FICHE · JAVA

Le ClassLoader Java

Comment la JVM trouve et charge vos classes et vos ressources.

Frédéric Guyomarch · octobre 2026

Compiler une fois, exécuter partout

Le .class est portable

javac produit du bytecode, un langage pour une machine virtuelle et non pour un processeur réel.

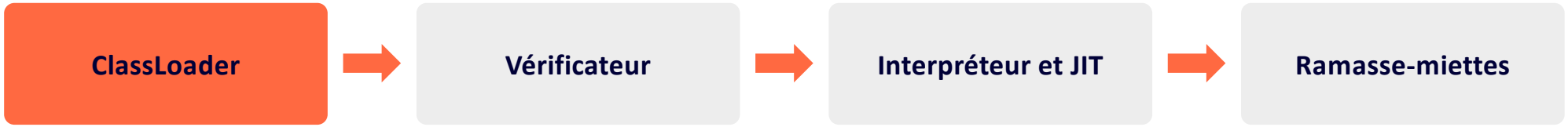
La JVM est propre à chaque système

Linux, Windows, macOS : chacun a sa JVM, qui exécute le même fichier .class.

Le ClassLoader est la porte d'entrée

Rien ne s'exécute dans la JVM sans avoir d'abord été chargé par un ClassLoader.

Ce que fait la JVM quand vous tapez **java Hello** :



Du fichier source à la classe en mémoire



Le ClassLoader lit des **octets** (le fichier .class) et les transforme en un objet **Class** que la JVM sait exécuter.

À quoi ressemble un fichier .class ?

```
$ 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
```

CAFEBABE : la signature de tout fichier .class.

0x41 = 65 : version du format, ici Java 21.

Des **instructions** pour la JVM, pas pour le processeur.

Les autres classes sont citées **par leur nom** (java/lang/System) : il faudra les trouver et les charger à leur tour.

Chaque classe chargée devient un objet Class

```
String s = "Bonjour";
Class<?> c1 = s.getClass();
Class<?> c2 = String.class;
Class<?> c3 = Class.forName("java.lang.String");

print(c1 == c2 && c2 == c3);    // true
print(c1.getName());            // java.lang.String
print(c1.getMethods().length);  // une centaine
print(Hello.class.getClassLoader());
// jdk.internal.loader.ClassLoaders$AppClassLoader@…
```

Un seul objet par classe

Trois façons de l'obtenir, toujours le même objet.

Il décrit la classe

Nom, méthodes, champs, superclasse : c'est la base de la réflexion.

Il connaît son loader

getClassLoader() dit quel ClassLoader l'a chargée.

Les classes sont chargées à la demande

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

```
$ java -verbose:class Main | grep Outil
# rien : Outil n'est jamais chargée
$ java -verbose:class Main x | grep Outil
[class,load] Outil source: file:/…
```

La JVM ne charge pas tout au démarrage.

Une classe est chargée la première fois que le programme en a réellement besoin.

Démarrage plus rapide, mémoire économisée, et une classe absente ne gêne que si on l'utilise.

Combien de classes pour un Hello World ?

```
$ 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
```

Environ **450 classes** chargées pour afficher une ligne.

Les classes du JDK passent en premier : Object, String, System…

« shared objects file » : une archive préchargée (CDS) qui accélère le démarrage.

Votre classe Hello arrive presque à la fin, lue depuis le dossier courant.

Paquets et dossiers : compiler proprement

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

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

package fr.ulille.but;

La ligne package fixe le dossier : fr/ulille/but/.

Nom complet

La JVM ne connaît que fr.ulille.but.Main, jamais Main tout court.

javac -d out

Range les .class dans out/, en recréant l'arborescence des paquets.

Le classpath : où chercher les classes

```
$ java -cp out:lib/gson.jar fr.ulille.but.Main
# fr.ulille.but.Main → cherche out/fr/ulille/but/Main.class
#                 puis fr/ulille/but/Main.class dans lib/gson.jar
```

Paquet = dossier

fr.ulille.but.Main se trouve dans fr/ulille/but/Main.class, à partir d'une racine du classpath.

Dossiers et .jar

Un .jar est une archive zip de .class. Séparateur « : » sous Linux et macOS, « ; » sous Windows.

Le premier gagne

Les entrées sont lues dans l'ordre. Deux versions d'une même classe : la première trouvée est utilisée.

