INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

centre de recherche RENNES - BRETAGNE ATLANTIQUE

SGDN

Projet: JAVASEC

Type : rapport d'étude

Rapport d'étude sur le langage Java

Référence	:	JAVASEC_NTE_001
Version	:	1.3
Nb pages	:	227
Date	:	14 octobre 2009

TABLE DES MATIÈRES

1 INTRODUCTION

1.1 Objet du document

Ce document est réalisé dans le cadre du Projet JAVASEC, relatif au marché 2008.01801.00.2.12.075.01 notifié le 23/12/2008. Il correspond à la version finale du premier livrable technique contractuel émis au titre du poste 1 : rapport sur le langage Java en version finale (identifiant 1.1.2 dans le CCTP).

Il constitue l'état de l'art du langage Java et des problématiques de sécurité associées.

1.2 Présentation du contexte

Java est un langage de programmation orienté objet développé par Sun. En plus d'un langage de programmation, Java fournit également une très riche bibliothèque de classes pour tous les domaines d'application de l'informatique, d'Internet aux bases de données relationnelles, des cartes à puces et téléphones portables aux superordinateurs. Java présente des caractéristiques très intéressantes qui en font une plate-forme de développement constituant l'innovation la plus intéressante apparue ces dernières années.

Dans le cadre de ses activités d'expertise, de définition et de mise en œuvre de la stratégie gouvernementale dans le domaine de la sécurité des systèmes d'information, l'ANSSI souhaite bénéficier d'une expertise scientifique et technique sur l'adéquation du langage Java au développement d'applications de sécurité, ainsi que d'une démarche permettant d'améliorer la confiance vis-à-vis de la sécurité de ces applications.

Le projet JAVASEC a pour objectif principal l'établissement de recommandations relatives au langage Java et la validation des concepts proposés pour l'implémentation dans une machine virtuelle Java (JVM). Ce document constitue un état de l'art des propriétés intrinsèques du langage Java, afin de permettre la définition d'un guide méthodologique, ainsi qu'un ensemble de restrictions, modifications et compléments sur le langage.

1.3 Organisation du document

Chapitre	Intitulé	Contenu
2	Glossaire, Acronymes	Description des différents acronymes du document.
3	Présentation de Java et des problématiques de sécurité	Présentation de l'architecture J2SE, des problématiques de sécurité pour cet environnement et définition du périmètre de l'étude.
4	Caractéristiques et propriétés du langage Java	Etude du paradigme objet sous Java, du typage et de la sémantique du langage source
5	Caractéristiques et propriétés du *bytecode* Java	Présentation du jeu d'instruction, du typage et de la sémantique du *bytecode*. Etude du *bytecode verifier* et des vérifications dynamiques de la JVM.
6	Mécanismes de sécurité offerts par les classes de base du JRE	Etude des mécanismes cryptographiques, de programmation concurrente, de contrôle d'accès et de chargement de classe (confinement).
7	Bibliothèques d'extensions relatives à la sécurité	Etat de l'art des bibliothèques d'extensions de sécurité. Sélection et étude des composants pertinents.
8	Apport des méthodes formelles	Etat de l'art de l'apport des techniques de méthodes formelles appliquées à Java en termes de sécurité.
9	Identification des déviations possibles	Etude des risques de déviations possibles et des pertes de propriétés de sécurité (mécanisme d'audit dynamique et interfaçage avec le code natif).
10	Annexes	

2 GLOSSAIRE, ACRONYMES

Acronyme	Définition
API	Application Programming Interface : interface de programmation
CLDC	Connected Limited Device Configuration : sous-ensemble des classes bibliothèques Java qui contient le minimum de programmes nécessaires pour faire fonctionner une JVM
CORBA	Common Object Request Broker Architecture : architecture logicielle pour le développement de composants pouvant être écrits dans des langages et pouvant être exécutés sur des processus différents voire sur des machines différentes
CSP	Cryptographic Service Provider
DAC	Discretionary Access Control : contrôle d'accès discrétionnaire
GC	Garbage Collector (ou glaneur de cellules) : mécanisme en charge de la gestion mémoire
IDL	Interface Description Language : langage voué à la définition de l'interface de composants logiciels afin de faire communiquer les modules implémentés dans des langages différents
J2EE	Java 2 Enterprise Edition : version de Java spécialisée pour le développement et le déploiement d'applications d'entreprise (renommée Java EE)
J2ME	Java 2 Micro Edition : version de Java spécialisée pour le développement d'applications mobiles (renommée Java ME)
J2SE	Java 2 Standard Edition : version standard de Java (renommé récemment Java SE)
JAAS	Java Authentication and Authorization Service : *package* de la bibliothèque standard pour l'authentification et le contrôle d'accès
JCA	Java Cryptography Architecture : *package* de la bibliothèque standard pour la cryptographie
JCE	Java Cryptography Extension : *package* de la bibliothèque standard pour la cryptographie
JCP	Java Community Process ou Java CertPath (suivant le contexte)
JDBC	Java DataBase Connectivity : *package* de la bibilothèque standard pour la gestion de bases de données
JDK	Java Development Kit : environnement de développement Java
JIT	Compilation Just In Time : compilation à la volée

JNI	Java Native Interface : interface de programmation pour l'intégration des fonctions natives
JPDA	Java Platform Debugger Architecture : interface de mise au point
JPSA	Java Platform Security Architecture : interface pour le contrôle d'accès de Java
JRE	Java Runtime Environment : environnement d'exécution Java
JSASL	Java Simple Authentication and Security Layer
JSR	Java Specification Request
JSSE	Java Secure Socket Extension
JVM	Java Virtual Machine : machine virtuelle Java
JVMTI	Java Virtual Machine Tool Interface : interface de mise au point
MAC	Mandatory Access Control : contrôle d'accès obligatoire
PKI	Public Key Infrastructure
PRNG	Pseudorandom Number Generator
RFC	Request For Comments
RMI	Remote Method Invocation
TPM	Trusted Platform Module
TSS	TPM Software Stack
XML	eXtensible Markup Language : langage de balisage générique

3 PRÉSENTATION DE JAVA ET DES PROBLÉMATIQUES DE SÉCURITÉ

3.1 Introduction

3.1.1 Historique

Le langage et l'environnement d'exécution Java sont issus de travaux de recherche menés dans les années 1990 par Sun Microsystems dans le cadre du projet *Stealth* (renommé *Green project* par la suite). Les ingénieurs de Sun, en particulier Patrick Naughton et James Gosling, sont amenés à développer un nouveau langage qu'ils nomment *Oak*. Celui-ci sera renommé Java en 1994 (le nom *Oak* étant utilisé par une autre société). Le but était de définir un langage de haut niveau, orienté objet et s'inspirant de C++ (la syntaxe de Java reprend en grande partie celle de C++), tout en comblant les lacunes de ce dernier en ce qui concerne la gestion de la mémoire (absence de ramasse-miettes) et la programmation concurrentielle (gestion native de plusieurs *threads*). Le projet visait initialement le marché des clients légers (PDA, etc.), mais il se réoriente au début de l'année 1990 sur les applications liées au Web. La technologie Java est caractérisée par le langage à proprement parler et un environnement d'exécution constitué d'une machine virtuelle (JVM, *Java Virtual Machine*) et d'une bibliothèque de classes de base permettant d'exécuter les programmes Java sur différentes plates-formes matérielles. Le langage et la plate-forme d'exécution sont présentés officiellement le 23 Mai 1995 lors de la conférence SunWorld avec l'application HotJava, un navigateur Web. Cette présentation est suivie de l'annonce de Netscape du support de Java dans son navigateur et marque le début de l'essor de la technologie Java. Sun crée au début de l'année 1996 l'entité Javasoft en charge du développement de la technologie Java. La première version standard de l'environnement Java (JDK 1.0) est mise à disposition par Sun en janvier 1996.

L'essor et la popularité de Java dépendent en grande partie de l'essor des technologies Web. Dès les premières versions du langage, celui-ci permet à la fois de développer des applications clientes ou serveurs ainsi que des applications (*applets*) téléchargées depuis un réseau informatique (internet) et s'exécutant au sein d'un navigateur Web. La mise à disposition gratuite par Sun de l'environnement d'exécution et des outils de développement (compilateur) via internet a également contribué à son succès. Les différentes versions du langage et de son environnement d'exécution sont présentées dans le tableau 1. Pour chaque version, les principales modifications apportées au langage, à la JVM et aux classes de l'environnement standard sont indiquées. Deux changements majeurs sont à noter dans la numérotation des versions :

- de la version 1.2 à la version 1.4, le nom commercial est Java 2 (Java Second Edition). L'environnement standard est nommé J2SE (Java 2 Standard Edition), afin de le différencier des environnements destinés aux applications Web (J2EE, Java 2 Entreprise Edition) et des applications embarquées (J2ME, Java 2 Micro Edition).

– à partir de la version 1.5, seul le numéro de version mineur apparaît dans la dénomination commerciale. Ainsi, la version 1.5 (numéro de version interne) est nommée J2SE 5.0.

Version	**Date de publication**	**Nombre de classes et interfaces**	**Principales fonctionnalitées ajoutées**
JDK 1.0	23 janvier 1996	211	Version initiale
JDK 1.1	19 février 1997	477	Ajout de nombreuses classes internes : – JavaBeans – JDBC – Java Remote Invocation (RMI)
J2SE 1.2	9 décembre 1998	1524	Révision majeure (Java 2). Plusieurs ajouts : – réflexion – API SWING – compilateur Just In Time – java IDL – framework Collections
J2SE 1.3	8 mai 2000	1840	– intégration de la JVM HotSpot – RMI basé sur CORBA – Java Naming and Directory Interface – Java Platform Debugger Architecture

Version	Date de publication	Nombre de classes et interfaces	Principales fonctionnalitées ajoutées
J2SE 1.4	6 février 2002	2723	Première révision sous Java Community Process : – mot-clé `assert` – expressions rationnelles basées sur Perl – API de journalisation – JAXP (parser XML et moteur XSLT) – intégration des extensions de sécurité JCE, JSSE et JAAS – Java Web Start
J2SE 5.0	30 septembre 2004	3270	Révision majeure : – Programmation générique – Metadata (annotations) – Autoboxing/unboxing (convertion automatique pour les types primitifs) – Varargs – Imports statiques – Extension de la syntaxe de `for` pour les itérations
Java SE 6	11 décembre 2006	3777	– libération du code source (OpenJDK) – XML Digital Signature API – Scripting for the Java platform – Stack Maps

TABLE 1: Les différentes versions de l'architecture standard Java

JAVASEC se concentre principalement sur l'étude des applications autonomes exécutées sur un environnement d'exécution standard Java. L'étude considère que l'environnement d'exécu-

tion Java couvre au minimum la version Java SE 5.0, mais les nouveautés apparues dans Java SE 6.0 sont également abordées.

