FICHE · JAVA

Le ClassLoader Java

Comment la JVM trouve et charge vos classes et vos ressources

Frédéric Guyomarch · octobre 2026

Quand on lance un programme Java, la machine virtuelle (la JVM) doit retrouver, une à une, toutes les classes dont il a besoin : les siennes, celles des bibliothèques, et celles du JDK lui-même. Cette fiche explique comment elle procède. On part du fichier `.class` produit par le compilateur, on suit son chargement en mémoire, puis on détaille la recherche des classes à partir de leur nom complet et du classpath. On termine par les erreurs de chargement les plus courantes et par la lecture de fichiers de ressources, qui repose sur le même mécanisme.

Les exemples utilisent le paquet `fr.ulille.but` et supposent que l'on travaille en ligne de commande, sous Linux, macOS ou Windows.

1. Du code source à la classe en mémoire

1.1 Compiler une fois, exécuter partout

Le compilateur `javac` ne produit pas du code pour un processeur réel, mais du bytecode : un langage destiné à une machine virtuelle. Le fichier `.class` obtenu est donc portable, et le même fichier peut être copié d'un système à l'autre. Ce qui change, c'est la JVM : Linux, Windows et macOS ont chacun la leur, et chacune sait exécuter le même fichier `.class`.

Quand on tape `java Hello`, plusieurs composants de la JVM entrent en jeu : le ClassLoader, le vérificateur de bytecode, l'interpréteur accompagné du compilateur JIT, et le ramasse-miettes. Le ClassLoader est la porte d'entrée : rien ne s'exécute dans la JVM sans avoir d'abord été chargé par un ClassLoader.

Le chemin parcouru par une classe est donc le suivant : le fichier source `Hello.java` est compilé par `javac` en `Hello.class` ; le ClassLoader lit les octets de ce fichier et les transforme en un objet `Class<Hello>`, présent en mémoire, que la JVM sait exécuter.

1.2 À quoi ressemble un fichier .class ?

Un fichier `.class` est un fichier binaire. La commande `od` permet d'en afficher les premiers octets, et `javap -c` de le désassembler :

```
$ od -A x -t x1 -N 8 Hello.class
000000 ca fe ba be 00 00 00 41

$ javap -c Hello
  public static void main(java.lang.String[]);
    Code:
      0: getstatic     #7   // java/lang/System.out
      3: ldc           #13  // String Bonjour
      5: invokevirtual #15  // java/io/PrintStream.println
      8: return
```

Les quatre premiers octets, `CAFEBABE`, sont la signature de tout fichier `.class`. Les octets suivants donnent la version du format : `0x41` vaut 65 en décimal, ce qui correspond à Java 21. Le reste est constitué d'instructions destinées à la JVM, et non au processeur : récupérer `System.out`, charger la chaîne « Bonjour », appeler `println`, puis retourner.

Un détail est essentiel pour la suite : les autres classes, comme `java/lang/System` ou `java/io/PrintStream`, ne sont citées que par leur nom. La JVM devra donc les trouver et les charger à leur tour. Toute la question de cette fiche est là : à partir d'un nom, retrouver les octets.

1.3 Chaque classe chargée devient un objet Class

Une fois chargée, une classe est représentée par un unique objet de type `Class`. Il existe trois façons de l'obtenir, qui donnent toutes le même objet :

```
String s = "Bonjour";
Class<?> c1 = s.getClass();             // depuis une instance
Class<?> c2 = String.class;             // depuis le nom écrit dans le code
Class<?> c3 = Class.forName("java.lang.String");    // depuis une chaîne

c1 == c2 && c2 == c3      // true
c1.getName()              // "java.lang.String"
c1.getMethods().length    // une centaine
```

Cet objet décrit la classe : son nom, ses méthodes, ses champs, sa superclasse. C'est la base de la réflexion. Il connaît aussi son chargeur : `getClassLoader()` renvoie le ClassLoader qui l'a chargée.

2. Un chargement à la demande

La JVM ne charge pas toutes les classes au démarrage : une classe n'est chargée que la première fois que le programme en a réellement besoin. Prenons une classe `Main` qui n'utilise la classe `Outil` que si on lui passe un argument :

```
public class Main {
  public static void main(String[] args) {
    System.out.println("Début");
    if (args.length > 0) {
      new Outil().travailler();
    }
  }
}
```

L'option `-verbose:class` fait afficher chaque chargement. Sans argument, `Outil` n'apparaît jamais ; avec un argument, elle est chargée au moment où le code en a besoin :

```
$ java -verbose:class Main | grep Outil
# rien : Outil n'est jamais chargée

$ java -verbose:class Main x | grep Outil
[class,load] Outil source: file:/…
```

Ce fonctionnement apporte un démarrage plus rapide, une mémoire économisée, et une classe absente qui ne pose de problème que si on l'utilise vraiment.