Du nom de la classe au chemin du fichier

1 · Nom complet demandé

```
fr.ulille.but.Main
```

2 · Les points deviennent « / »

```
fr/ulille/but/Main
```

3 · On ajoute « .class »

```
fr/ulille/but/Main.class
```

```
// le ClassLoader ne parcourt pas les dossiers : il calcule un chemin
String chemin = nom.replace('.', '/') + ".class";
// "fr.ulille.but.Main" → "fr/ulille/but/Main.class"
```

Points → dossiers

Chaque point du nom de paquet devient un « / », y compris sous Windows.

Un fichier par classe

Une classe imbriquée a son fichier : Main$Interne donne Main$Interne.class.

Un simple calcul

La JVM ne parcourt pas les dossiers : elle déduit le chemin du nom, puis teste s'il existe.

Le parcours du classpath, entrée par entrée

```
$ java -cp out:lib/gson.jar:lib/outils.jar fr.ulille.but.Main
# classe demandée : com.google.gson.Gson → com/google/gson/Gson.class
```

Entrée du classpath	Type	Fichier cherché	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

Dossier

Un simple test : le fichier out/chemin existe-t-il ?

Archive .jar

Le chemin est cherché comme nom d'entrée dans l'archive zip, sans la décompresser.

Introuvable

Aucune entrée ne contient le fichier : ClassNotFoundException.

Le paquet doit correspondre au dossier

Commande (fichier : out/fr/ulille/but/Main.class)	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)

Racine + paquet = chemin

Le dossier du classpath est la racine : il ne fait pas partie du nom de paquet.

Les deux doivent concorder

Le fichier trouvé doit déclarer le même package que son chemin, sinon « wrong name ».

Le .jar : des classes dans une archive

```
$ 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 zip et un manifeste

META-INF/MANIFEST.MF indique la classe principale (Main-Class).

java -jar ignore -cp

Le classpath vient alors du manifeste, ligne Class-Path.

IDE, Maven, Gradle

Ils construisent le classpath pour vous : regardez la commande que lance l'IDE.

Deux erreurs à ne pas confondre

	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 cassée 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

Démo : provoquer une NoClassDefFoundError

```
// 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)
```

A a été compilée avec B : à l'exécution, la JVM s'attend à trouver B.

Le ClassLoader cherche B.class dans le classpath, échoue, et lève une **ClassNotFoundException**.

La JVM l'enveloppe dans une **NoClassDefFoundError** : lire toujours la cause, le « Caused by ».

Charger un fichier avec le ClassLoader

```
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);
}
```

Même recherche que les classes

Les dossiers et .jar du classpath sont parcourus dans le même ordre.

Marche aussi dans un .jar

new File("config.properties") dépend du dossier courant et échoue une fois l'application empaquetée.

null si absente

Pas d'exception : il faut tester.

Class ou ClassLoader : attention au « / »

Appel (Main est dans le paquet fr.ulille.but)	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, renvoie null**

Class.getResourceAsStream ajoute le paquet puis délègue à son ClassLoader. Le séparateur est toujours « / », même sous Windows.

Ressources : 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
```

Préciser l'encodage

UTF-8 explicite, sinon le résultat dépend de la machine.

Pas de File dans un .jar

getResource renvoie une URL jar: : lire le flux, ne pas convertir en File.

Toutes les copies

getResources(nom) renvoie chaque fichier du même nom : c'est ainsi que ServiceLoader trouve ses implémentations.

Quel loader ?

Dans un framework, utilisez celui du thread : getContextClassLoader().

À RETENIR

Conclusion

1. javac produit du bytecode portable ; la JVM l'exécute sur chaque système.
2. Un ClassLoader lit les octets d'un .class et en fait un objet Class.
3. Les classes sont chargées à la demande, au premier usage.
4. Le nom complet donne le chemin fr/ulille/but/Main.class, cherché depuis chaque racine.
5. Le classpath (dossiers et .jar) est parcouru dans l'ordre ; la première classe trouvée gagne.
6. Les ressources suivent la même recherche ; getResourceAsStream renvoie null si absent.
7. NoClassDefFoundError : classe connue à la compilation mais introuvable à l'exécution.

Pour toute question, un mail !

frederic.guyomarch@univ-lille.fr