L'exécution de code distribué (*applet*) ou d'applications côté serveur (JSP, Servlet, Web-Services, etc.) sort *a priori* du cadre de cette étude. L'OWASP a récemment débuté un projet concernant la sécurisation des applications Web écrites en Java [1]. Ce site fournit notamment des recommandations spécifiques au développement pour la plate-forme J2EE (prévention des injections de code, validation des entrées, etc.). Les classes des bibliothèques des environnements J2EE et J2ME pourront éventuellement être considérées dans le cadre de l'étude des extensions de l'environnement standard, s'il s'avère qu'elles peuvent apporter un gain significatif dans la sécurité des applications Java développées pour l'environnement Java SE.

3.1.2 Les principaux concepts de Java

La philosophie de Java permet de répondre à plusieurs objectifs :

1. utiliser un langage de haut niveau et orienté objet ;
2. faciliter le développement des applications en proposant un langage simple et inspiré de C++ ;
3. faciliter le déploiement des applications en s'assurant qu'une même application puisse s'exécuter sur des environnements différents (UNIX, Windows, Mac, etc.) ;
4. permettre l'utilisation de manière native des réseaux informatiques ;
5. permettre l'exécution de code distant de manière sécurisée.

Le langage Java, à proprement parler, permet de répondre aux deux premiers objectifs. Il s'agit en effet d'un langage orienté objet fortement inspiré de C++ de par sa syntaxe. Les principales différences par rapport à C++ sont les suivantes :

- la gestion de la mémoire est automatisée via l'utilisation d'un ramasse-miettes (*garbage collector*). Le programmeur ne peut gérer directement les allocations mémoires et manipule les objets via les références qui permettent d'abstraire la notion de pointeur utilisée en C++ ;
- l'héritage multiple n'est pas possible en Java qui propose en revanche la notion d'interface. À la différence d'une classe, une interface ne contient pas de code, mais seulement des descriptions de méthodes et des constantes. Une classe Java hérite au plus d'une classe et peut implémenter plusieurs interfaces.
- l'intégralité du code doit être implémentée sous la forme de méthodes de classes (statique ou non) ;
- le compilateur Java n'utilise pas de préprocesseur. Il n'est donc pas possible d'utiliser des macros ou des définitions de constantes (`#define`, etc.) ;

1. `http://www.owasp.org/index.php/Category:OWASP_Java_Project`

- le type primitif `boolean` n'est pas compatible avec les types numériques (`int`, `long`, etc.) ;
- l'instruction `goto` a été supprimée.

L'environnement standard de Java définit, en plus du langage, un certain nombre de classes de base regroupées dans des *packages* de la bibliothèque standard. Ces classes offrent différents services, tels que l'accès aux ressources du système (fichiers, base de données, etc.). Elles permettent notamment de programmer des applications communiquant via un réseau informatique grâce à la gestion des *sockets* ou des applications utilisant la programmation concurrente (gestion de plusieurs files d'exécutions ou *threads*). Elles répondent donc en partie aux objectifs 2 et 4.

Afin de répondre au troisième objectif, le modèle standard d'exécution des programmes Java repose sur l'utilisation d'une machine virtuelle (JVM). Le compilateur Java effectue la traduction du code source Java de haut niveau vers un langage de niveau intermédiaire ou *bytecode* Java. Ce dernier est, par la suite, exécuté sur la machine virtuelle. Celle-ci constitue une interface standard qui permet d'abstraire les différentes plates-formes d'exécution. Elle assure notamment l'exécution du *bytecode* sur l'architecture de la plate-forme d'exécution (X86, Sparc, PPC, etc.). Historiquement, l'exécution est réalisée par un interpréteur mais, pour des raisons d'optimisation, les JVM actuelles intègrent généralement un compilateur « à la volée » (ou *JIT*). L'utilisation d'une machine virtuelle permet de déployer les applications Java sous forme compilées (bytecode) tout en garantissant que l'application s'exécutera (dans le cas idéal) de la même manière sur les différentes plates-formes natives. La JVM permet également, en utilisant certaines classes de l'environnement standard, de confiner les différentes classes d'une application Java. La spécification de la JVM impose que celle-ci effectue un certain nombre de vérifications sur le *bytecode* qu'elle exécute. Ces vérifications limitent les attaques qu'il est possible de réaliser. L'exécution sur JVM permet donc aussi de traiter l'objectif 5.

La notion d'application Java désigne un programme autonome fourni sous forme de *bytecode* et s'exécutant sur une JVM. La spécification de cette dernière ne précise pas si une instance d'une JVM permet d'exécuter une ou plusieurs applications. Toutefois, dans la majorité des cas, les implémentations de la JVM adoptent le modèle suivant : chaque application Java s'exécute sur une instance différente de la JVM sous la forme d'un processus de l'OS de la plate-forme native.

3.2 L'environnement d'exécution standard (Java SE 6)

Comme évoquée précédemment, l'architecture de l'environnement Java SE, illustrée par la figure 2, comprend différents éléments. Cette architecture couvre le modèle d'exécution standard, c'est-à-dire celui reposant sur l'utilisation du *bytecode* et de la JVM. Les autres modèles d'exécution (compilation native, *Java Processor*, etc.) seront présentés dans le document « Rapport sur les modèles d'exécution Java » [13]. La figure 1 résume les principaux composants de

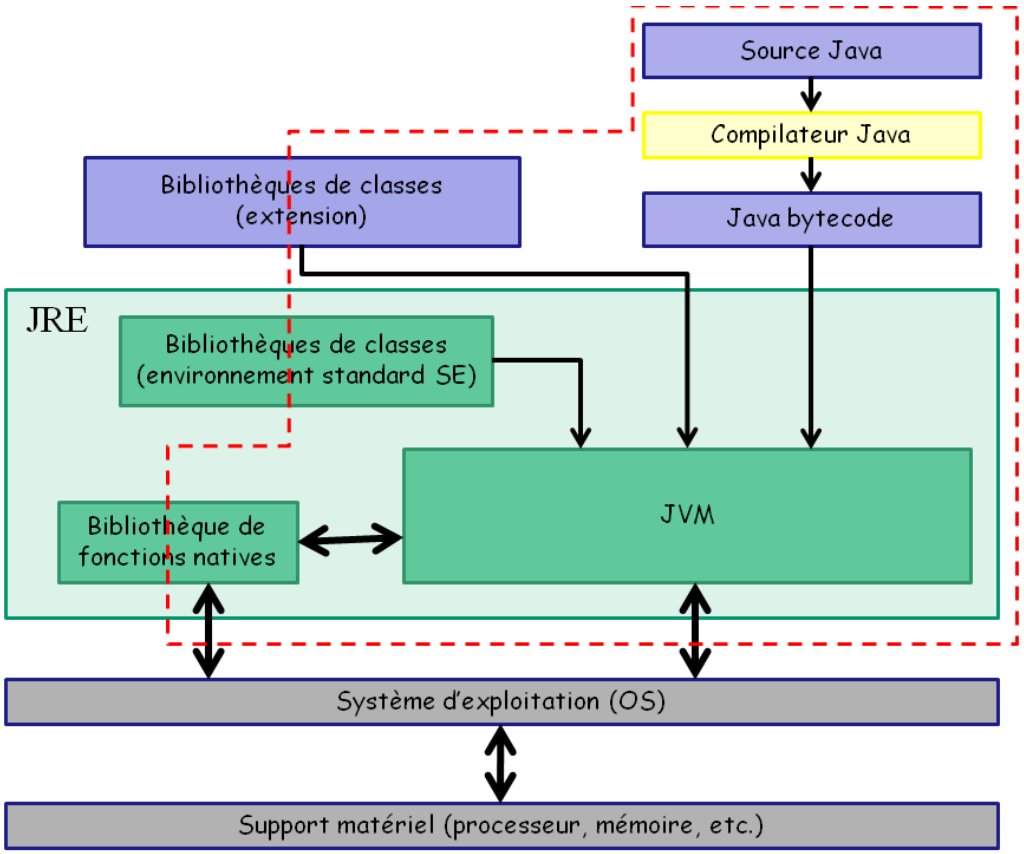


FIGURE 1 – Architecture Java

cette architecture et fixe le périmètre de l'étude JAVASEC (matérialisé par les pointillés rouges). Ces composants sont les suivants :

1. La machine virtuelle Java (JVM) dont HotSpot constitue l'implémentation de Sun. Ce composant est décrit dans les rapports [14], [11] et [9] de l'étude JAVASEC. Sa configuration est abordée dans le rapport [12]. Il assure l'interface entre la plate-forme d'exécution native (*hardware* + système d'exploitation) et les applications Java compilées sous forme de *bytecode*. La spécification de la JVM est ouverte et publiée par Sun [49]. Cette spécification couvre, entre autres, la définition du *bytecode* et les principales structures de données de la JVM. D'autres éditeurs et projets *open source* peuvent ainsi proposer leur propre implémentation de la JVM. En pratique, la JVM assure les fonctions suivantes :
   - la **vérification et l'exécution du *bytecode*** (le jeu d'instructions de la machine virtuelle) sur la plate-forme native. Cette exécution peut être réalisée par un interpréteur ou un compilateur JIT (*Just In Time*). Certaines implémentations de JVM comme HotSpot intègrent ces deux modes d'exécution et commutent dynamiquement d'un mode d'exécution à l'autre (compilation dynamique) ;
   - la **gestion de la mémoire**, notamment en ce qui concerne l'allocation et la libération dynamique de la mémoire pour les objets créés. Cette fonction est assurée par le ramasse-miettes (*garbage collector*) ;

- la **gestion des différents *threads*** d'exécution (partage du temps processeur). La JVM peut pour cela s'appuyer en partie sur la gestion des *threads* fournie par l'OS. La JVM doit gérer différents types de *threads* :
  - les *threads* Java définis par le programmeur de l'application exécutée,
  - les *threads* d'exécution du code natif (utilisés notamment dans la bibliothèque standard),
  - les *threads* internes à la JVM correspondant aux différents services de cette dernière (*garbage collector*, *debugger*, compilateur JIT, etc.)

