Java : d'où ça vient, pourquoi c'est comme ça

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

2026-10-31

Table des matières

1 1991 : le projet Green et l'électronique embarquée 1
2 Concevoir contre C++ : des arbitrages, pas des excentricités 2
3 1995 et le pari de la JVM 3
4 Sunrise, Oracle, 2010 : la guerre des licences 3
5 Le train des versions 4
6 Le cadre technique qui en découle 5
7 Ce que vous retiendrez 5

Bien des aspects de Java surprennent quand on les rencontre sans contexte : l'absence de surcharge d'opérateurs, l'héritage multiple interdit, les exceptions que le compilateur vous impose de traiter, ou encore ce réflexe de tout faire passer par une machine virtuelle. Cette leçon soutient une thèse simple : rien de tout cela n'est une excentricité. Chaque particularité est la trace d'un arbitrage posé au début des années 1990, dans des contraintes précises. Comprendre ces arbitrages, c'est déjà apprendre à lire le langage.

1 1991 : le projet Green et l'électronique embarquée

Java naît chez Sun Microsystems, sous l'impulsion de James Gosling et de son équipe. Les origines du langage remontent à 1991 et au projet « Green », dont l'objectif était de développer une technologie pour les appareils électroménagers intelligents. L'équipe constate très vite que les langages existants — C et C++ — sont inadaptés à cette cible : les appareils visés ont des capacités matérielles hétérogènes et limitées, sont destinés à être connectés au réseau, et doivent fonctionner des années sans maintenance logicielle. Un langage dont la fiabilité dépend de la rigueur manuelle du programmeur dans la gestion de la mémoire n'y suffit pas.

De cette origine embarquée vient la matrice de toutes les décisions que nous allons examiner : Java n'a pas été conçu pour un système que le programmeur contrôle, mais pour une ferme de machines que personne ne contrôle entièrement. La porta-

bilité, la robustesse et la sécurité ne sont pas des fonctionnalités ajoutées après coup : ce sont les exigences fondatrices. Le langage est officiellement lancé en 1995, et c'est sa capacité à fonctionner sur n'importe quel dispositif grâce à la machine virtuelle qui en fera, dans les années qui suivent, un outil central du développement web et des applications distribuées.

2 Concevoir contre C++ : des arbitrages, pas des excentricités

L'équipe de Green a pratiqué C++ et vu ses fragilités sur le terrain. Le langage qui deviendra Java est délibérément conçu contre C++, avec un objectif affiché : simplifier le développement en éliminant les aspects complexes et sources d'erreurs. Quatre choix méritent d'être présentés non comme des interdits arbitraires, mais comme des échanges — chaque renoncement achète une garantie, et chaque garantie a un prix.

Premier arbitrage : la surcharge d'opérateurs, abandonnée. En C++, on peut redéfinir `+` ou `=` pour ses propres types ; le même symbole peut alors signifier des choses très différentes selon le contexte, et un code formellement correct devient sémantiquement imprévisible. Java renonce à ce pouvoir : une opération s'exprime par une méthode nommée, point. Le prix payé est l'absence de notation infixée pour les types utilisateur ; ce qui est gagné, c'est qu'un appel se lit toujours de la même façon. Nous verrons, en manipulant les collections, à quoi ressemble ce style « tout est méthode » dans la pratique.

Deuxième arbitrage : l'héritage multiple de classes, refusé. Quand une classe hérite de deux ancêtres qui se recouvrent, la résolution des conflits — le problème classique du diamant — est une source durable de bogues subtils. Java tranche : un seul héritage de classe, mais la possibilité de réaliser plusieurs interfaces — c'est-à-dire d'appartenir à plusieurs types sans hériter deux fois d'une implémentation. Ce choix réoriente tout le style de conception Java : plutôt que de composer des hiérarchies complexes, on assemble des objets. La maxime « préférer la composition à l'héritage » n'est pas une mode : cet arbitrage est l'un des fondements historiques de ce principe.

Troisième arbitrage : la mémoire gérée par le runtime. En C++, le programmeur alloue et libère ; l'oublier produit une fuite, le faire deux fois ou trop tôt produit un pointeur pendillant (dangling pointer) : accès à une mémoire déjà libérée — famille d'erreurs structurellement ingérables à l'échelle d'un système réparti. Java confie la libération de la mémoire à un ramasse-miettes (garbage collector) intégré à l'exécution. Le prix payé : l'exécution doit connaître le graphe des objets, les libérations ne sont pas prévisibles à la ligne près, et certaines pauses accompagnent le nettoyage. Ce qui est gagné : la disparition pure et simple de toute une catégorie de

bogues — les erreurs de libération. Vous le vivrez dès les premières heures de code, sans jamais le remarquer — c'est le signe que l'arbitrage fonctionne.

