méthode `toString()` est présente dans tous les objets et retourne l'objet sous forme de `String`
- On peut l'appeler explicitement, mais elle est aussi appelée automatiquement dans certaines conditions

```
public String toString() {
 return "Personne{" +
 "age=" + age +
 ", nom='" + nom + "'" +
 "};"
}
```

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

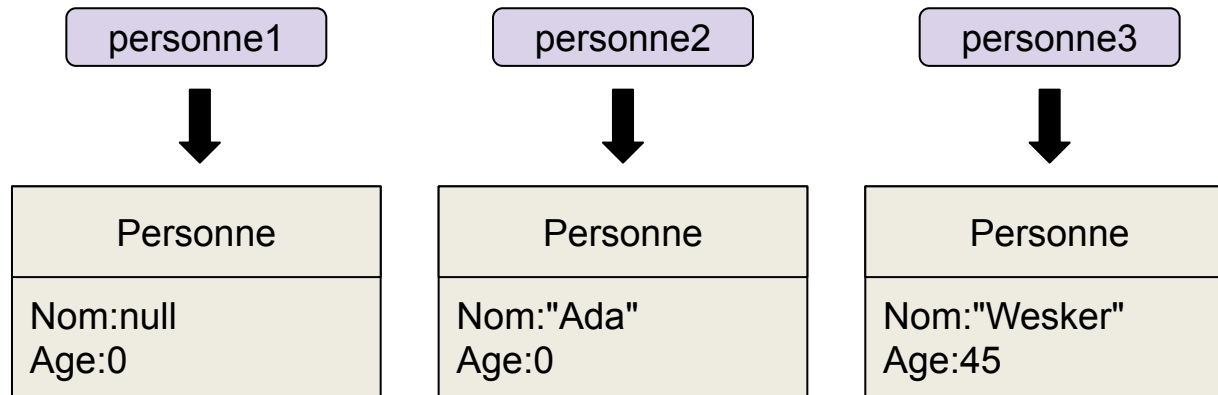


```
System.out.println(personne.toString());
```

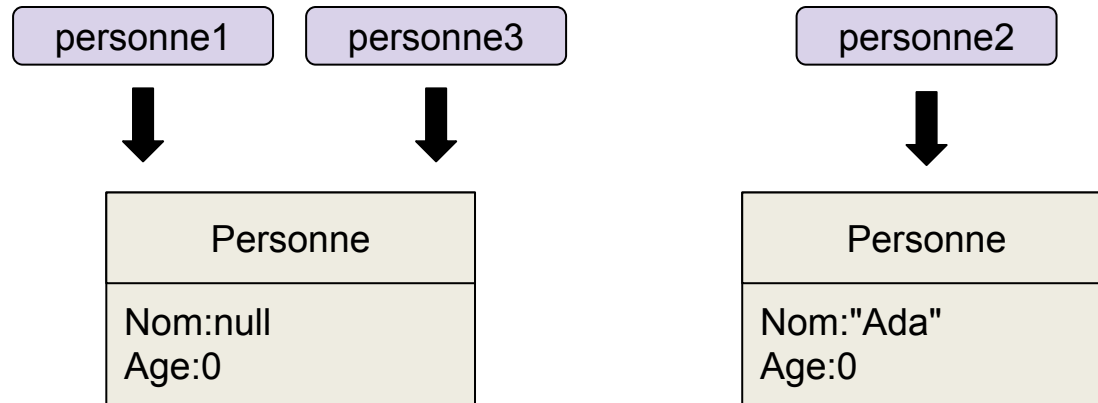
- Pour utiliser un objet, il faut d'abord le créer. On obtient alors une instance de la classe.

```
Personne personne1 = new Personne();
Personne personne2 = new Personne("Ada");
Personne personne3 = new Personne(45, "Wesker");
```

- Et pendant ce temps là dans la mémoire :



```
Personne personne1 = new Personne();
Personne personne2 = new Personne("Ada");
Personne personne3 = personne1;
```