  Elle assure également la synchronisation de ces différents *threads* ;
- le **chargement, l'édition de liens et l'initialisation des classes** de l'application et de la bibliothèque standard. Le chargement des classes est réalisé par les chargeurs de classes (*class loaders*) dont la plupart sont implémentés dans certaines classes de la bibliothèque standard. Le chargement des classes système et les autres étapes sont réalisés par la JVM ;
- l'**interfaçage avec certaines fonctionnalités** offertes par le système d'exploitation (gestion des signaux) ;
- la **gestion des erreurs et des exceptions** Java ;
- l'**interface avec les fonctions de la plate-forme native** (JNI, *Java Native Interface*) ;
- l'**implémentation de certaines méthodes** utilisées par la bibliothèque standard pour des raisons d'optimisation (par exemple, `java.lang.Math.sin`), de facilité d'implémentation (par exemple, `java.lang.String.indexOf`) ou de sûreté (par exemple, `sun.misc.Unsafe.compareAndSwapInt`).

2. La bibliothèque standard Java, abordée en partie dans ce document. Depuis la version 2 de Java, celle-ci est déclinée en trois « éditions » : Standard Edition (qui fait l'objet de cette étude), Enterprise Edition (pour les applications Web, Servlet/JSP, etc.) et Micro Edition (pour les applications embarquées). L'API est publique et documentée par Sun pour chaque version de Java [2]. L'ensemble des classes de la bibliothèque standard sont regroupées en *packages*. La bibliothèque standard comprend des classes ainsi que des fonctions natives implémentées en C/C++ qui sont utilisées par les classes de l'API mais qui n'implémentent pas de méthodes de l'API (par exemple, `sun.misc.VM`). Ces classes sont parfois disponibles uniquement sur certaines architectures natives (par exemple, `sun.security.provider.NativePRNG` n'est disponible, dans l'implémentation de Sun, que pour les architectures LINUX et Solaris). Ces classes sont propres à chaque implémentation et ne sont donc pas normalisées. L'implémentation de Sun de la bibliothèque standard est maintenant en partie ouverte au sein du projet OpenJDK. La bibliothèque standard de Java offre un nombre important de fonctionnalités dont entre autres :
   - la gestion des interfaces graphiques (`java.lang.awt`, `javax.swing`, etc.) ;
   - la gestion des entrées/sorties, notamment sur les fichiers gérés par l'OS (`java.lang.io`, `java.lang.nio`, etc.) ;

2. `http://java.sun.com/javase/6/docs/api/`

- la gestion des communications réseau, notamment via la notion de socket (`java.net`, `javax.net`, etc.) ;
- les classes fondamentales du langage Java (`java.lang`) comme les chaînes de caractères (`String`) ou les classes enveloppant les types primitifs (`Integer`, `Character`, etc.) ;
- les classes offrant des fonctions mathématiques (`java.lang.math`, `javax.math`, etc.) ;
- les classes implémentant le mécanisme de contrôle d'accès (`java.security`) ou des primitives cryptographiques (`javax.crypto`).

3. Le langage Java à proprement parler, abordé dans ce document. Ce composant est décrit dans la norme Java Language Specification [33] qui définit notamment la syntaxe du langage.
4. Des outils graphiques ou en ligne de commande utilisés pour le développement et le déploiement d'applications Java. JAVASEC se focalise notamment sur l'étude du compilateur Java, décrit dans le rapport [10], qui assure la traduction des fichiers source en Java vers les fichiers de classe en *bytecode*. Le compilateur s'appuie donc sur la spécification du langage Java [33, 49] et sur celle du *bytecode* [49]. Le programme `javac` constitue l'implémentation de Sun dont le code source est maintenant ouvert et disponible au sein du programme OpenJDK. Les outils de lancement de la JVM (`java`), d'audit dynamiques (JVMTI, JPDA, etc.) et de configuration de la politique sont également abordés.

L'ensemble des composants énumérés précédemment permet de développer et d'exécuter des applications Java (sous réserve d'implémentation effective de l'architecture Java pour la plate-forme utilisée). Sun les regroupe sous le terme JDK (Java Development Kit). Une version épurée du JDK (privé des outils de développement) est nommée JRE (Java Runtime Environment). Le JRE comprend la JVM et la bibliothèque standard. Il regroupe l'ensemble minimal des composants de l'architecture Java nécessaire pour l'exécution d'applications Java fournies sous la forme de fichiers de classes (en *bytecode*).

3.3 La problématique de la sécurité pour les applications Java

JAVASEC se focalise sur les aspects de sécurité informatique liés à l'environnement Java. Il convient donc de définir, dans un premier temps, les propriétés de sécurité à assurer ainsi que les biens à protéger situés dans le périmètre de l'étude. Puis, dans un second temps, il convient d'identifier les fonctions et mécanismes de sécurité utilisés dans l'environnement Java.

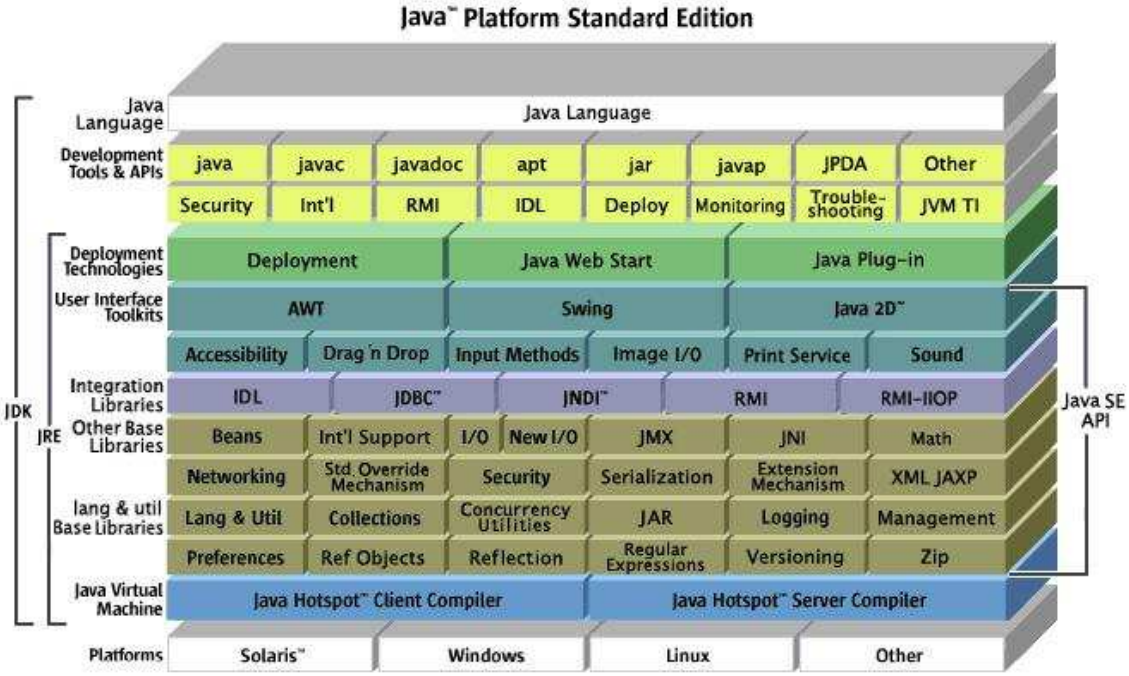


FIGURE 2 – Architecture J2SE (source Sun)

3.3.1 Propriétés de sécurité

La sécurité informatique consiste à assurer des propriétés de sécurité sur les données et services qui sont classiquement regroupées en trois catégories :

- les propriétés de confidentialité qui concernent la lecture des données (ou l'inférence d'informations sur les données) ;
- les propriétés d'intégrité qui concernent la modification des données ;
- les propriétés de disponibilité qui concernent l'accès aux données ou services en un temps fini.

Ces propriétés sont garanties par l'utilisation de mécanismes de sécurité ou par la mise en place de mesures organisationnelles. Malheureusement, ces mécanismes et ces mesures peuvent être contournés par un attaquant. Il est donc nécessaire de recourir également à des mesures d'audit *a posteriori* du système. Une quatrième catégorie de propriétés doit donc être considérée : l'auditabilité. Celle-ci garantie l'existence d'une trace dans un fichier journal de tous les accès réalisés sur les données ou les services à protéger. Les différents fichiers journaux constituent également un bien qu'il convient de protéger (notamment en intégrité).

Ces propriétés peuvent s'appliquer sur différents types de biens et de services qui sont précisés par la suite.

3.3.2 Biens et services à protéger

Dans le cadre de cette étude, quatre grandes familles de biens ont été identifiées :

- les données et le code exécutable de l'application Java (D_APP et C_APP) ;
- les données et le code exécutable de la JVM (D_JVM et C_JVM) ;
- les données et le code exécutable des bibliothèques du JRE et des extensions (D_LIB et C_LIB) ;
- les données présentes sur le système natif qui n'entrent pas dans les catégories précédentes et qui sont potentiellement accessibles par une application Java (D_OTHER).

La dernière catégorie ne correspond pas à des biens propres à la plate-forme Java. Ces biens sont donc protégés par des mécanismes de sécurité de la plate-forme d'exécution native (par exemple, le contrôle d'accès du système d'exploitation). Toutefois, dans le cadre de cette étude, il peut s'avérer nécessaire d'assurer des propriétés différentes sur ce type de biens suivant les éléments de la plate-forme Java. Typiquement, certains éléments (la JVM, certaines classes de confiance, etc.) nécessitent des privilèges qui ne doivent pas être attribués à d'autres éléments (par exemple, une application Java qui peut s'avérer malveillante ou présenter une faille de sécurité). La plate-forme d'exécution native n'étant pas toujours à même de distinguer les différents éléments de la plate-forme Java, il peut s'avérer nécessaire de mettre en œuvre des mécanismes pour protéger ce type de bien au sein même de la plate-forme Java.

Cette classification peut être raffinée suivant des critères de forme et de localisation. Ainsi, le code exécutable de l'application peut exister sous différentes formes :

- sous forme de fichiers texte contenant le code source Java (C_APP_SRC) ;
- sous forme de *bytecode* Java contenu dans les fichiers classes et chargé en mémoire par la JVM (C_APP_BC) ;
- sous forme de code binaire exécutable par l'architecture matérielle après compilation (C_APP_BIN). Cette forme compilée peut être obtenue par l'utilisation d'un compilateur « hors-ligne »s (ce modèle d'exécution particulier, non standard, est étudié dans le « Rapport sur les modèles d'exécution de Java »). Elle peut également être produite par le compilateur JIT de la JVM. Cette forme inclut également le code binaire des fonctions natives développées pour l'application. Ce code est contenu dans des fichiers de bibliothèques partagées qui peuvent être chargées en mémoire par la JVM.

