TP — La mécanique Java, à la main

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-31

Table des matières

1 Objectifs et cadre de la séance 1
2 Connaître la machine du cours 2
3 Du source au bytecode 2
3.1 La classe la plus petite possible 2
3.2 `javac` : la première moitié du pari 3
4 Ouvrir la boîte : contrat, instructions, octets 3
4.1 Le contrat, puis les instructions 3
4.2 Le `.class` vu comme un fichier : les huit premiers octets 4
5 La JVM accueille le bytecode 5
5.1 Exécuter 5
5.2 Qui vous exécute, exactement ? 5
6 Échanger le bytecode, pas le source 6
7 Lire la javadoc (compétence transversale du cours) 6
8 Sceller la séance : premier commit 8
9 Ce que vous devez pouvoir expliquer · Livré 9

1 Objectifs et cadre de la séance

La leçon d'origines a posé le pari de 1995 : ne pas porter le compilateur sur chaque machine, porter la machine — la JVM — et figer entre les deux un artefact défini par une spécification, le bytecode. Cette séance en est le TP de vérification : compiler un programme minuscule, ouvrir le `.class` obtenu, l'exécuter sur la JVM de l'image du cours, le sceller par un premier commit.

Aucune installation n'intervient ici : le JDK est fourni par l'image du cours, et l'installation sur une machine personnelle relève de l'annexe de prérequis de la séquence d'environnement (`90_prerequis_environnement`). Vous n'utiliserez dans cette séance que les outils bruts du JDK.

À la fin du TP, vous devez être capable de :

1. **Compiler** un fichier `.java` en bytecode `.class` avec `javac`, et nommer les deux artefacts produits sur le disque.
2. **Lire** un `.class` : son contrat avec `javap`, ses instructions avec `javap -c`, ses premiers octets (nombre magique et version majeure).
3. **Exécuter** ce bytecode sur la JVM et identifier la machine qui l'accueille (distribution, système, architecture).
4. **Committer** ce travail dans un dépôt Git créé pour l'occasion, et retrouver la trace dans `git log --oneline`.
5. **Documenter** une classe et un paquetage au format javadoc, et générer leur documentation.

2 Connaître la machine du cours

Toutes les commandes s'exécutent dans le shell **bash de l'image du cours** — la création du source par heredoc et l'examen du `.class` avec `od` n'existent pas sous PowerShell ; seul l'équivalent d'écriture du source sur machine personnelle est traité dans l'annexe de prérequis de la séquence d'environnement (`90_prerequis_environnement`). Identifiez d'abord la machine qui répond aux commandes :

```
java --version
javac --version
```

La sortie attendue, sur l'image du cours, annonce `openjdk 25.0.4` et nomme la distribution répondante : Temurin, éditée par Eclipse Adoptium. Savoir quelle distribution répond est un réflexe : depuis que le JCP a adopté la spécification du noyau de la plateforme (JSR 337, Java SE 7), « Java » désigne une spécification que plusieurs implémentations distinctes mettent en œuvre.

Toutes les cellules supposent le shell positionné dans `$HOME/work/tp-mecanique` ; reprendre une session commence par `cd "$HOME/work/tp-mecanique"`.

Créez le répertoire de travail du TP (sources, bytecode, dépôt Git) :

```
mkdir -p "$HOME/work/tp-mecanique"
cd "$HOME/work/tp-mecanique"
```

3 Du source au bytecode

3.1 La classe la plus petite possible

```
cd "$HOME/work/tp-mecanique"
cat > Handwritten.java <<'EOF'
class Handwritten {
```

```
    static String greet(String name) {
        return "Bonjour " + name + " !";
    }

    public static void main(String[] args) {
        System.out.println(greet("l'atelier"));
    }
}
EOF
ls -l
```

Le fichier ne contient qu'une classe **sans paquetage** : les paquetages sont définis dans l'annexe de prérequis de la séquence d'environnement (`90_prerequis_environnement`) ; s'en affranchir limite le bytecode aux méthodes `greet` et `main` et garantit que chaque ligne du `.class` provient du geste effectué. La sortie attendue du `ls -l` ne montre qu'un fichier.

