



Cours 1 : Rappels et Spring

Mardi 20 Octobre 2020

Antoine Flotte (aflotte@excilys.com)

Excilys
Développeurs de passion

Planning prévisionnel

- Tous les mardi de 9h à 12h45
- 7 TP (avec des cours en introductions quand nécessaire)
- Un mini projet

SOMMAIRE

- I. Rappels de Java
- II. Rappels d'architecture
- III. Rappels de Maven
- IV. Spring

Rappels 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 */`

- 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**

- 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**

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

- 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;

- 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
```

- 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);
```

- 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** incréméntation)
 - Incrémentation postfixe : `i++`
(incrémente `i` et retourne sa valeur **avant** incréméntation)
 - Décrémentation : `--`
 - Décrémentation préfixe : `--i`
(décrémente `i` et retourne sa valeur **après** décréméntation)
 - Décrémentation postfixe : `i--`
(décrémente `i` et retourne sa valeur **avant** décréméntation)

- 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"

Rappels de Java

Les structures conditionnelles et boucles

- **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**
 - 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;
```

- Souvent illisible (exemple affectation de j) donc on évite

- 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);
```

Rappels de Java

Concept de classes

- **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)

```
public      int    nbHabitantParAge      ( int age )      { ... }  
          |        |            |          |            |  
          accès   type   nom de la méthode  paramètre corps  
                  |  
                  de retour
```

- **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) { ... }

- **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);
```

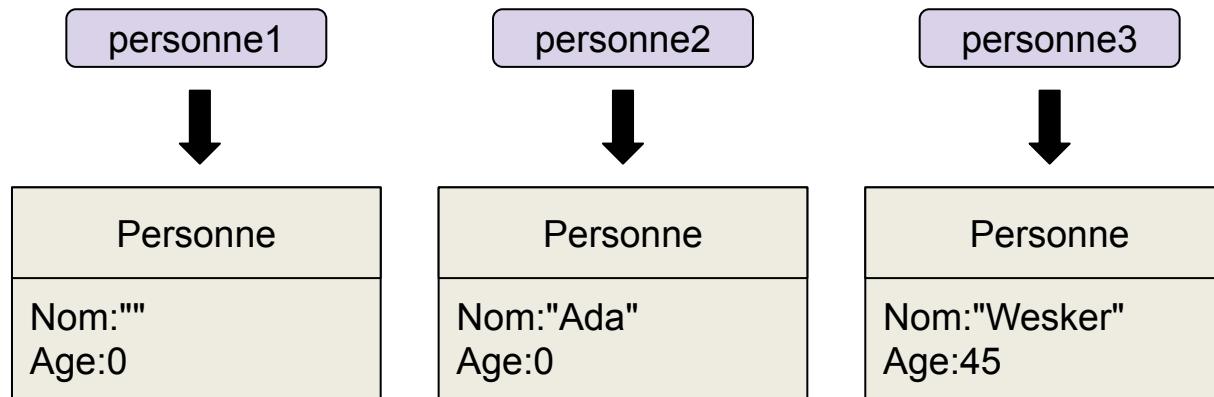


```
System.out.println(personne.toString());
```

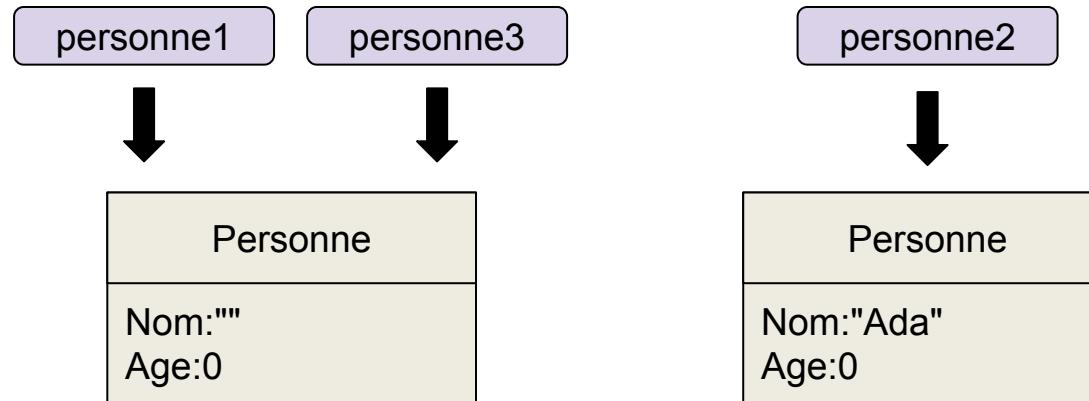
- Pour utiliser un objet, il faut d'abord le créer.
On obtient alors une instance de la classe.

```
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;  
    }  
}  
  
// On appelle la méthode directement avec la classe  
Personne.getCommunityNumber();  
// Ou avec une instance de la classe  
Personne personnelInst = new Personne(0, "");  
personnelInst.getCommunityNumber();
```

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

```
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

Rappels de Java

Exemple complet

Rappels de Java

```
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);  
    }  
}
```