La même catégorisation peut être retenue pour le code des bibliothèques Java. Le code source est cependant considéré hors périmètre pour cette famille de bien. En effet, l'hypothèse est faite que les classes des bibliothèques sont des COTS ou des composants fournis par le JRE. Dans les deux cas, seule la forme *bytecode* est utilisée. Les bibliothèques dont le code source est utilisé par les concepteurs de l'application (par exemple à des fins de modification ou d'adaptation) sont considérées comme faisant partie du code de l'application (C_APP). Les catégories retenues sont donc les suivantes :

- le *bytecode* des classes des bibliothèques Java (C_LIB_BC) ;

– le code binaire des bibliothèques Java (dont les fonctions natives) (C_LIB_BIN).

La JVM est généralement implémentée grâce à un langage compilé (C/C++). C'est notamment le cas de celle de Sun qui est l'implémentation de référence pour cette étude. Seule la forme de code binaire est donc retenue pour la JVM (C_JVM_BIN) qui est généralement implémentée sous la forme d'une bibliothèque partagée.

De plus, les données et le code exécutable peuvent être situés dans différents types de conteneurs. Ainsi, pour les applications Java, il est possible de distinguer les différents cas suivants :

– les données et le code stockés dans une mémoire de masse gérée par le système d'exploitation. Il s'agit par exemple, pour les données de l'application (D_APP_HD), de fichiers de configuration ou de bases de données. Ce cas comprend également les fichiers sources (C_APP_SRC_HD) et les fichiers de classes (C_APP_BC_HD) de l'application. Dans le cas particulier où la compilation native est utilisée (au lieu du modèle d'exécution standard via une JVM), le code de l'application existe également sous forme binaire (C_APP_BIN_HD).
– les données et le code de l'application stockés en mémoire vive (dans l'espace mémoire associée à l'application Java). Les données (D_APP_MEM) sont stockées, pour la partie Java, sous la forme de constantes, de variables locales, de champs d'objets (ou de classes) et de tableaux. Elles peuvent elles-mêmes provenir de données réseau ou de données stockées au préalable dans un fichier. Dans le modèle d'exécution standard, le code est stocké en mémoire par la JVM sous forme de *bytecode* (C_APP_BC_MEM) et, si la JVM utilise un compilateur JIT, sous forme binaire (C_APP_BIN_MEM). Dans le modèle d'exécution native, le code de l'application est exécuté directement sous sa forme binaire par la plate-forme d'exécution native (C_APP_BIN_MEM).
– les données échangées sur le réseau (D_APP_NET). Il s'agit des paquets émis et reçus par une application cliente ou serveur. En théorie, le code de l'application peut également être échangé sur le réseau. Toutefois, cette étude ne porte pas sur les problématiques de code mobile. Cette catégorie de biens n'a donc pas été retenue.

La même classification peut être retenue pour les données (D_LIB_HD, D_LIB_NET, D_LIB_MEM) et le code (C_LIB_BC_HD, C_LIB_BIN_HD, C_LIB_BC_MEM, C_LIB_BIN_MEM) des bibliothèques Java.

Elle peut également l'être pour les données propres à la JVM (D_JVM_HD, D_JVM_NET et D_JVM_MEM) et le code de la JVM (C_JVM_BIN_HD, C_JVM_BIN_MEM). Les données de la JVM excluent les biens suivants qui, bien que pouvant être considérés comme des données utilisées et gérées par la JVM, sont regroupés dans d'autres catégories identifiées au préalable :

– le code des applications, sous forme binaire et sous forme de *bytecode* (C_APP_BC et C_APP_BIN) ;
– les données des applications (D_APP_HD, D_APP_NET et D_APP_MEM) ;
– le code des bibliothèques, sous forme binaire et sous forme de *bytecode* (C_LIB_BC et C_LIB_BIN) ;

– les données des applications (D_LIB_HD, D_LIB_NET et D_LIB_MEM).

Elle s'applique enfin aux autres données du système : D_OTHER_HD, D_OTHER_NET, D_OTHER_MEM.

Les biens à protéger comprennent également des services offerts pas les applications Java (SER_APP), les bibliothèques (SER_LIB) et la JVM (SER_JVM). En effet, la plate-forme d'exécution Java doit garantir la disponiblité de ces services et assurer l'auditabilité de certains services.

Ces différentes catégories de biens et de services sont résumées dans les tableaux 2, 3, 4, 5 et 6. Ces tableaux indiquent également les menaces qui doivent être couvertes par des mécanismes de sécurité Java pour les différents biens et services. Deux catégories de menaces ne sont pas retenues pour cette étude :

– les menaces à l'encontre de la disponibilité sur les biens (données ou codes) stockés sur la mémoire de masse à l'exception du *bytecode* (et du code binaire des bibliothèques), dont la disponibilité est en partie assurée par le mécanisme de chargeur de classes. Ces menaces doivent être couvertes par des mécanismes implémentés au sein du système d'exploitation. En revanche, les menaces à la confidentialité et à l'intégrité doivent êtres couvertes par la JVM, car il peut être nécessaire de restreindre les droits d'accès de certains composants de la plate-forme Java.
– les menaces à l'encontre de la confidentialité sur le code binaire ou le *bytecode*. En effet, le code binaire et le *bytecode* des bibliothèques et de la JVM sont considérés comme publics. Il peut exister une exigence de confidentialité sur le code des applications. Celle-ci concerne le code source et éventuellement le *bytecode* de l'application. La confidentialité peut dans ce cas être partiellement couverte par des techniques d'offuscation ou de chiffrement de classes.

Propriétés de sécurité	C_APP						D_APP		
	C_APP_SRC	C_APP_BC		C_APP_BIN			D_APP_HD	D_APP_NET	D_APP_MEM
	C_APP_SRC_HD	C_APP_BC_HD	C_APP_BC_MEM	C_APP_BIN_HD	C_APP_BIN_MEM				
Confidentialité	✓						✓	✓	✓
Intégrité	✓	✓	✓	✓	✓		✓	✓	✓
Disponibilité		✓	✓	✓	✓				✓
Auditabilité		✓	✓	✓	✓		✓	✓	✓

TABLE 2 – Menaces sur l'architecture Java (Applications)

Propriétés de sécurité	C_LIB						D_LIB		
	C_LIB_BC			C_LIB_BIN			D_LIB_HD	D_LIB_NET	D_LIB_MEM
	C_LIB_BC_HD	C_LIB_BC_MEM		C_LIB_BIN_HD	C_LIB_BIN_MEM				
Confidentialité							✓	✓	✓
Intégrité	✓	✓		✓	✓		✓	✓	✓
Disponibilité	✓	✓		✓	✓				✓
Auditabilité	✓	✓		✓	✓		✓	✓	✓

TABLE 3 – Menaces sur l'architecture Java (Bibliothèques)

Propriétés de sécurité	C_JVM		D_JVM	
	C_JVM_BIN		D_JVM_HD	D_JVM_MEM
	C_JVM_BIN_HD	D_JVM_BIN_MEM		
Confidentialité			✓	✓
Intégrité	✓	✓	✓	✓
Disponibilité				✓
Auditabilité			✓	✓

TABLE 4 – Menaces sur l'architecture Java (JVM)

Propriétés de sécurité	D_OTHER		
	D_OTHER_HD	D_OTHER_MEM	D_OTHER_NET
Confidentialité	✓	✓	✓
Intégrité	✓	✓	✓
Disponibilité			
Auditabilité	✓	✓	✓

TABLE 5 – Menaces sur les autres biens du système

Propriétés de sécurité	S_APP	S_LIB	S_JVM
Disponibilité	✓	✓	✓
Auditabilité	✓	✓	✓

TABLE 6 – Menaces sur les services

3.3.3 Fonctions et mécanismes de sécurité offerts par la plate-forme Java

Différentes fonctions et mécanismes de sécurité sont présents ou peuvent être mis en œuvre dans Java. Ces mécanismes peuvent être classés en deux catégories :

- les mécanismes natifs qui s'appliquent à toutes les applications Java de manière relativement transparente pour l'utilisateur et le développeur ;
- les mécanismes optionnels qui peuvent être utilisés par les développeurs pour implémenter des fonctions de sécurité au sein des applications Java.

3.3.3.1 Mécanismes de sécurité natifs

La sécurité de Java repose, en premier lieu, sur des mécanismes intrinsèques au langage. Il s'agit essentiellement de mécanismes de vérification de l'innocuité du code des applications et des bibliothèques Java. Ainsi, le compilateur Java effectue un certain nombre de vérifications sur le code source des applications (syntaxe, vérification de types, visibilité, etc.).

Une part importante de la sécurité de la plate-forme d'exécution Java est assurée par la JVM, notamment au niveau du *bytecode*. Ainsi l'innocuité [3] du *bytecode* est assurée par un mécanisme de vérification de *bytecode* (*bytecode verifier*) intégré à la JVM. Lors du chargement des classes, ce mécanisme vérifie la syntaxe du *bytecode*, effectue un contrôle sur les types des variables, vérifie la taille et le format de la pile d'exécution, etc. D'autres mécanismes de la JVM intègrent des fonctions de sécurité :

- les mécanismes de gestion de la mémoire (vérification de l'index des tableaux, *garbage collector*, gestion de la mémoire pour les différents mécanismes internes de la JVM, etc.) assurent l'intégrité (et dans une certaine mesure la confidentialité) des données gérées par la JVM (données et code des applications, données et code des bibliothèques, données propres de la JVM).
- lorsqu'il est implémenté, le mécanisme de compilation à la volée (compilateur JIT) doit assurer l'intégrité et la disponibilité du code binaire généré.

D'autres mécanismes de sécurité sont mis en œuvre conjointement par la JVM et la bibliothèque standard :

- Le mécanisme de chargement de classe (*class loader*) assure la disponibilité des classes des applications et bibliothèques ainsi que leur intégrité (via le mécanisme de signature de classe). Il assure également pour partie (avec le gestionnaire de sécurité) le cloisonnement entre les différents composants Java s'exécutant sur une même instance de la JVM. Il a donc un impact sur la confidentialité, l'intégrité et la disponibilité des données des applications et des bibliothèques Java. Toutefois, le cloisonne-

3. Il s'agit d'une forme d'intégrité, différente du sens habituel utilisé en cryptographie, qui caractérise le fait que le *bytecode* ait été généré en respectant certaines règles, notamment de typage, ce qui permet d'assurer des propriétés désirées.

ment nécessite un paramétrage explicite de la part du développeur (il s'agit donc pour partie d'un mécanisme de sécurité optionnel).

- Les mécanismes de partage des ressources (gestion des *threads*) assurent en partie la disponibilité des différents services (la JVM, des bibliothèques et des applications). Il a également un impact sur la disponibilité des ressources partagées entre plusieurs fils d'exécution (*threads* natifs et *threads* Java). Toutefois, la disponibilité des services et des données des programmes et des bibliothèques Java ne peut être assurée que si le programmeur synchronise correctement les accès concurrentiels.
- Les mécanismes natifs d'audit fournis par la JVM.