3.2 `javac` : la première moitié du pari

```
cd "$HOME/work/tp-mecanique"
javac Handwritten.java
ls -l
```

`javac` a produit `Handwritten.class` ; la sortie attendue du second `ls -l` montre deux fichiers, la source et le bytecode. Ce second fichier n'est pas un exécutable natif : un flux d'instructions et des tables de symboles définis par la spécification de la machine virtuelle, sans référence au processeur ni au système d'exploitation de la machine qui l'a vu naître. Le coût de « compiler une fois, exécuter partout » se paie ici : la moitié « une fois » du pari est cette traduction, effectuée une seule fois, en amont de toute exécution.

4 Ouvrir la boîte : contrat, instructions, octets

4.1 Le contrat, puis les instructions

```
javap Handwritten.class
```

Sans option, `javap` n'affiche que les signatures — le contrat compilé : ce que la classe expose, sans son implémentation ; la sortie attendue tient en trois lignes (la classe, `static java.lang.String greet(java.lang.String)`, `public static void main(java.lang.String[])`).

```
javap -c Handwritten.class
```

Avec `-c`, chaque corps de méthode devient une suite de mnémoniques : les instructions d'une machine à pile, exécutées par la JVM et non par votre processeur. Chaque instruction manipule une pile d'opérandes interne — `ldc` y pousse une constante, `invokestatic` appelle une méthode de classe en consommant ses arguments, `getstatic` y charge un champ statique tel que `System.out`, `invokevirtual` invoque une méthode d'instance comme `println`, `return` termine la méthode. Dans le corps de `main`, vous reconnaîtrez ces cinq mnémoniques : leur présence, non une instruction native, rend le fichier transportable. `javap` donne ainsi la possibilité de voir le contrat — l'idée fondatrice de la leçon d'origines, écrite en toutes lettres dans le bytecode.

4.2 Le `.class` vu comme un fichier : les huit premiers octets

```
od -A x -t x1 -N 8 Handwritten.class
```

La sortie attendue, sur l'image du cours, est une ligne de huit octets, `000000 ca fe ba be 00 00 00 45`, suivie de l'adresse de fin `000008`. Commentons : les quatre premiers octets disent « ceci est un class file » ; les deux suivants, `00 00`, sont la version mineure ; les deux derniers, `00 45`, sont la version **majeure** — `0x0045` vaut 69, la version de bytecode du JDK 25.0.4 de l'image. Cette valeur fixe la version minimale de JVM qui acceptera votre bytecode : le contrat de 1995 est inscrit en hexadécimal dans le fichier.

5 La JVM accueille le bytecode

ℹ Définition — JVM, JRE, JDK

- **JVM (machine virtuelle Java)** : l'exécuteur — elle charge une classe, la vérifie (le vérificateur contrôle la conformité des instructions à la spécification avant toute exécution), puis exécute le bytecode en s'appuyant sur le système hôte pour les entrées-sorties, la mémoire et les threads.
- **JRE (environnement d'exécution Java)** : la JVM accompagnée des bibliothèques nécessaires à l'exécution, sans outils de développement.
- **JDK (kit de développement Java)** : le contenu du JRE et les outils de ce TP — `javac`, `javap`, `javadoc`, `jar`, et `java`, le lanceur qui démarre une JVM.

Exemple ancré : `java Handwritten` lance une JVM Temurin qui charge et vérifie le bytecode vu en section 4 ; `javap`, lui, n'existe que dans le JDK.

Cas limite : avant la modularisation du JDK 9, la JRE se distribuait séparément (et alimentait les applets de navigateur) ; ce modèle a disparu et reste hors périmètre.

Source : Vue d'ensemble de l'installation du JDK, documentation Oracle, Java SE 25.

