Chapitre 1

Origines du langage

1.1 *The network is the computer*

Java est un langage de programmation orienté objet développé par la société *Sun Microsystems* et adapté au développement d'applications distribuées. Le langage et son outillage favorisent *l'abstraction* de la répartition des ressources utiles au développement. Ce principe permet de considérer un *réseau* comme une unique machine programmable : *the network is the computer* [17].

1.2 *Write once, run everywhere*

Le langage a été conçu pour être *compilé* vers un langage machine intermédiaire appelé *bytecode*, exécutable par une *machine virtuelle*. Une machine virtuelle est spécifique à une plateforme d'exécution (*e.g.* PC Linux, Mac OS), mais chaque machine virtuelle est supposée pouvoir exécuter de la même façon n'importe quel *bytecode*. Ce mécanisme favorise donc la *portabilité* d'un programme (*i.e.* sa capacité à fonctionner dans différents environnements d'exécution) selon le principe *write once, run everywhere* [29]. Une fois compilé, un programme Java est supposé fonctionner indifféremment dans n'importe quel environnement d'exécution disposant d'une machine virtuelle.

1.3 Les racines du langage

Le langage a hérité de plusieurs caractéristiques de langages existants :

Smalltalk : références d'objet, polymorphisme, liaison dynamique.

Ada : packages, exceptions.

C++ : types primitifs, types génériques, syntaxe.

1.4 Historique

1.4.1 Naissance

En 1991, Naughton, Gosling et Sheridan, des employés de *Sun*, s'associent pour créer une alternative à C++ nommée *Oak* [7]. Il s'agit d'un langage orienté objet, simple d'utilisation, fortement typé, sécurisé, multitâche, indépendant d'une quelconque plateforme d'exécution et adapté au développement d'applications distribuées. En mars 1993, l'équipe fonde la société *FirstPerson*. En juin de cette même année, le navigateur *Mosaic* est lancé par le centre de recherche américain NCSA (*National Center for Supercomputing Applications*). Le Web, créé en 1989, commence à s'imposer sur le réseau. En août, *Sun* sous-traite à *FirstPerson* la tâche d'adapter Oak à internet. En 1995, Oak devient officiellement Java. En janvier 1996, le navigateur *Netscape 2.0* qui démocratisera l'usage du Web prend en charge les *applets* Java en intégrant une machine virtuelle.

1.4.2 Succès

Java s'est rapidement imposé pour de nombreuses raisons :

la portabilité : le *bytecode* exécutable par n'importe quelle machine virtuelle permet d'éviter de maintenir plusieurs versions d'un même programme en fonction des plateformes d'exécution cibles.

la sûreté : le langage est fortement typé et de nombreuses vérifications garantissent le bon fonctionnement de l'application au cours de son exécution.

l'adaptation à internet grâce au chargement dynamique de classes par le réseau ; à la notion d'*applet* ; et aux mécanismes de distribution des traitements entre plusieurs machines comme RMI (*Remote Method Invocation*) ou Corba (*Common Object Request Broker Architecture*).

l'assimilabilité par les développeurs C/C++ grâce à la syntaxe commune et aux types primitifs usuels (au prix d'une entorse au dogme du *tout objet*).

la richesse de l'API (*Application Programming Interface*) : les très nombreuses classes fournies en standard et structurées en *packages* constituent une base solide et cohérente pour le développement rapide d'applications (multimédia, interfaces graphiques, connectivité réseau, bases de données, etc.).

Chapitre 2

Outillage et documentation

La création d'un programme Java nécessite de disposer au minimum d'un *compilateur* chargé de transformer des fichiers de code source en fichiers de code exécutable, et d'une *machine virtuelle* chargée d'interpréter le code exécutable produit par le compilateur. Le compilateur comme la machine virtuelle sont des *programmes* qui s'exécutent dans un certain *environnement d'exécution*.