Quatrième arbitrage : les exceptions checkées. Java distingue les erreurs qu'un appelant peut raisonnablement traiter (elles doivent être déclarées dans la signature de la méthode et soit capturées, soit propagées) des erreurs de programmation, laissées libres. C'est le choix le plus discuté du langage : sa verbosité agace, et beaucoup de frameworks modernes s'en accommodent autrement. Mais la logique est celle du projet Green : dans un système où des sous-programmes écrits par des inconnus s'appellent à distance, « cette méthode peut échouer ainsi » fait partie du contrat, et le compilateur refuse qu'on ignore le contrat. Nous y reviendrons longuement au moment de connecter une base de données, là où ce choix devient tangible.

3 1995 et le pari de la JVM

La réponse de Java à la hétérogénéité des machines ne consiste pas à compiler pour chaque plateforme — ce qui multiplierait les chaînes d'outils et les versions à maintenir. Elle consiste en un pari d'architecture : insérer une représentation intermédiaire et une machine abstraite. Le compilateur `javac` traduit les fichiers `.java` en bytecode (fichiers `.class`), un pseudo-code défini par une spécification, compréhensible par la machine virtuelle Java (JVM). C'est la JVM qui est réimplémentée pour chaque système (Windows, Unix, macOS), une seule fois chacun ; le bytecode, lui, est produit une seule fois pour tous. D'où la formule consacrée : « compiler une fois, exécuter partout ».

Ce pari a un prix : le programme n'est pas exécuté directement par le processeur. La JVM interprète le bytecode, et rattrape la performance par une compilation à chaud (just-in-time, JIT) qui traduit les portions de code chaud en instructions natives pendant l'exécution ; des compilations anticipées (AOT) existent également. Le gain, lui, dépasse la portabilité : la JVM est un point de contrôle unique qui vérifie le bytecode à son chargement, borne la sécurité d'exécution, et offre un sol commun à d'autres langages. Surtout, ce pari explique la forme concrète de vos premiers TP : lorsque vous compilerez puis exécuterez un programme, vous manipulerez les deux objets de 1995 — le `.class` portable et la machine qui l'accueille — et c'est la leçon de mécanique du cours qui vous les fera toucher du doigt.

4 Sunrise, Oracle, 2010 : la guerre des licences

Pendant près de vingt ans, Java est la propriété de Sun Microsystems. En 2006, Sun ouvre son implémentation sous licence GPL : c'est le projet OpenJDK, qui devient la base commune de toutes les implémentations libres. En 2010, Oracle rachète Sun Microsystems — l'accord était préparé en interne sous le nom de code « Sunrise »

— et acquiert de ce fait Java. La situation devient ambiguë : le code source est ouvert (OpenJDK), mais les binaires distribués par Oracle relèvent d'une licence propriétaire, et les conditions d'utilisation commerciale se durcissent par paliers.

Cette incertitude provoque une mobilisation de l'écosystème : le projet AdoptOpenJDK, devenu **Eclipse Temurin**, fournit des binaires libres construits depuis OpenJDK, avec une licence permissive en termes d'utilisation et de distribution — c'est la distribution que ce cours recommande. Dans le même mouvement, les API historiques de Java EE passent à l'Eclipse Foundation, et leur espace de noms est renommé de `javax.*` à `jakarta.*`, Oracle détenant les marques liées à `javax`.

Deux conséquences pratiques à retenir dès aujourd'hui. D'abord, « installer Java » ne désigne plus un téléchargement unique mais une distribution parmi plusieurs, toutes conformes à la même spécification. Ensuite, la licence n'est pas un détail juridique lointain : c'est un risque industriel qui a failli fragmenter le langage, et nous le retrouverons dans la leçon d'histoire 2014–2025.