Lorsque le modèle d'exécution diffère du modèle d'exécution standard (par exemple, utilisation de la compilation native), les propriétés assurées par les mécanismes de la JVM ne le sont plus. Plusieurs solutions peuvent être adoptées :

- la propriété est assurée par un autre mécanisme (par exemple, le compilateur) ;
- le mécanisme est intégré nativement au code exécutable.

L'étude sur les modèles d'exécution, présentée dans [14], identifie ces déviations entre le modèle d'exécution standard et les autres modèles (compilation native et utilisation de processeurs Java).

Enfin, la plate-forme d'exécution peut s'appuyer sur des mécanismes de sécurité fournis par l'OS. L'analyse de ces mécanismes à proprement parler se situe en dehors du périmètre de cette étude. En revanche, il convient d'identifier les propriétés qui sont assurées par ces mécanismes. Il convient également de s'assurer que l'implémentation de la plate-forme d'exécution s'interface correctement avec ces mécanismes.

3.3.3.2 Mécanismes de sécurité optionnels

La confidentialité du code source peut être assurée en partie par des extensions utilisant des techniques d'offuscation ou de chiffrement de classes. De même, certains outils (notamment d'analyse statique) peuvent être utilisés afin de vérifier l'innocuité du code source.

La bibliothèque standard de Java propose également des mécanismes de sécurité qui peuvent être mis en œuvre pour sécuriser une application Java :

- Les mécanismes cryptographiques permettent d'assurer la confidentialité et l'intégrité des données de l'application via les opérations de chiffrement, de signature et d'authentification.
- Les mécanismes d'authentification et de contrôle d'accès permettent de définir des règles de contrôle d'accès pour les différents composants (classes Java) de la plate-forme. Ils permettent également d'auditer en partie les accès réalisés par les applications et les bibliothèques Java.
- Le mécanisme de signature des classes et des fichiers d'archives (JAR) permet de garantir l'intégrité de la provenance des classes chargées par la JVM.

3.4 Périmètre de l'étude

Cette étude porte sur les propriétés de sécurité assurées par l'environnement d'exécution standard, ainsi que sur les mécanismes de sécurité offerts par cet environnement. L'étude porte également sur des extensions de sécurité, non présentes actuellement dans l'environnement standard, qui pourraient être ajoutées afin de garantir de nouvelles propriétés ou d'offrir de nouveaux mécanismes. Cette étude se focalise donc sur les éléments suivants de la plate-forme standard Java :

- le langage Java (sémantique, typage, paradigme objet, etc.) ;
- le *bytecode* Java et le mécanisme de vérification de *bytecode* de la JVM ;
- certains mécanismes de sécurité offerts par la bibliothèque standard, parfois en collaboration avec la JVM :
  - les mécanismes de contrôle d'accès (*security manager*),
  - les mécanismes de chargement de classes,
  - la programmation par *threads*,
  - les API de contrôle d'accès et de cryptographie.

Certaines classes de la bibliothèque standard, qui n'implémentent pas de mécanismes de sécurité à proprement parler, mais dont l'utilisation entraîne potentiellement un impact sur la sécurité sont également abordées. Il s'agit notamment des classes suivantes :

- les classes permettant de réaliser de l'introspection (API de réflexion) ;
- les classes concernées par le contrôle d'accès. Il s'agit par exemple des classes permettant d'accéder aux ressources du système d'exploitation accessibles en Java (fichiers, sockets, etc.). Ces classes s'interfacent avec le mécanisme de contrôle d'accès via la présence de points d'ancrage (ou *hooks*) permettant d'effectuer des appels aux méthodes du gestionnaire de sécurité.

Cette étude comporte également un état de l'art des extensions pertinentes qui pouraient être intégrées à l'environnement standard afin d'offrir de nouveaux mécanismes de sécurité ou d'améliorer les mécanismes existants. Cet état de l'art aborde les aspects suivants :

- les bibliothèques d'extensions implémentant des mécanismes de sécurité (cryptographie, contrôle d'accès) ;
- les mécanismes de vérification de code source ou de *bytecode* s'appuyant sur des démarches formelles ou semi-formelles.

Dans un premier temps, la JVM est considérée comme une boîte noire implémentant les services identifiés dans la spécification de Sun [49]. Les détails concernant l'implémentation des mécanismes fournis par la JVM (gestion de la mémoire et des *threads*, chargement interne des classes, interprétation et compilation *JIT*, etc.) sont abordés dans l'étude « Modèles d'exécutions de Java » [14], ainsi que dans le « Comparatif des JVM » [11].

Cette étude aborde en revanche les déviations possibles lors de l'interfaçage entre la plate-forme Java et la plate-forme d'exécution native. Les impacts liés à l'utilisation de JNI sont notamment étudiés.

4 CARACTÉRISTIQUES ET PROPRIÉTÉS DU LANGAGE JAVA

Cette section vise à examiner les propriétés intrinsèques du langage Java. L'analyse est divisée en sous-parties traitant chacune un aspect du langage. Chaque sous-partie est structurée de la façon suivante :

- description de l'aspect du langage ;
- risques sécuritaires liés à cet aspect ;
- renforcements possibles de la sécurité liée à cet aspect (par restriction du langage et/ou analyse supplémentaire).

4.1 Paradigme objet

Nous présentons d'abord les concepts de base de la programmation objet en Java. La présentation n'a pas comme objectif de fournir une description exhaustive du langage, mais vise à décrire les éléments pertinents pour la sécurité.

4.1.1 Classes

Un programme Java est composé de plusieurs classes, organisées dans une hiérarchie d'héritage. Dans la programmation par objets, une classe sert à modéliser une entité par les données qui constituent l'entité et les méthodes qui s'appliquent à ces données. Une application peut instancier une classe pour créer des objets.

Une classe a plusieurs types de membres :

- les champs ;
- les méthodes ;
- les constructeurs.

Les champs (ou *fields*) d'un objet stockent les données de l'objet. Un champ peut être soit une valeur de base (un entier, un nombre flottant, un booléen, un caractère), soit une référence à un autre objet, soit un tableau à une ou plusieurs dimensions contenant des objets utilisant un type de base ou une référence. Une classe liste les noms des champs avec leur type qui indique s'il agit d'une valeur de base ou d'une référence à un objet d'une classe donnée.

Les méthodes d'une classe définissent les opérations et les calculs possibles sur les objets de cette classe. Les méthodes peuvent être appelées (on dit aussi invoquées) en indiquant une référence à un objet et le nom de la méthode à appeler. Une méthode d'un objet peut accéder à toutes

les données de l'objet, ainsi qu'aux données **visibles** (voir la description des modificateurs de visibilité plus bas) des autres objets sur lesquels la méthode possède une référence.

Une classe peut hériter une méthode d'une classe parent (voir section 4.1.7), ce qui a un impact sur le code qui est effectivement exécuté lors d'un appel de méthode.

Les constructeurs d'un objet sont des méthodes particulières qui sont appelées lors de la création d'un objet pour initialiser les champs de l'objet. Un constructeur d'une classe est une méthode avec le même nom que sa classe. Une classe peut avoir plusieurs constructeurs ayant le même nom, mais prenant différents arguments. Un constructeur doit commencer par appeler un constructeur de sa super-classe ou un autre constructeur de sa classe [4], l'objectif étant d'assurer que tous les constructeurs des classes ayant contribué à la définition de la classe de l'objet soient appelés. Si le développeur ne programme pas cet appel explicitement, le compilateur Java insère, lors de la compilation, un appel au constructeur par défaut de sa super-classe, c'est-à-dire celui sans paramètre.

Une classe peut finalement contenir des fragments de code entre `{` et `}`. Si ces fragments sont précédés du mot-clé `static`, alors ils sont considérés comme faisant partie du bloc d'initialisation de la classe [5] ; sinon le code de ces fragments sera considéré comme faisant partie des constructeurs de la classe et sera ajouté au code exécuté dans chaque constructeur.

Le langage Java offre un ensemble de mots-clés (les **modificateurs**) qui peuvent être attribués aux classes, méthodes et champs d'un programme Java. L'impact des mots-clés varie en fonction de l'entité (classe, méthode, champ) à laquelle ils s'appliquent. Ces mots-clés sont :

`public, protected, private, static, abstract, final, transient, strictfp, synchronized, volatile, native.`

Les mots-clés `private, protected, public` seront présentés dans la section 4.1.2. Le mot-clé `static` sera expliqué en section 4.1.3 et le mot-clé `abstract` sera expliqué dans la section 4.1.5 sur les interfaces et classes abstraites. Le mot-clé `final` sera expliqué dans la section 4.1.4. Le mot-clé `transient` ne s'applique qu'aux champs non-statiques et concerne le stockage d'objets comme données explicites ; voir la description des objets sérialisables (section 4.1.11). Le mot-clé `strictfp` sera expliqué dans la section 4.1.13. Les mots-clés `synchronized` et `volatile` concernent la programmation parallèle en Java et seront expliqués en section 6.1.2.1. Le mot-clé `synchronized` ne peut s'appliquer qu'à une méthode, alors que `volatile` ne peut s'appliquer qu'à un champ. Enfin, le mot-clé `native` ne s'applique qu'aux méthodes et permet de déclarer qu'une méthode sera native et définie en C ou C++. Ceci sera décrit plus en détail dans la section 9.1.

4. Cet autre constructeur devra lui aussi faire appel soit à un constructeur de sa super-classe, soit à un autre constructeur de sa classe. Le compilateur détecte les cycles dans les appels aux constructeurs de sa propre classe, mais par contre la JVM, et plus particulièrement son vérificateur de *bytecode*, ne le vérifie pas.

5. En termes de *bytecode*, ils font partie de la méthode `<clinit>`.

4.1.1.1 Risques

Aucun risque à signaler concernant la notion de classe. Les risques liés aux modificateurs seront présentés dans chacune des sous-parties les présentant.

4.1.1.2 Recommandations

Nous n'avons pas de recommandations particulières à faire ici. Les recommandations liées aux modificateurs seront présentées dans chacune des sous-parties les présentant.

4.1.2 Modificateurs de visibilité

Les mots-clés `private`, `protected`, `public` sont des modificateurs de visibilité auxquels s'ajoute le modificateur par défaut. Un modificateur de visibilité indique, pour un ensemble de classes organisé en packages, depuis quelles classes une entité peut être référencée.