5.1 Exécuter

```
java Handwritten
```

La sortie attendue est le message écrit dans la source, `Bonjour l'atelier !`. `java Handwritten` se forme **sans** extension : vous nommez une **classe**, que la JVM cherche dans le chemin de classes (répertoire courant par défaut), vérifie, puis exécute depuis sa méthode `main`. Le porteur du `.class` portable est une machine virtuelle, non le système : la seconde moitié du pari de 1995 s'est exécutée dans cette commande.

5.2 Qui vous exécute, exactement ?

```
java -XshowSettings:properties -version 2>&1 \
  | grep -E 'os\.(name|arch)|java\.vm\.vendor|java\.version ='
```

Sur l'image du cours, la sortie attendue montre `java.vm.vendor` égal à `Eclipse Adoptium` et `java.version` égal à `25.0.4` ; `os.name` et `os.arch`, en revanche, dépendent de la machine hôte du conteneur et varient d'un poste à l'autre — c'est attendu, et cela fait partie de l'expérience. La symétrie avec la section 4 est complète : le bytecode ne porte aucune de ces propriétés — il ne connaît que la spécification. « Compiler une fois, exécuter partout » veut dire que la connaissance de la machine est entièrement regroupée dans la JVM : vous venez d'observer les deux côtés de la frontière.

6 Échanger le bytecode, pas le source

À faire

Récupérez le `Handwritten.class` compilé par un voisin (le binaire, pas le `.java`), placez-le dans un répertoire séparé (`mkdir -p echange`) et exécutez-le dans votre environnement **sans le recompiler** : `java -cp echange Handwritten`. À défaut de voisin physique (travail à distance), échangez le `.class` par le canal du cours — c'est le binaire qui compte, pas le moyen. Écrivez votre phrase — qu'est-ce qui a voyagé, qu'est-ce qui n'a pas voyagé ? — dans un fichier `b3.md`, que vous committerez à l'étape de clôture.

La sortie attendue est le même message que la première exécution de `java Handwritten`, `Bonjour l'atelier !`, obtenu d'un fichier que votre `javac` n'a jamais compilé : la portabilité n'est plus une affirmation héritée mais le résultat de votre propre exécution. Ce qui a voyagé est un artefact de la machine abstraite ; la JVM, elle, est restée locale.

7 Lire la javadoc (compétence transversale du cours)

La commande `javadoc` génère de la documentation HTML à partir de commentaires insérés dans le source entre `/** ... */`. Compétence transversale — la javadoc est la source normative des encadrés du cours — et consigne explicite : **dès cette séance, tout programme rendu est commenté au format javadoc.**

Définition — Commentaire javadoc

Un commentaire javadoc précède immédiatement l'élément qu'il documente (paquetage, classe, membre) : description, puis balises. La structure d'une page générée est close : page par paquetage, page par classe ou interface (sommaire puis détail des membres), page d'héritage, index général, fichiers d'information de génération. Les balises de ce cours sont `@author`, `@version`, `@param`, `@return`, `@since` ; la liste exhaustive et la position de chacune sont définies par la spécification. Le texte accepte quelques balises HTML (`<code>`, `<p>`) ; les listings plus bas en montrent l'usage. Un **paquetage** se documente par un fichier dédié, `package-info.java`, qui ne contient que le commentaire et la déclaration de paquetage : sans lui, la page du paquetage est vide.

Source : Java Documentation Comments Specification, Java SE 25.