5 Le train des versions

Jusqu'au milieu des années 2010, Java avançait par grandes versions espacées, au point que le langage donnait l'impression de stagner. Depuis 2017, le rythme est radicalement différent : une version sort **tous les six mois**, en mars et en septembre, qu'elle soit riche ou modeste. Les fonctionnalités ne sont plus promises à une grande version lointaine ; elles montent dans le train au fil de leur achèvement — et celles qui ne sont pas prêtes passent au train suivant. Les versions intermédiaires non pérennes (par exemple Java 22 en mars 2024, Java 23 en septembre 2024) ne sont maintenues que six mois.

Certaines versions sont désignées **LTS** (Long-Term Support) et reçoivent un support étendu : Java 8 (mars 2014), Java 11 (septembre 2018), Java 17 (septembre 2021), Java 21 (septembre 2023, supporté jusqu'en 2028) et Java 25 (septembre 2025, supporté jusqu'en 2030). Pour un projet industriel, choisir une LTS est un calcul rationnel : on s'achète une fenêtre de correctifs de sécurité longue, une chaîne d'outils et de dépendances stabilisée, et l'on évite de migrer tout l'édifice tous les six mois. C'est pourquoi une part considérable du parc mondial tourne encore sur Java 8 ou 11 — d'où la réputation de retard du langage, d'ailleurs largement périmée : la langue bouge vite désormais, ce sont les déploiements qui restent prudents.

Ce régime explique un phénomène que vous observerez dans tous les stages et tous les entrepôts : le décalage constant entre la dernière version disponible et celle qu'un projet ose employer. Notre cours fait le choix de la dernière LTS en date — Java 25 — pour vous exposer aux fonctionnalités modernes sans quitter la situation réelle d'un projet professionnel.

6 Le cadre technique qui en découle

Tout ce vocabulaire d'origine se matérialise en trois boîtes emboîtées. La **JVM** est la machine virtuelle qui exécute le bytecode. Le **JRE** (Java Runtime Environment) réunissait la JVM et les bibliothèques standard pour exécuter des programmes Java. Le **JDK** (Java Development Kit) ajoute le compilateur `javac` et les outils de développement : c'est lui qu'il vous faut. Pour le développement comme pour l'exécution, un JDK suffit désormais — la distinction JRE/JDK s'est estompée dans les distributions modernes, mais les trois sigles restent dans le langage courant.

Comme il existe plusieurs distributions d'un même OpenJDK (voir la section précédente), la question « quelle version tourne ici ? » se pose toujours en deux exemplaires : la version du langage et la marque de la distribution. Le réflexe professionnel est de le vérifier systématiquement :

```
java --version    # version de la machine virtuelle
javac --version   # version du compilateur
```

L'installation guidée — le JDK **Eclipse Temurin**, **SDKMan** et la variable `JAVA_HOME` — est le premier geste de la séance « Mise en œuvre de Java ».

Le choix de l'éditeur ou de l'IDE est sans effet sur le contenu de ce cours ; la seule exigence de base est un JDK valide. Retenez la chaîne complète, elle est celle de 1995 revisitée par 2010 : une spécification, plusieurs distributions, un train de versions, et un outil — SDKMan — pour domestiquer tout cela sur votre machine.

7 Ce que vous retiendrez

Une seule idée, en fait : la syntaxe de Java est un document historique. Les contraintes de l'embarqué de 1991 ont produit les quatre grands arbitrages ; le pari de la JVM en 1995 a produit le bytecode et la JIT ; le rachat d'Oracle en 2010 a produit Temurin et les questions de licence ; le train de versions depuis 2017 a produit le dilemme LTS. La table ci-dessous vous donne, pour les principaux faits d'origine, le moment précis du cours où vous le ressentirez dans vos propres doigts — car l'histoire de Java n'est utile que si elle éclaire le code que vous écrirez.

Fait d'origine	Choix qui en résulte	Où vous le vivrez dans le cours
La JVM et le bytecode (1995)	compilation en `.class`, exécution par la machine virtuelle	TP mécanique
Le ramasse-miettes (arbitrage Green)	vous ne libérez jamais la mémoire manuellement	Bases

Fait d'origine	Choix qui en résulte	Où vous le vivrez dans le cours
Exceptions checkées (contrat d'appel)	le compilateur exige que vous traitiez l'échec annoncé	JDBC+exceptions
Pas d'héritage multiple de classes	la composition devient le style par défaut	Réutilisation/composition
Le train des versions	`record` et les syntaxes récentes n'existaient pas à Java 8	Bases+record
Guerres de licences et distributions	Temurin, SDKMan, OpenJDK : « Java » n'est pas un téléchargement	Histoire 2014-25