La même option permet de mesurer l'ampleur du travail. Pour un simple programme qui affiche une ligne, la JVM charge environ 450 classes :

```
$ java -verbose:class Hello
[class,load] java.lang.Object source: shared objects file
[class,load] java.io.Serializable source: shared objects file
[class,load] java.lang.Comparable source: shared objects file
…
[class,load] Hello source: file:/home/etu/tp/
Bonjour

$ java -verbose:class Hello | grep -c class,load
455
```

Les classes du JDK passent en premier : `Object`, `String`, `System` et bien d'autres. La mention « shared objects file » indique qu'elles proviennent d'une archive préchargée, appelée CDS, qui accélère le démarrage. La classe de l'utilisateur, `Hello`, n'arrive que presque à la fin, lue depuis le dossier courant.

3. Organiser le code en paquets

Dès qu'un programme dépasse quelques fichiers, on range le code en paquets. Chaque paquet correspond à un dossier : le fichier `Main.java` du paquet `fr.ulille.but` se place dans `src/fr/ulille/but/`, et sa première ligne déclare ce paquet :

```
package fr.ulille.but;
```

Voici une organisation de travail classique, avec les sources dans `src/` et les classes compilées dans un dossier `out/` créé par `javac` :

```
tp/
├── src/fr/ulille/but/Main.java
├── src/fr/ulille/but/outils/Calcul.java
└── out/                        // créé par javac
```

L'option `-d out` de `javac` range les fichiers `.class` dans `out/` en recréant l'arborescence des paquets :

```
$ javac -d out src/fr/ulille/but/*.java \
                src/fr/ulille/but/outils/*.java
$ find out -name '*.class'
out/fr/ulille/but/Main.class
out/fr/ulille/but/outils/Calcul.class
```

On lance ensuite le programme en donnant à `java` le dossier où chercher (l'option `-cp`, détaillée plus bas) et le nom complet de la classe :

```
$ java -cp out fr.ulille.but.Main
```

Le nom complet est indispensable : pour la JVM, la classe s'appelle `fr.ulille.but.Main`, jamais `Main` tout court.

4. Le classpath : où chercher les classes

4.1 Principe

Le classpath est la liste des endroits où la JVM cherche les classes. On le fournit avec l'option `-cp` (ou `--class-path`). Il contient des dossiers et des archives `.jar` ; un `.jar` est une archive zip de fichiers `.class`. Les entrées sont séparées par « : » sous Linux et macOS, et par « ; » sous Windows.

```
$ java -cp out:lib/gson.jar fr.ulille.but.Main
```

Avec cette commande, pour charger `fr.ulille.but.Main`, la JVM regarde d'abord le fichier `out/fr/ulille/but/Main.class`, puis `fr/ulille/but/Main.class` dans l'archive `lib/gson.jar`. Les entrées sont lues dans l'ordre, et la première qui contient la classe l'emporte : si deux versions d'une même classe existent dans le classpath, c'est la première trouvée qui est utilisée.

4.2 Du nom de la classe au chemin du fichier

Le ClassLoader ne parcourt pas les dossiers pour deviner où se trouve une classe : il calcule un chemin à partir du nom, puis teste simplement s'il existe. Le calcul comporte deux étapes. Les points du nom complet deviennent des barres obliques, puis on ajoute l'extension `.class` :

```
fr.ulille.but.Main            // nom complet demandé
fr/ulille/but/Main            // les points deviennent des « / »
fr/ulille/but/Main.class      // on ajoute « .class »

// en code :
String chemin = nom.replace('.', '/') + ".class";
```

Le séparateur est la barre oblique « / » sur tous les systèmes, y compris Windows. Chaque classe a son propre fichier, y compris les classes imbriquées : `fr.ulille.but.Main$Interne` correspond à `fr/ulille/but/Main$Interne.class`.