`Handwritten` étant sans paquetage, on construit un mini-paquetage d'atelier de deux fichiers, plus un contre-exemple examiné en fin de section, dans le répertoire de la séance :

```
mkdir -p atelier
cat > atelier/Calculette.java <<'EOF'
package atelier;

/**
 * Calculette minimale de la séance mécanique.
 *
 * @author Prénom Nom
 * @version 1.0
 * @since 1.0
 */
public class Calculette {

    /**
     * Additionne deux entiers.
     *
     * @param a premier addend
     * @param b second addend
     * @return la somme de {@code a} et de {@code b}
     */
    public static int addition(int a, int b) {
        return a + b;
    }
}
EOF
cat > atelier/package-info.java <<'EOF'
/**
 * Paquetage d'atelier de la séance mécanique.
 * <p>
 * Support de la démonstration javadoc ; la notion de paquetage est définie
 * dans l'annexe de prérequis de la séquence d'environnement.
 * </p>
 *
 * @author Prénom Nom
 * @version 1.0
 * @since 1.0
 */
package atelier;
EOF
```

```
javadoc -d doc -sourcepath . atelier
```

Générez la documentation : la sortie attendue est double — sur la console, `javadoc` énumère les paquets traités sans erreur ; dans l'arborescence, `doc/index.html` (page d'entrée, à ouvrir dans le navigateur) et le sous-répertoire `atelier/` avec ses pages. Vérifiez que la description de `package-info.java` figure sur la page du paquetage et que `addition(int, int)` y porte ses `@param` et `@return`. Cas limite : un commentaire absent.

```
cat > atelier/SansCommentaire.java <<'EOF'
package atelier;

public class SansCommentaire {
    public int valeur;
}
EOF
javadoc -d doc -sourcepath . atelier
```

La sortie attendue change de nature : `javadoc` émet des avertissements et la page de `SansCommentaire` est dépourvue de description. Un document incomplet n'est pas une erreur de compilation — le programme fonctionnerait ; c'est une documentation absente là où le cours l'exige.

8 Sceller la séance : premier commit

Git, ici, est un rituel minimal ; la séquence Git du cours (plus loin dans la galerie) en prend la charge complète.

```
cd "$HOME/work/tp-mecanique"
git init
git config user.name  "Prénom Nom"              # à adapter
git config user.email "prenom.nom@exemple.fr"   # à adapter
git status --short
git add Handwritten.java b3.md atelier
git commit -m "Premier livré : source compilé, bytecode vérifié, JVM identifiée"
git log --oneline
```

La sortie attendue de `git log --oneline` est une ligne : identifiant court et message. Vous avez committé le **source** et `atelier/` documenté — pas le `.class`, pas `doc` : sauvegarder un artefact régénérable multiplierait les versions sans rien ajouter à l'information. (Ne rejouez ce bloc qu'après une modification : sinon Git répond `nothing to commit` — absence de nouveauté.)

Tout ce cours apprend à lire Java à travers ses traces — chaque particularité du langage est la trace d'un arbitrage. À partir de cette séance, votre travail laisse les siennes : datées, vérifiables, relues.

9 Ce que vous devez pouvoir expliquer · Livré

- Que contient un `.class` — et que ne contient-il **pas** ?
- Pourquoi `java Handwritten` se passe de l'extension `.class` ?
- À quoi servent le nombre magique `ca fe ba be` et la version majeure 69 (`0x0045`, produite par le JDK 25.0.4 de l'image du cours) ?
- Pourquoi le `.class` d'un voisin s'exécute-t-il sans recompiler ?
- Que promet — et que ne promet pas — « compiler une fois, exécuter partout » ?
- Quel est l'intérêt d'un commit dans une démarche de travail vérifiable ? (pris en charge par la séquence Git du cours)
- Comment documenter un paquetage pour que sa page générée ne soit pas vide ?

Livré

Le répertoire `$HOME/work/tp-mecanique` contient : `Handwritten.java`, le mini-paquetage `atelier/` (avec `package-info.java`), `b3.md`, le commit de séance visible dans `git log --oneline` ; `Handwritten.class` sur le disque, absent du dépôt, volontairement. Vérification : `git log --oneline` montre le commit de clôture ; `git status --short` montre les artefacts (`Handwritten.class`, `doc/`, `echange/`) en non suivis (`??`) et rien d'`atelier/` en attente.