- Lorsqu'on dispose d'une instance de classe, on peut accéder à ses méthodes et attributs
- Pour accéder à une méthode ou un attribut d'une instance de classe, on utilise le point « . »

```
int age = personne.getAge();
```

```
Village village = maison.village;
```

- Lorsqu'on passe une variable en paramètre d'une fonction, deux comportements sont possibles :
  - Si la variable est de **type primitif**, elle est passée par **copie**  
⇒ Si la valeur du paramètre est modifié dans la fonction, **la variable ne change pas de valeur**
  - Si la variable est une **instance de classe**, elle est passée par **référence : c'est une copie de la référence.**  
⇒ Si les attributs sont modifiés, alors l'instance le sera. Si la variable est affectée à une autre instance, alors la première n'est pas affectée.

- On peut définir des attributs ou des méthodes liés à la classe, et non pas à une instance de la classe
- Cela se fait à l'aide du mot-clé `static`
- Un attribut `static` est propre à la classe, il est donc partagé par toutes les instances de la classe
- On peut accéder à un attribut ou une méthode `static` soit via une instance de la classe, soit directement via la classe

- Exemple de classe avec attribut et méthode statiques

```
public class Personne {
 private static int NOMBRE_DE_PERSONNES = 0;

 public static int getCommunityNumber(){
 return NOMBRE_DE_PERSONNES;
 }
}
```

```
public Personne(int age, String nom){
 this.age = age;
 this.nom = nom;
 NOMBRE_DE_PERSONNES++;
}
```

// On appelle la méthode directement avec la classe

```
Personne.getCommunityNumber();
```

// Ou avec une instance de la classe

```
Personne personnelInst = new Personne(0, "");
```

```
personnelInst.getCommunityNumber();
```



```
private static int NOMBRE_DE_PERSONNES = 0;

public int getCommunityNumber() {
 return NOMBRE_DE_PERSONNES;
}
```

```
Personne personne1 = new Personne();
personne1.getCommunityNumber();
```

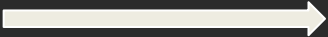
```
Personne personne2 = new Personne();
Personne personne3 = new Personne();
```

```
personne2.getCommunityNumber();
personne3.getCommunityNumber();
```

```
private static int NOMBRE_DE_PERSONNES = 0;

public int getCommunityNumber() {
 return NOMBRE_DE_PERSONNES;
}
```

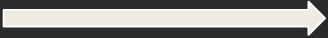
```
Personne personne1 = new Personne();
personne1.getCommunityNumber();
```



1

```
Personne personne2 = new Personne();
Personne personne3 = new Personne();

personne2.getCommunityNumber();
personne3.getCommunityNumber();
```



3

# Java, un langage orienté objet

## Exemple complet

# Java, un langage orienté objet

## Exemple complet

```
public class Personne {

 private static int NOMBRE_DE_PERSONNES = 0;

 private int age;
 private String nom;

 public Personne() {
 this(0, "");
 }

 public Personne(int age, String nom) {
 this.age = age;
 this.nom = nom;
 NOMBRE_DE_PERSONNES++;
 }

 public Personne(String nom) {
 this(0, nom);
 }
}
```

```
 public int getAge() { return age; }

 public void setAge(int age) { this.age = age; }

 public String getNom() { return nom; }

 public void setNom(String nom) { this.nom = nom; }

 public static int getCommunityNumber() {
 return NOMBRE_DE_PERSONNES;
 }

 public String toString() {
 return "Personne{" +
 "age=" + age +
 ", nom=" + nom + "\" +
 '}'
 }
}
```

# Références

- Le site web de J.-M. Doudoux : <http://www.imdoudoux.fr/java/dej/>
- Le cours « *Initiation à la programmation orientée-objet avec le langage Java* » de Gauthier PICARD et Laurent VERCOUTER