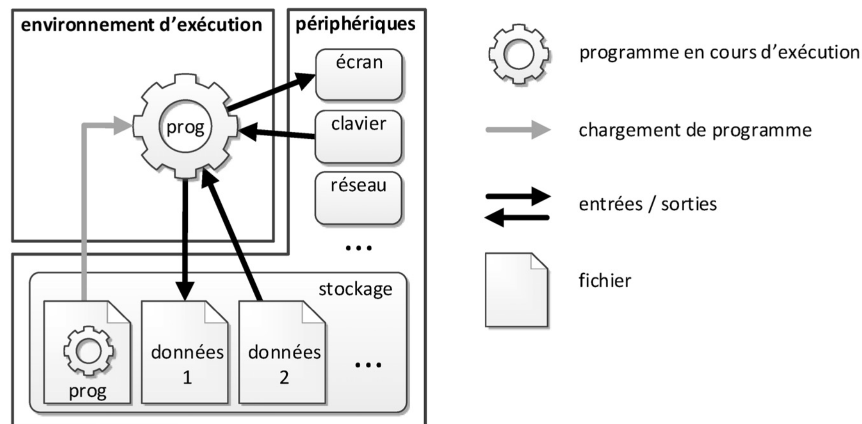

Figure 2.1 – Chargement, exécution d'un programme et entrées/sorties

L'environnement d'exécution contient en particulier un *espace de mémoire vive*. Le programme en cours d'exécution peut avoir accès aux différents *périphériques* de la machine sur laquelle il s'exécute. Il s'agit par exemple d'un *écran*, d'un *clavier*, d'une *connexion réseau* ou encore d'un *espace de stockage* dans lequel se trouvent des *fichiers*. On appelle *entrées/sorties* les échanges

entre le programme en cours d'exécution et les différents périphériques auxquels il a accès.

Un programme en cours d'exécution provient d'un code exécutable stocké dans un fichier. La première opération pour exécuter un programme consiste donc à *charger* le fichier correspondant dans l'environnement d'exécution.

Toutes ces notions sont illustrées dans la figure 2.1 et sont utilisées par la suite pour décrire notamment le fonctionnement du compilateur Java, de la machine virtuelle Java (JVM) et des programmes Java compilés et exécutables.

2.1 Compilateur

2.1.1 Production de code exécutable

Un programme Java est *compilé*, *i.e.* il est transformé en une série d'instructions en langage machine exécutable. Les figures 2.2 et 2.3 illustrent ce principe. Le compilateur charge un fichier de code source, l'analyse, et produit le code machine correspondant s'il ne contient aucune erreur.

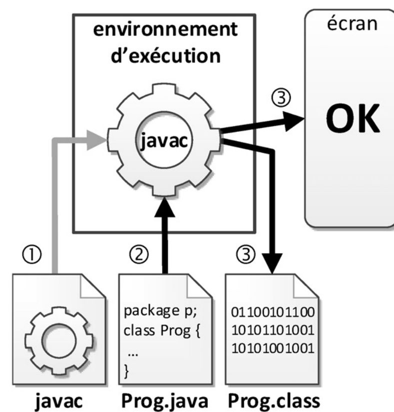

FIGURE 2.2 – Compilation réussie

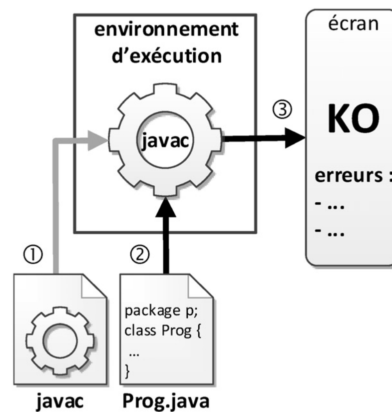

FIGURE 2.3 – Compilation échouée

Selon le compilateur employé, le code produit ainsi peut être du *code natif*, *i.e.* du code composé d'instructions spécifiques à un processeur. Le code natif est donc lié à une famille particulière de processeurs partageant le même jeu d'instructions.

Mais plus généralement en ce qui concerne Java, le code produit par un compilateur est du *bytecode* exécutable par une *machine virtuelle*. Le code ainsi produit n'est donc pas lié à un quelconque jeu d'instructions spécifiques à un certain processeur.

2.1.2 Vérifications et optimisations

Avant de produire le code, le compilateur fait un certain nombre de vérifications syntaxiques et sémantiques (notamment liées au *typage* des données). Par exemple, il refuse les programmes contenant des instructions syntaxiquement incorrectes (*e.g.* absence de point-virgule à la fin d'une affectation). Il détecte et refuse également le *code mort*, *i.e.* du code qui ne peut pas être exécuté comme après l'instruction `return`. Il détecte et refuse aussi les incohérences de type comme une valeur numérique affectée à une variable booléenne.