Classes

Une classe peut être déclarée `public`, ce qui veut dire que la classe peut être référencée depuis tout autre *package*. Si aucun modificateur de visibilité n'est indiqué, la classe est uniquement visible à l'intérieur du package dans lequel elle est définie. Il est alors impossible par exemple de créer un objet de cette classe ou de transtyper un objet vers cette classe à l'extérieur du *package* d'appartenance.

Il est possible (même si cela est déconseillé dans la majeure partie des cas) d'encapsuler la déclaration d'une classe à l'intérieur d'une autre déclaration de classe, on les appelle des classes internes (ou *inner class*). Dans ce cas, il sera également possible d'appliquer, à cette classe interne, les modificateurs de visibilité `private` ou `protected`. Dans le premier cas, la classe interne ne pourra être utilisée que par la classe dans laquelle elle est déclarée. Dans le cas où le modificateur `protected` est utilisé, cette classe interne pourra être utilisée dans toutes les classes qui héritent de (ou qui sont du même *package* que) celle qui encapsule la classe interne.

Méthodes et champs

L'accès aux membres d'un objet depuis un autre objet est contrôlé lors de la compilation et lors du chargement du *bytecode* par les modificateurs de visibilité associés à chaque membre d'une classe. Ils sont au nombre de quatre :

- `public` : le membre est visible depuis n'importe quelle classe, quel que soit son *package* ;

- `protected` : le membre est visible par toutes les classes du même *package* et par celles qui héritent de la classe déclarant ce membre, sous certaines conditions (voir plus bas) ;
- (*package*) le modificateur par défaut. Les membres ne sont visibles que par les classes du *package* qui contient la classe ;
- `private` : un membre privé, même s'il est hérité par toutes les sous-classes de sa classe, n'est visible que dans la classe dans laquelle il a été déclaré.

Il faut noter que ces modificateurs ne permettent pas de spécifier qu'un champ est uniquement accessible en lecture ou en écriture : un champ visible peut être lu et écrit. Le modificateur `final` permet d'interdire la réécriture d'un champ. Un contrôle d'accès plus fin doit être programmé à l'aide des méthodes d'accès et des permissions (voir 6.2).

Il faut également noter que le modificateur `private` ne veut pas dire que le membre est uniquement accessible depuis son objet d'appartenance. Tous les objets de la même classe peuvent accéder à un champ privé d'un objet de cette classe s'ils possèdent une référence vers celui-ci.

Le code suivant illustre ce phénomène :

```
public class Account{
    private int balance;

    public Account(int balance){this.balance = balance;}

    public void debit (Account account){account.balance -= 100;}

    public static void main(String[] args){
        Account account1 = new Account(100);
        Account account2 = new Account(100);
        account1.debit(account2);
        System.out.println("Balance␣account2␣:" + account2.balance);
    }
}
```

La sémantique exacte du modificateur `protected` est plus compliquée qu'indiqué ci-dessus. Afin d'expliciter cette notion plus clairement, nous allons nous référer à la figure 3. Les classes `A` et `E` sont déclarées dans le *package* `package1`, `B` et `F` dans le *package* `package2`, `C` dans le *package* `package3` et `D` dans le *package* `package4`. Concernant les relations d'héritage, `B` et `D` héritent de `A` et `C` hérite de `B`. La classe `A` déclare un champ `protected int val;`.

1. toutes les sous-classes d'une classe `A` peuvent accéder aux membres `protected` déclarés dans la classe `A` sur leurs instances et celles de leurs sous-classes. Donc, dans l'exemple, `A` peut accéder au champ `val` des instances de `A`, `B`, `C` et `D` ; `B` peut accéder à celui de `B` et `C` ; et les classes `C` et `D` ne peuvent accéder qu'au champ `val` de leurs propres instances.
2. toutes les classes appartenant au même *package* que la classe `A` peuvent accéder aux membres `protected` déclarés dans la classe `A` sur les instances de `A` et sur celles

des sous-classes de `A`. Dans l'exemple, la classe `E` peut accéder au champ `val` des instances des classes `A`, `B`, `C` et `D`.

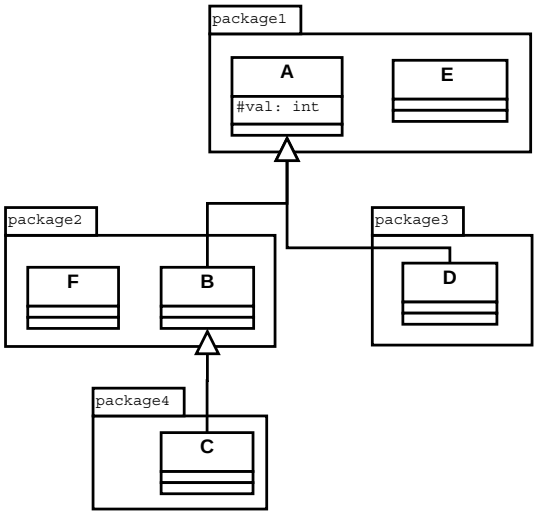


FIGURE 3 – Hiérarchie de classe

Il faut noter que le second point ne s'applique qu'aux classes appartenant au même *package* que la classe ayant déclaré le membre `protected`. Dans l'exemple de la figure 3, la classe `F` ne peut accéder ni au champ `val` des instances de la classe `B`, ni à ceux d'une autre classe héritant de `A`.

Enfin, il est important de noter que les mécanismes de réflexion (voir section 4.1.12) permettent d'accéder aux champs quels que soient les modificateurs qui y sont apposés et de les modifier. Le seul moyen pour se prémunir de l'utilisation de la réflexion est dans l'utilisation d'un *Security Manager*.

4.1.2.1 Risques

- Donner une trop large visibilité aux membres d'une classe (surtout les champs), ce qui permet à d'autres applications d'accéder à des données à protéger ou de les modifier.
- Mauvaise utilisation du niveau de visibilité *protected* dont la sémantique est plus complexe que celle des trois autres niveaux.

4.1.2.2 Recommandations

- Réduire au maximum la visibilité des champs d'une classe. Privilégier des champs privés avec des méthodes pour y accéder (des *getters* et *setters*) qui peuvent effectuer des vérifications dynamiques de contrôle d'accès et de validité des valeurs.
- Sceller les *packages* pour éviter que des classes malveillantes puissent se déclarer membre du *package* et ainsi gagner une meilleure visibilité sur les membres (décla-

rés `protected` ou (*package*)) des classes du *package*. La fermeture d'un *package* consiste, dans un fichier `JAR`, à mettre les fichiers `.class` accompagnés d'un fichier `Manifest.MF` où l'attribut *sealed* est à la valeur `true`. Ainsi, aucune autre classe ne peut se déclarer du même *package* et être chargée sans lever une exception de sécurité (`java.lang.SecurityException: sealing violation`).

– Considérer les méthodes `protected` comme des méthodes publiques qui peuvent être redéfinies par des sous-classes externes (ce qui entre autre permet de contourner des contrôles d'accès effectués dans ces méthodes).

4.1.3 Membres statiques

Un champ déclaré statique (mot-clé `static`) est un champ partagé et accessible par toutes les instances d'une classe. Il fonctionne comme une variable globale. Les champs statiques peuvent être référencés soit par le nom de la classe, soit au travers d'un objet, instance de la classe. Il est cependant conseillé d'y accéder en utilisant le nom de la classe pour des raisons de clarté et donc de maintenabilité.

Les méthodes statiques s'apparentent de la même façon à des fonctions globales qui peuvent être appelées en donnant la classe de définition et leur nom. Par exemple, la méthode statique `f` définie dans la classe `C` est appelée par l'expression `C.f()`.

Les modificateurs de visibilité présentés en section 4.1.2 peuvent s'appliquer aux champs et méthodes statiques pour éventuellement restreindre leur visibilité.

4.1.3.1 Risques

Les champs statiques publics sont plus facile d'accès que les champs d'un objet puisqu'ils ne nécessitent pas d'avoir une référence vers l'objet (mais juste de connaître le nom de la classe). Un tel niveau d'accès est uniquement souhaité pour des constantes (par exemple les constantes mathématiques utilisées dans une application). Un champ `public static` qui n'est pas déclaré `final` peut être modifié par n'importe quel code.

Le seul moyen pour restreindre l'accès à un champ statique est de restreindre sa visibilité en utilisant des modificateurs d'accès (voir 4.1.2). Si un contrôle d'accès plus fin est souhaité, il est nécessaire de passer par des méthodes d'accès (*getters* et *setters*) qui contrôlent l'accès à cette variable.

Il y a un risque potentiel lié à la mauvaise compréhension du mot-clé `static` : il ne signifie pas que la valeur du champ ne change pas. Pour obtenir cette propriété, le champ doit être déclaré `final`.

4.1.3.2 Recommandations

- Ne pas stocker des données sensibles dans des champs statiques.
- Déclarer les constantes (mathématiques ou autres) d'une application comme `static final` et effectuer leur initialisation lors de la déclaration.
- Rendre privé les autres champs statiques et ajouter des méthodes d'accès (*getters* et *setters*) qui contrôlent l'accès à cette variable.

4.1.4 Les champs `final`

Le modificateur `final` s'applique à des classes et à ses membres, avec une sémantique différentes dans les deux cas. La sémantique du modificateur `final` appliqué sur les classes ou sur les méthodes sera traitée dans la section 4.1.7. Dans cette section, nous ne traiterons que la sémantique du modificateur `final` appliqué aux champs (statiques ou non) d'une classe.

Champs d'instance : un champ déclaré `final` fonctionne comme une constante d'objet. Chaque instance de cette classe a une constante qui peut être affectée au plus une fois. Cette affectation doit avoir lieu obligatoirement lors de la construction de l'objet. Différentes instances d'une même classe peuvent alors donner différentes valeurs à cette constante.

Champs statiques : un champ déclaré `final static` fonctionne comme une constante partagée par toutes les instances de la classe. Comme pour les champs de classe, ce champ ne peut être affecté qu'une seule fois, au moment de sa déclaration ou au sein d'un bloc `static{...}`.