Exemple complet

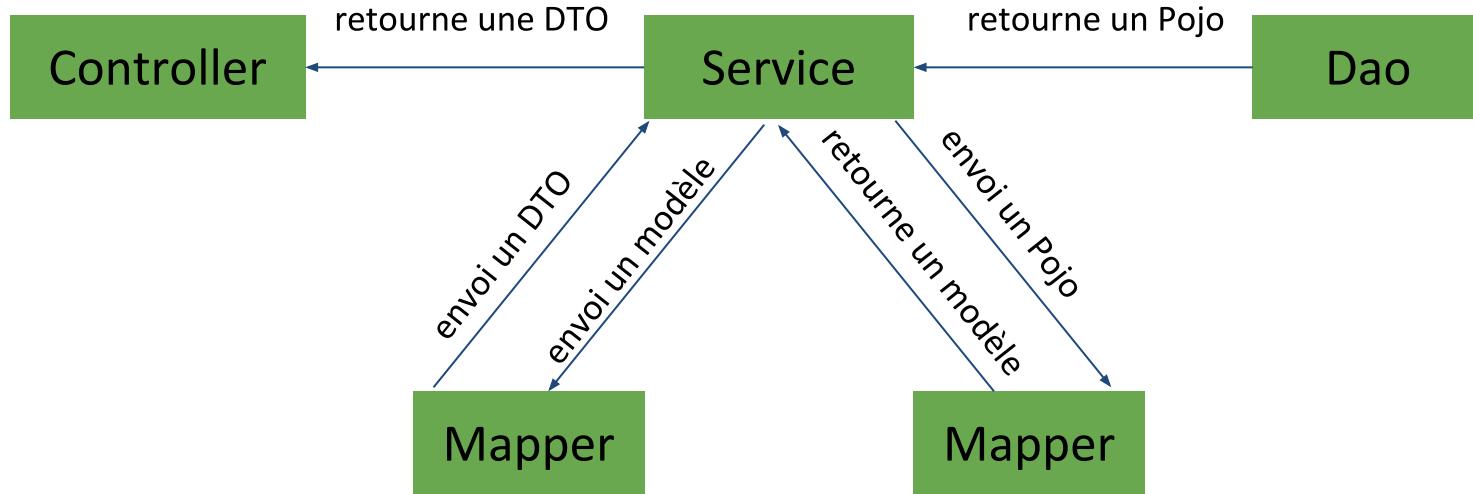
```
public int getAge() { return age; }  
  
public void setAge(int age) { this.age = age; }  
  
public String getNom() { return nom; }  
  
public void setNom(String nom) { this.nom = nom; }  
  
public static int getCommunityNumber() {  
    return NOMBRE_DE_PERSONNES;  
}  
  
public String toString() {  
    return "Personne{" +  
        "age=" + age +  
        ", nom=" + nom + "\\" +  
        '}';  
}
```

Architecture

- On a le **back**, c'est la partie Java dans laquelle on met l'**intelligence**.
- On a le **front**, c'est la partie qui s'exécute sur le pc du client. Il s'agit de faire de l'**affichage**, aucune intelligence. Le front appelle le back.

- L'année dernière on a vu le principe de l'architecture Controller - Service - DAO
- Ajoutons les objets qui leurs sont associés :
- DTO - Model - Pojo

- DTO : objet légers envoyé au front, ne contient que des objets Java simple et des listes d'objets Java simple
- Modèle : contient tout ce dont on a besoin dans le code
- Pojo : objet représentant le contenu de la base de donnée, ne contient que des objets Java simple, aucune liste



Rappels de Maven

- Maven repose sur l'utilisation de plusieurs concepts :
- Les artefacts : composants identifiés de manière unique
- Le principe de convention over configuration
- Le cycle de vie et les phases : les étapes de construction d'un projet sont standardisées
- Les dépôts (local et distant)

Répertoire	Contenu
/src	les sources du projet (répertoire qui doit être ajouté dans le gestionnaire de sources)
/src/main	les fichiers sources principaux
/src/main/java	le code source (sera compilé dans /target/classeses)
/src/main/resources	les fichiers de ressources (fichiers de configuration, images, ...). Le contenu de ce répertoire est copié dans target/classes pour être inclus dans l'artefact généré
/src/main/webapp	les fichiers de la webapp
/src/test	les fichiers pour les tests
/src/test/java	le code source des tests (sera compilé dans /target/test-classeses)
/src/test/resources	les fichiers de ressources pour les tests
/target	les fichiers générés pour les artefacts et les tests (ce répertoire ne doit pas être inclus dans le gestionnaire de sources)
/target/classes	les classes compilées
/target/test-classes	les classes compilées des tests unitaires
/target/site	site web contenant les rapports générés et des informations sur le projet
/pom.xml	le fichier POM de description du projet

- Le fichier **POM** (Project Object Model) contient la description du projet Maven.
- Le fichier POM doit être à la racine du répertoire du projet.
- Le tag racine est le tag `<project>`.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.ensta</groupId>
    <artifactId>cours-maven</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <description>Exemple de projet Maven</description>

    <dependencies>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.9</version>
        </dependency>
    </dependencies>
</project>
```

Un exemple de fichier pom.xml

- mvn clean install

Spring

- Spring est une alternative à JEE
- AOP : Programmation Orientée Aspect
- Injection de dépendance : les instances nécessaire vous sont donnée quand vous en avez besoin
- Inversion de contrôle : Spring contrôle le flux d'exécution
- On l'ajoute juste dans le POM de maven

- L'injection de dépendance peut se faire de 3 façon :
- @autowired sur un attribut
- @autowired sur un getter
- @autowired sur un constructeur
- Spring va chercher un bean qui correspond au type attendu

- Un bean est défini par annotation
- @Bean dans une classe de configuration (@Configuration)
- @Controller,@Service,@Repository
- Nécessaire pour l'injection de dépendance

- Une API REST donne des entrées à l'extérieur
- On a les verbes HTML : GET, POST, PUT, et autres qu'on ne verra pas ici
- PUT = idempotent donc souvent Update
- POST = souvent create

- `@PathVariable` (`http://localhost:8080/foos/abc/info`)
- `@RequestBody` (dans le body de la requête)
- `@RequestParam`
(`http://localhost:8080/api/foos?id=abc`)