Lorsque le programme fourni passe avec succès toutes les vérifications syntaxiques et sémantiques, le compilateur effectue quelques *optimisations* puis il produit le code cible (typiquement du *bytecode*, ou éventuellement du code natif).

2.1.3 Directives de compilation

Il est possible de fournir avec un programme des *directives* au compilateur, *i.e.* des instructions spécifiquement destinées au compilateur afin d'adapter son action. Par exemple, il est possible d'indiquer au compilateur de ne pas générer d'avertissements liés à une certaine portion de code source. Ces directives sont appelées *annotations* et sont abordées dans le chapitre 12.

2.1.4 Compilateurs Java usuels

Un compilateur Java doit respecter la *spécification* du langage [11] qu'il prend en charge. Le compilateur fourni gratuitement par *Oracle* (qui a racheté *Sun* en 2010) respecte la spécification du langage et produit du *bytecode*.

Le compilateur GCJ *GNU Compiler for Java* [31] est un autre exemple de compilateur Java. Il fait partie de la collection des compilateurs GNU qui prennent notamment en charge C/C++, Fortran et Pascal. Ce compilateur peut ainsi tirer partie du *back-end* commun aux autres compilateur GNU afin de générer du code natif.

2.2 Machine virtuelle

2.2.1 Chargement de *bytecode*

Le *bytecode* Java produit par le compilateur est conforme aux spécifications de la machine virtuelle Java, ou JVM (*Java Virtual Machine*), éditées par *Oracle* [21].

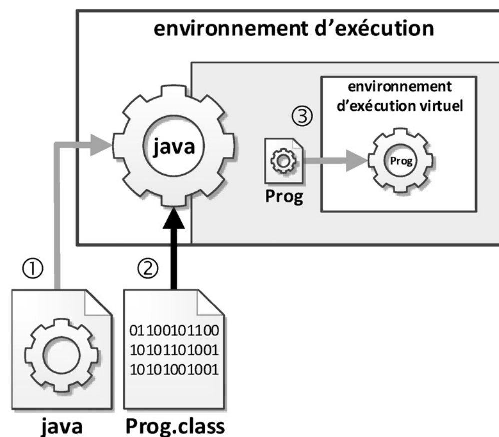

FIGURE 2.4 – Exécution de *bytecode* par une machine virtuelle

Comme illustré par la figure 2.4, une machine virtuelle est un programme qui simule le fonctionnement d'une machine qui dispose d'un *environnement d'exécution virtuel*, notamment muni de son propre espace mémoire. Le *bytecode* produit par le compilateur n'est pas directement exécutable dans l'environnement d'exécution natif. Pour exécuter un programme Java compilé, la première opération consiste donc à charger la machine virtuelle Java (JVM) dans l'environnement d'exécution natif. La deuxième opération est effectuée par la machine virtuelle et consiste à lire le fichier de *bytecode*. La troisième opération est également effectuée par la machine virtuelle et consiste à charger le programme Java dans l'environnement d'exécution virtuel.

Une fois chargé par la machine virtuelle, le programme Java peut être exécuté. Toutes les opérations d'entrées/sorties (*e.g.* vers l'écran, depuis le clavier, sur des connexions réseaux, sur l'espace de stockage, etc.) s'effectuent par l'intermédiaire de la machine virtuelle. Elle dispose également de son propre *ordonnanceur* qui lui permet de gérer l'exécution concurrente de différentes *tâches* (*threads*) à l'intérieur même de l'environnement d'exécution virtuel.

2.2.2 Portabilité des programmes Java

Le principe de portabilité mis en œuvre par une JVM est illustré par les figures 2.5 et 2.6. La première figure représente un programme écrit en C. Pour pouvoir être exécuté sur une plateforme Linux, ce programme devra être compilé en code natif pour cette plateforme cible. Pour pouvoir être exécuté sur une autre plateforme (*e.g.* Windows), il devra être *recompilé*. Selon les cas, il devra d'abord être *adapté*. En effet, des différences dans l'environnement d'exécution pourront amener le concepteur du programme à réécrire une partie de son code, par exemple à cause des fonctions de programmation d'IHM qui seront typiquement différentes dans les deux environnements. En revanche, comme illustré par la figure 2.6, un programme écrit en Java et compilé une fois pour toutes en *bytecode* conforme aux spécifications des JVM pourra être exécuté indifféremment sur une JVM Windows ou sur une JVM Linux, y compris s'il intègre des composants d'IHM.