Le modificateur `final` peut également être utilisé au sein d'une méthode soit dans la déclaration d'une variable locale, soit dans celle d'un paramètre de la méthode. Dans le premier cas, il n'est pas obligatoire d'initialiser une variable locale `final` au moment de sa déclaration, mais par contre cette variable ne pourra être affectée qu'une seule fois durant une exécution de la méthode (par contre, la variable peut avoir une valeur différente à chaque exécution de la méthode). Un paramètre de méthode déclaré `final` ne peut être redéfini dans le corps de la méthode. Ce modificateur devient, par contre, obligatoire quand on veut utiliser un paramètre ou une variable locale au sein d'une classe anonyme qui serait définie dans cette méthode, comme le montre l'exemple suivant.

```
public class A {
  public void m(final int val, String ch){
    final String str = "str␣doit␣être␣final␣" + ch;
    Object obj = new Object(){
      int f = val;

      public String toString(){
        return str + "␣et␣f␣=␣" + f;
      }
    };
```

```
    System.out.println("obj: " + obj);
  }
}
```

Dans l'exemple précédent, la variable `obj` contient un objet dont la classe est anonyme. Une classe anonyme est une classe dont on ne définit pas de nom et qui, par conséquent, ne peut être instanciée que pour un seul et unique champ ou variable (excepté en utilisant les mécanismes de réflexion décrits en section 4.1.12). Pour que la classe `A` puisse être compilée, il faut absolument que le paramètre `val` de la méthode `m` et la variable locale `str` soient déclarés `final`.

4.1.4.1 Risques

– Utiliser le modificateur `final` sur des champs contenant des données confidentielles. Si c'est le cas, il n'est plus possible d'écraser la donnée quand elle n'est plus utile.

4.1.4.2 Recommandations

– Déclarer les champs `final` quand on veut garantir qu'ils ne seront pas modifiés (cf. section 4.1.10).
– Ne pas utiliser le modificateur `final` sur les champs contenant des données confidentielles (cf. section 4.1.8).

4.1.5 Interfaces et classes abstraites

Les interfaces permettent de déclarer une collection de constantes et méthodes. La déclaration d'une méthode consiste en une signature de la méthode *i.e.* son nom, les types des arguments et le type du résultat. Les constantes définies dans une interface disposent nécessairement des modificateurs `final static`. S'ils ne sont pas mis dans le source, ils sont automatiquement ajoutés par le compilateur Java. Au sens strict, on ne peut utiliser le terme « constante » que pour les champs d'une interface ayant pour type un des types primitifs ou une classe immuable (voir section 4.1.10). En effet, si une constante déclarée dans une interface a pour type une classe mutable, les champs de cette classe mutable pouvant être modifiés, le qualificatif de « constante » n'est pas adapté.

Une classe **implémente** une interface si elle fournit une implémentation des méthodes listées dans l'interface. Les interfaces peuvent être utilisées comme type d'une variable et il est alors possible d'affecter des objets à cette classe s'ils sont d'une classe qui implémente l'interface.

Entre les notions de classe et d'interface se trouvent les **classes abstraites** qui sont des

classes dont une ou plusieurs méthodes sont déclarées (*i.e.* leurs signatures sont spécifiées) mais pas définies. Ces méthodes sont des **méthodes abstraites** et leurs déclarations doivent être préfixées par le mot-clé `abstract`. La définition d'une classe abstraite doit être préfixée par le mot-clé `abstract` et ne peut pas être instanciée. Une classe héritant d'une classe abstraite devra soit implémenter toutes ses méthodes abstraites, soit être elle-même déclarée abstraite.

4.1.5.1 Risques

La vérification du bon typage, que ce soit pour les interfaces ou pour les classes abstraites, n'est pas effectuée statiquement pour des raisons d'efficacité (même si cela est théoriquement possible), mais lors de l'exécution. Une JVM doit donc implémenter cette vérification pour éviter des erreurs de typage.

4.1.5.2 Recommandations

Aucune recommandation particulière si ce n'est d'utiliser le plus possible les classes abstraites et les interfaces. Les interfaces sont très utiles dans la structuration de programmes Java et permettent de séparer l'implémentation des méthodes de leur déclaration. En revanche, leur vérification doit être assurée par la JVM.

4.1.6 Classes internes

Il est possible de définir une classe à l'intérieur d'une autre classe, soit comme un champ d'une classe soit à l'intérieur d'un bloc de code d'une méthode. Dans ce dernier cas, on parle de classe anonyme qui empêche donc son utilisation en dehors de son contexte de définition.

Nous ne revenons pas ici sur les motivations qui justifient l'utilisation des classes internes en termes de structuration de code et bonnes pratiques de génie logiciel. D'un point de vue de la sécurité, nous mentionnons seulement que les classes internes ont été signalées comme une source de problèmes de sécurité lors de la compilation vers du *bytecode* Java. En effet, la notion de classe interne n'existe pas au niveau *bytecode*, il est donc nécessaire de rendre une classe interne globale lors de sa compilation en *bytecode*. Or, pour donner accès aux membres de la classe englobante à la classe interne, il est parfois nécessaire de rendre des membres privés accessibles à l'intérieur du *package* où sont définies les deux classes.

4.1.6.1 Risques

L'impact sécuritaire de l'utilisation de ce type de structure reste encore un sujet de recherche. En effet, même s'il est considéré comme une bonne pratique de ne pas utiliser ce genre de structure, nous n'avons trouvé aucune information concernant d'éventuelles failles de sécurité.

4.1.6.2 Recommandations

Ne pas utiliser des classes internes quand cela n'est pas réellement nécessaire.

4.1.7 Héritage

Une classe peut hériter au plus d'une autre classe. La relation d'héritage (ou *inheritance*) et le fait qu'il n'y a pas d'héritage multiple en Java imposent une structure arborescente sur les classes, appelée la hiérarchie de classes. Au sommet de la hiérarchie se trouve la classe prédéfinie `java.lang.Object` qui est une super-classe de toutes les autres classes. Une classe peut faire référence à sa super-classe soit en la nommant explicitement, soit en utilisant le mot-clé `super`. Par exemple, le constructeur (voir 4.1.1) de la super-classe peut être appelé par `super();` à partir d'un constructeur de la sous-classe.

Une classe hérite de l'ensemble des membres de sa super-classe, mais ne « voit » et donc ne peut utiliser au sein de ses méthodes que ceux qui ne sont pas déclarés `private` (ou (*package*) si la sous-classe ne fait pas partie du même *package*). La classe peut également définir ses propres membres (champs et méthodes).

Pour interdire l'héritage et donc l'extension d'une classe, il suffit de la déclarer `final`.

En ce qui concerne les champs, il est possible de définir un champ ayant le même nom qu'un champ dans sa super-classe. Ce champ masque alors le champ de la super-classe qui fait néanmoins partie de l'objet. Le champ de la super-classe est accessible à l'aide du transtypage (*cast* en anglais). Soient `A` et `B` des classes déclarant un champ `f` avec `B` une sous-classe de `A`. Avec `b` un objet de classe `B`, l'expression `((A)b).f` donne accès au champ `f` de la classe `A` malgré le masquage par le champ `f` de la classe `B` de l'objet `b`. Il est cependant préférable d'éviter de masquer des champs pour faciliter la compréhension et la maintenance du code.

En ce qui concerne les méthodes, une classe a la possibilité de redéfinir une méthode héritée visible, en fournissant sa propre définition d'une méthode avec les mêmes nom et paramètres. Lors de la redéfinition, il est possible d'augmenter la visibilité d'une méthode (par exemple de changer son niveau de `protected` à `public`). En revanche, il n'est pas possible de diminuer sa

visibilité.

À cause de l'héritage de méthodes, il n'est pas immédiat de déterminer quelle méthode sera effectivement appelée lors de l'appel d'une méthode `x.f()`. La variable `x` référence un objet et c'est la classe de cet objet qui détermine quelle méthode sera effectivement appelée. La procédure pour déterminer quelle méthode sera appelée s'appelle « résolution d'appels virtuels » (ou *virtual method call resolution*). La résolution d'appels virtuels cherche, à partir de la classe de l'objet, dans les super-classes en montant vers la racine de la hiérarchie jusqu'à trouver une définition d'une méthode avec le nom `f` et les bons paramètres.

L'héritage et la redéfinition de méthodes peuvent être source de failles de sécurité si une sous-classe arrive à redéfinir des méthodes faisant des vérifications de sécurité. En effet, si une classe utilise une méthode particulière pour effectuer des vérifications de sécurité, une sous-classe peut redéfinir cette méthode (par exemple en une méthode qui n'effectue aucune vérification), à moins que la classe n'ait interdit la redéfinition de la méthode en la déclarant `final`. Dans l'exemple, ci dessous, le message `Permission granted : BadGuy` est affiché lors de l'exécution.

```
class A{
  public void service(String client){
    if (authorized(client))
      System.out.println("Permission␣granted␣:␣" + client);
    else
      System.out.println("Permission␣denied␣:␣" + client);
  }

  public boolean authorized(String client){
    return client.equals("GoodGuy");
  }
}

class B extends A{
  public boolean authorized(String client){
    return true;
  }
}

public class Redefinition{
  public static void main(String[] args){
    B b = new B();
    b.service("BadGuy");
  }
}
```

Une méthode (de classe ou d'instance) peut être déclarée `final`. Pour les méthodes non-privées, cela signifie qu'il est impossible pour une sous-classe de redéfinir l'implantation de

cette méthode. La méthode sera forcement héritée dans toutes les sous-classes. Pour les méthodes privées, le mot-clé `final` ne « sert » à rien puisqu'elles ne peuvent être surchargées. En effet, même si les sous-classes héritent de ces méthodes privées, elles ne peuvent ni les utiliser directement, ni les surcharger ; les méthodes privées de la super-classe ne peuvent être appelées que par les méthodes de la super-classe. Toutefois, rien n'empêche une classe de définir une méthode possédant la même signature qu'une méthode privée de sa super-classe. L'exemple suivant illustre ce propos.

```
public class A{
  public A(){
    super();
    print();
  }
  private final void print(){
    System.out.print("A");
  }
}
```

```
public class B extends A{
  public B(){
    super();
  }

  public void print(){
    System.out.print("B");
  }
}
```

4.1.7.1 Risques

- Redéfinition de méthodes critiques par des sous-classes malveillantes.
- Perte de clarté et donc ajout potentiel d'erreurs dans le code en cas de masquage de champ.

4.1.7.2 Recommandations

- Utiliser le mot-clé `final` pour interdire la redéfinition des méthodes ou des classes importantes pour la sécurité d'une application.
- S'il n'est pas possible d'interdire la redéfinition des méthodes qui sont importantes pour la sécurité, une analyse (manuelle ou automatique) du code doit vérifier que les méthodes des super-classes effectuant des contrôles de sécurité sont appelées.
- Ne pas masquer les champs hérités en déclarant des champs du même nom.