4.3 Le parcours du classpath, entrée par entrée

Voyons la recherche complète sur un exemple. Le classpath contient un dossier et deux archives, et la JVM cherche la classe `com.google.gson.Gson`, c'est-à-dire le chemin `com/google/gson/Gson.class` :

```
$ java -cp out:lib/gson.jar:lib/outils.jar fr.ulille.but.Main
```

Entrée du classpath	Type	Chemin testé	Résultat
1. out/	dossier	`out/com/google/gson/Gson.class`	absent : entrée suivante
2. lib/gson.jar	archive	`com/google/gson/Gson.class`	trouvé : la classe est chargée
3. lib/outils.jar	archive	`(non consulté)`	jamais lu

Pour un dossier, la JVM vérifie simplement qu'un fichier existe à l'emplacement calculé. Pour une archive `.jar`, elle cherche le chemin comme nom d'entrée dans le zip, sans avoir besoin de décompresser l'archive. Dès qu'une entrée contient la classe, la recherche s'arrête : les entrées suivantes ne sont pas consultées. Si aucune entrée ne la contient, la recherche échoue et une `ClassNotFoundException` est levée.

4.4 Le paquet doit correspondre au dossier

Cette mécanique explique plusieurs erreurs classiques. Supposons que le fichier compilé se trouve dans `out/fr/ulille/but/Main.class` :

Commande	Fichier cherché	Résultat
`java -cp out fr.ulille.but.Main`	`out/fr/ulille/but/Main.class`	Trouvée et chargée
`java -cp out Main`	`out/Main.class`	Introuvable : il faut le nom complet
`java -cp out/fr fr.ulille.but.Main`	`out/fr/fr/ulille/but/Main.class`	Introuvable : mauvaise racine
`cd out/fr/ulille/but ; java Main`	`./Main.class (classpath par défaut)`	NoClassDefFoundError: Main (wrong name: fr/ulille/but/Main)

Le dernier cas est le plus subtil. Le classpath par défaut est le dossier courant, et le fichier `Main.class` y est bien trouvé ; mais il déclare appartenir au paquet `fr.ulille.but`, alors que la JVM le cherchait sous le nom `Main`. Elle refuse de le charger et signale un `wrong name`.

Deux règles en découlent. Le dossier indiqué dans le classpath est la racine : il ne fait pas partie du nom de paquet. Et le chemin sous lequel le fichier est trouvé doit toujours concorder avec le `package` déclaré à l'intérieur.

4.5 Les archives .jar

En pratique, on distribue rarement des dossiers de classes : on les empaquette dans un `.jar`. La commande `jar` construit l'archive à partir du dossier de classes et peut y indiquer la classe principale :

```
$ jar --create --file app.jar \
      --main-class fr.ulille.but.Main -C out .

$ jar --list --file app.jar
META-INF/MANIFEST.MF
fr/ulille/but/Main.class
fr/ulille/but/outils/Calcul.class

$ java -jar app.jar
# ou : java -cp app.jar fr.ulille.but.Main
```

Un `.jar` est donc un zip accompagné d'un manifeste, le fichier `META-INF/MANIFEST.MF`, qui désigne la classe principale par la ligne `Main-Class`. Attention : lorsqu'on lance `java -jar`, l'option `-cp` est ignorée ; le classpath provient alors de la ligne `Class-Path` du manifeste.

Les environnements de développement, Maven et Gradle construisent le classpath à la place du programmeur. En cas de doute sur ce qu'il contient, le plus simple est de regarder la commande `java` qu'ils lancent.

5. Quand le chargement échoue

Deux erreurs se ressemblent mais ne sont pas de même nature :

	ClassNotFoundException	NoClassDefFoundError
Nature	Exception vérifiée (checked)	Error, non prévue pour être rattrapée
Quand	On demande une classe par son nom : Class.forName, loadClass	La classe existait à la compilation mais manque, ou est inutilisable, à l'exécution
Cause typique	Nom mal orthographié, driver absent du classpath	.jar oublié au lancement, ou bloc static qui a échoué plus tôt
Réflexe	Vérifier le nom complet et le classpath	Chercher la première erreur, souvent une ExceptionInInitializerError

Pour provoquer une `NoClassDefFoundError`, il suffit de compiler deux classes dont l'une utilise l'autre, puis de supprimer le fichier de la seconde avant l'exécution :

```
// A.java
public class A {
  public static void main(String[] a) {
    new B().hello();
  }
}

// B.java
public class B { void hello() { … } }

$ javac A.java B.java && rm B.class
$ java A
Exception in thread "main" java.lang.NoClassDefFoundError: B
    at A.main(A.java:3)
Caused by: java.lang.ClassNotFoundException: B
    at …AppClassLoader.loadClass(ClassLoaders.java:188)
```

La classe `A` a été compilée avec `B` : à l'exécution, la JVM s'attend à la trouver. Le ClassLoader cherche `B.class` dans le classpath, échoue, et lève une `ClassNotFoundException`. La JVM l'enveloppe alors dans une `NoClassDefFoundError`. Il faut toujours lire la cause, c'est-à-dire la ligne « Caused by », qui indique ce qui s'est réellement passé.