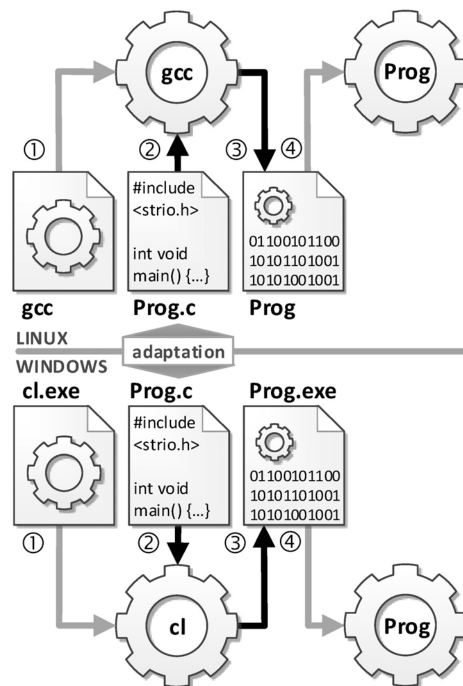

Figure 2.5 – Principe de portabilité d'un programme C

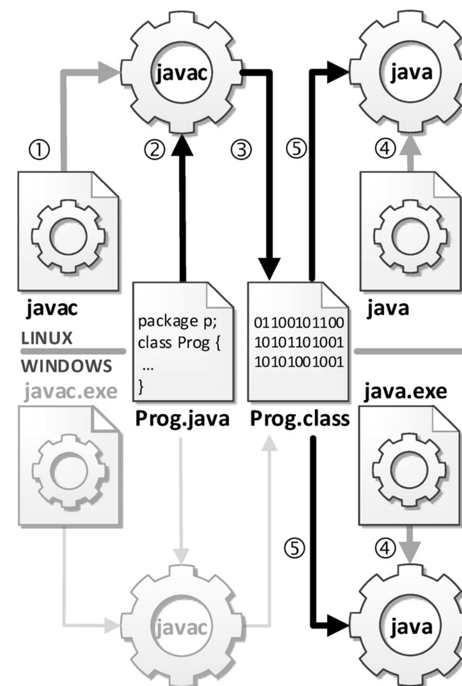

Figure 2.6 – Principe de portabilité d'un programme Java

2.2.3 Gestion mémoire de la JVM

Le concepteur de programme Java ne peut pas explicitement allouer de la mémoire de la JVM. Il n'est donc pas responsable de la libération de l'espace mémoire occupé par des objets devenus inutiles au cours de l'exécution du programme. Cette tâche est allouée à un dispositif de la JVM nommé *garbage collector*. Ce dispositif est activé périodiquement au cours de l'exécution d'un programme.

2.2.4 JVM usuelle

HotSpot est le nom commercial de la JVM proposée par *Oracle*. Elle présente de bonnes performances grâce au procédé de *compilation à la volée* (*just-in-time compilation* ou *JIT compilation* en anglais) [4] qu'elle met en œuvre. Ce procédé consiste à transformer les instructions du *bytecode* juste avant leur exécution en instructions natives. Le gain en efficacité vient de la répétition de l'exécution des mêmes instructions dans un programme (*e.g.* instructions en boucle, fonctions appelées fréquemment). En effet, ces instructions sont compilées la première fois puis sont exécutées en mode natif les fois suivantes, au lieu d'être interprétées par la JVM à chaque fois.

2.3 Frameworks de développement

2.3.1 Définition

Un framework de développement est un ensemble de bibliothèques et d'outils pour le développement d'applications. *Oracle* propose trois frameworks de développement Java :

Java SE *Java Platform, Standard Edition*, contenant en particulier :

- le compilateur Java : `javac`
- la machine virtuelle Java : `java`
- l'*API* Java (plus de 4000 classes dans Java 1.8)
- un debugger : `jdb`
- un générateur de documentation : `javadoc`
- un outil d'archivage : `jar`
- un visualisateur d'*applets* : `appletviewer`

Java EE (*Java Platform, Enterprise Edition*) composé du framework Java SE et d'un ensemble d'outils dédiés au développement d'applications n-tiers (*servlets*, EJB, JSP, jMaki (Ajax), etc.)