6. Charger des ressources

6.1 Lire un fichier avec le ClassLoader

Le ClassLoader ne sert pas qu'aux classes : il permet aussi de lire des fichiers placés dans le classpath, comme un fichier de configuration ou une image. Dans un projet Maven ou Gradle, le dossier `src/main/resources` est copié dans le classpath :

```
src/main/resources/            // copié dans le classpath
├── config.properties
└── fr/ulille/but/logo.png

ClassLoader cl = Main.class.getClassLoader();
try (InputStream in =
       cl.getResourceAsStream("config.properties")) {
  if (in == null)
    throw new IllegalStateException("config absente");
  Properties p = new Properties();
  p.load(in);
}
```

Trois points sont à retenir. La recherche est la même que pour les classes : les dossiers et les `.jar` du classpath sont parcourus dans le même ordre. Cela fonctionne aussi une fois l'application empaquetée dans un `.jar`, alors que `new File("config.properties")` dépend du dossier courant et échoue dans ce cas. Enfin, si la ressource est absente, la méthode ne lève pas d'exception : elle renvoie `null`, ce qu'il faut tester.

6.2 Class ou ClassLoader : attention à la barre oblique

On peut chercher une ressource depuis un objet `Class` ou directement depuis un ClassLoader, et la barre oblique initiale n'a pas le même effet. Dans le tableau suivant, `Main` se trouve dans le paquet `fr.ulille.but` et `cl` est son ClassLoader :

Appel	Chemin cherché dans le classpath
`Main.class.getResourceAsStream("logo.png")`	fr/ulille/but/logo.png, relatif au paquet de Main
`Main.class.getResourceAsStream("/config.properties")`	config.properties, à la racine
`cl.getResourceAsStream("config.properties")`	config.properties : toujours depuis la racine
`cl.getResourceAsStream("/config.properties")`	Rien : la méthode renvoie null

`Class.getResourceAsStream` ajoute donc le nom du paquet pour un chemin relatif, puis délègue à son ClassLoader. Dans tous les cas, le séparateur est « / », même sous Windows.

6.3 Les bons réflexes

```
String aide;
try (var in = Main.class.getResourceAsStream("/aide.txt")) {
  aide = new String(Objects.requireNonNull(in, "aide.txt absent")
                        .readAllBytes(), StandardCharsets.UTF_8);
}

URL u = Main.class.getResource("/aide.txt");
// file:/…/out/aide.txt   ou   jar:file:/…/app.jar!/aide.txt
```

1. Préciser l'encodage : lire en UTF-8 de façon explicite, sinon le résultat dépend de la machine.
2. Ne pas utiliser `File` dans un `.jar` : `getResource` renvoie alors une URL de la forme `jar:file:…` ; il faut lire le flux, et ne pas la convertir en `File`.
3. Récupérer toutes les copies : `getResources(nom)` renvoie chaque fichier portant ce nom dans le classpath. C'est ainsi que `ServiceLoader` trouve ses implémentations.
4. Dans un framework, utiliser le ClassLoader du thread courant, obtenu par `Thread.currentThread().getContextClassLoader()`.

7. À retenir

1. `javac` produit du bytecode portable ; la JVM l'exécute sur chaque système.
2. Un ClassLoader lit les octets d'un fichier `.class` et en fait un objet `Class`.
3. Les classes sont chargées à la demande, au premier usage.
4. Le nom complet d'une classe donne le chemin de son fichier, par exemple `fr/ulille/but/Main.class`, cherché depuis chaque racine du classpath.

5. Le classpath, fait de dossiers et de `.jar`, est parcouru dans l'ordre ; la première classe trouvée l'emporte.
6. Les ressources suivent la même recherche ; `getResourceAsStream` renvoie `null` si le fichier est absent.
7. `NoClassDefFoundError` signale une classe connue à la compilation mais introuvable à l'exécution : on vérifie le classpath et l'on lit la cause.