4.1.8 Gestion de la mémoire

Java permet d'allouer dynamiquement des zones de mémoires. Seuls les types référence peuvent être générés dynamiquement, ce qui englobe deux cas : les tableaux et les objets. Pour les objets, la création se fait en exécutant l'instruction `new C(arg1,..,argn)` qui alloue un nouvel objet de la classe `C`, exécute le **constructeur** de `C` ayant pour paramètres `arg1,...,argn` et rend comme résultat une référence à cet objet. Pour les tableaux, la création se fait en exécutant l'instruction `new C[dim1]..[dimN]` où `C` est la classe déclarée des éléments que doit

contenir le tableau, et les variables `dim1` à `dimN` sont des entiers indiquant les dimensions du tableau. Dans le cas des tableaux, on ne fait pas appel à un constructeur ; la création d'un tableau ne définit que les dimensions du tableau et le type déclaré des éléments que va pouvoir contenir ce tableau. Par contre, pour chaque élément du tableau, un constructeur sera appelé pour créer ces objets.

Les champs d'un objet alloué sont tous mis à zéro (ou à `null` si ce sont des références) avant d'être initialisés par le constructeur (cf section 4.1.1) de la classe. L'initialisation d'un objet est terminée une fois que le constructeur a fini son exécution.

Contrairement à des langages comme C, Java limite fortement les opérations qu'on peut effectuer sur une référence. En particulier, il est impossible d'utiliser « l'arithmétique de pointeurs » connue de C. Il est également impossible de créer une référence mémoire à partir d'un entier ou d'obtenir l'adresse mémoire d'un objet. Seul le test de l'égalité sur des références reste possible.

Même s'il est impossible d'obtenir l'adresse mémoire d'un objet, il est néanmoins possible d'obtenir des informations indirectes sur celle-ci, à travers la méthode `hashCode()` de `java.lang.Object` qui rend une valeur de *hachage* d'un objet, souvent calculée à partir de l'adresse mémoire de l'objet. [6]

L'utilisation de références mène potentiellement à des confusions entre une référence et une copie d'un objet. Par exemple, l'affectation d'une variable contenant un objet à une autre variable (`x = y;`) ne copie pas l'objet référencé par `y`, mais la référence stockée dans `y`. Il en est de même pour une référence passée en argument à une méthode.

Contrairement à la création d'un objet, il n'est pas possible de demander explicitement la **destruction** (ou la libération de la mémoire) d'un objet. Un objet qui n'est plus utilisé (*i.e.*, référencé par une variable ou par un autre objet) reste dans la mémoire de la machine jusqu'au moment où le *garbage collector* est activé par la machine virtuelle et réclame la mémoire occupée par l'objet. La notion de *garbage collection* (qui ne fait pas partie de la définition du langage Java) est traitée plus en détail dans [14] et [11].

Une classe peut redéfinir la méthode `finalize` de la classe `java.lang.Object` pour spécifier des actions à effectuer lors de la destruction de l'objet. Toutefois il ne s'agit pas d'une méthode qui peut être appelée pour forcer la destruction d'un objet. Par contre, en redéfinissant la méthode `finalize`, un attaquant peut empêcher un objet d'être détruit par le *garbage collector* en référencant l'objet sur le champ statique d'une classe par exemple. C'est le même principe que celui présenté dans l'exemple suivant. Par contre, dans l'exemple suivant, en redéfinissant la méthode `finalize`, ne sachant pas quand le *garbage collector* est appelé, il est donc nécessaire d'attendre que le champ statique référence l'objet.

6. Le hachage implanté par la méthode `hashCode()` sert à construire des tables de hachage. La spécification de cette méthode ne stipule cependant pas que la fonction de hachage implantée possède des propriétés de sécurité qui garantissent qu'il s'agit d'une fonction à sens unique.

Dans l'exemple suivant, le constructeur de la classe réalise un appel à la méthode `check()` pour vérifier une permission donnée. Si cette permission est refusée, une exception de sécurité est levée. Dans ce cas, cette exception sera transmise à l'appelant qui ne recevra pas la référence sur le nouvel objet. La vérification de permission a bien permis d'empêcher l'appelant d'obtenir une instance de la classe. Il est possible de contourner cette vérification en ayant recours à une forme d'échappement. La classe `B` redéfinit la méthode `check()` en ajoutant une copie de la référence à l'objet courant dans un champ statique (accessible globalement). Ainsi, quel que soit le résultat de la vérification de permission, la référence sur le nouvel objet est accessible. La méthode `main()` de la classe `Relax` montre comment l'appelant peut accéder à cette référence quel que soit le résultat.

```
class A{
  private boolean initialized = false;

  public A(){
    check();
    initialized = true;
  }
  public void check() throws SecurityException {
    System.getSecurityManager().checkPermission(...);
  }
}

class B extends A{
  public static B escape;
  public void check() throws SecurityException {
    escape = this;
    super.check();
  }
}

public class Relax{
  public static void main(String[] args){
    A a;
    try{
      a = new B();
    }
    catch (Exception e){
      a = B.escape;
    }
  }
}
```

À noter que ce comportement n'est possible que parce que le constructeur utilise une méthode qui peut être redéfinie. En particulier, il ne serait pas possible de réaliser l'échappement directement dans le constructeur de `B`, car l'appel au constructeur de la classe parent ou à un autre constructeur de la même classe doit être la première instruction exécutée. Par rapport à

l'exemple de la section précédente (sur l'héritage), cet exemple montre que, même si la nouvelle classe effectue le test de sécurité, l'objet peut être accessible.

4.1.8.1 Risques

- La surcharge d'une méthode appelée dans un constructeur peut conduire à un contournement de la politique de sécurité.
- Le mécanisme de libération de la mémoire (*garbage collector*) nest pas fiable s'agissant d'effacer des données confidentielles.

4.1.8.2 Recommandations

- Dans un constructeur, ne pas appeler des méthodes qui peuvent être surchargées.
- Afin de garantir l'effacement des données confidentielles, il est préférable de coder une méthode qui écrasera les données confidentielles sur les types primitifs [7] et qui sera appelée récursivement sur les types références [8]. Cette méthode devra être appelée explictement dès lors que les données confidentielles ne sont plus nécessaires.

4.1.9 Références et copies d'objets

Les types de données Java se divisent en deux catégories :

- les types primitifs : entiers (`byte, int, short, long`), flottants (`float, double`), booléen (`boolean`) et caractère (`char`) ;
- les types références (classes, interfaces et tableaux).

Il est important de comprendre que les valeurs stockées dans une variable de type objet (qu'il soit typé par une classe ou une interface) ou tableau sont en réalité des références vers des zones mémoire où sont stockées les données composant l'objet ou le tableau. Ainsi, l'effet d'une affectation de la valeur d'une variable `x` de type objet (ou tableau) à une autre variable `y` est que la référence vers l'objet ou le tableau sera copiée dans `y`. L'objet (ou le tableau) n'est pas copié. La même remarque est pertinente pour les appels de méthode (`x.foo(arg)` où `arg` est de type objet). La méthode reçoit une référence vers l'objet référencé par `arg`, pas une copie de cet objet. De ce fait, la méthode appelante et la méthode `foo` ont toutes les deux accès au même objet.

Tout code qui détient une référence à un objet peut potentiellement lire et modifier cet objet, ce qui pose des problèmes en termes de confidentialité et d'intégrité. Pour éviter d'avoir à par-

7. Il est préférable de ne pas utiliser `final` sur des variables contenant des données confidentielles.
8. Il est préférable de ne pas utiliser des classes immuables (voir section 4.1.10) pour contenir des données confidentielles.

tager la référence à un objet avec du code extérieur, il est possible de faire des copies explicites de tout ou partie d'un objet, en utilisant le **clonage** d'objets ou de tableaux.

De manière générale, on distingue deux types de copies d'un objet, en surface (*shallow copy*) et en profondeur (*deep copy*). Dans le cas de la copie superficielle, la copie des attributs d'un objet de type référence est faite par référence. L'objet et sa copie partagent les valeurs (de type référence) accessibles par leurs attributs. Dans le cas de la copie en profondeur, la copie des attributs est récursive, les attributs du nouvel objet reçoivent des références vers des copies des valeurs accessibles depuis les attributs de l'objet original. Il est possible de mélanger les deux façons de copier un objet, par exemple en effectuant une copie en profondeur uniquement de la partie d'un objet qui contient des données à protéger. Parfois, cela est même nécessaire comme dans le cas de structures cycliques où une copie purement en profondeur provoquerait une boucle infinie d'appels à la méthode `clone()` de la classe représentant l'élément de base de la structure cycle. Cette boucle infinie se traduirait finalement par un Stack Overflow.

Dans le cas de Java, la méthode `clone()` qui est responsable du clonage (copie) est une méthode de la classe `java.lang.Object`. Tout objet de type référence dispose donc de cette méthode qui retourne une copie superficielle de l'objet sur lequel elle est appelée. Par défaut un objet de type quelconque n'a pas le droit d'appeler cette méthode (celle de la classe `java.lang.Object`). Pour autoriser les instances d'une classe à appeler cette méthode, la classe doit implémenter l'interface vide `Cloneable`. Si la méthode `clone()` de la classe `java.lang.Object` est appelée par une instance d'une classe n'implémentant par cette interface, l'exception `java.lang.CloneNotSupportedException` est levée.

Pour effectuer une copie en profondeur (totale ou partielle), il est nécessaire de redéfinir la méthode `clone()`. Par convention, cette méthode qui a le modificateur `protected` dans la classe `java.lang.Object` doit être redéfinie avec une visibilité `public`. La redéfinition de la méthode `clone()` peut soit contenir un appel à `super.clone()` ce qui aura pour effet de créer une copie de l'objet en surface de l'objet, soit recréer un nouvel objet en appelant un des constructeurs de la classe et en appelant les méthodes pour mettre les champs à la valeur souhaitée et en utilisant ou non les mêmes références d'objet. Dans le premier cas, la redéfinition de la méthode doit alors gérer la copie en profondeur pour les éléments souhaités.

Remarque 1 *Il faut noter le caractère particulier de l'interface `java.lang.Cloneable` qui ne sert que d'étiquette à la classe qui l'implémente. En effet, la méthode `clone()` est héritée de la classe `java.lang.Object`, mais elle ne peut être utilisée qu'à partir du moment où la classe qui doit hériter de la méthode, implémente l'interface `java.lang.Cloneable`, sinon une exception `CloneNotSupportedException` est levée. La vérification de l'implémentation de l'interface `java.lang.Cloneable` et la levée de l'exception sont codées dans la méthode `clone()` de la classe `Object`. Il est donc possible de contourner ce comportement en redéfinissant la méthode `clone()` sans que celle-ci appelle celle de la classe `Object`.*

Dans l'exemple ci-dessous, la méthode `clone()` de la classe `A` effectue une copie en profondeur. En revanche, la classe `B` redéfinit ce comportement et n'effectue plus de copie du tout. On

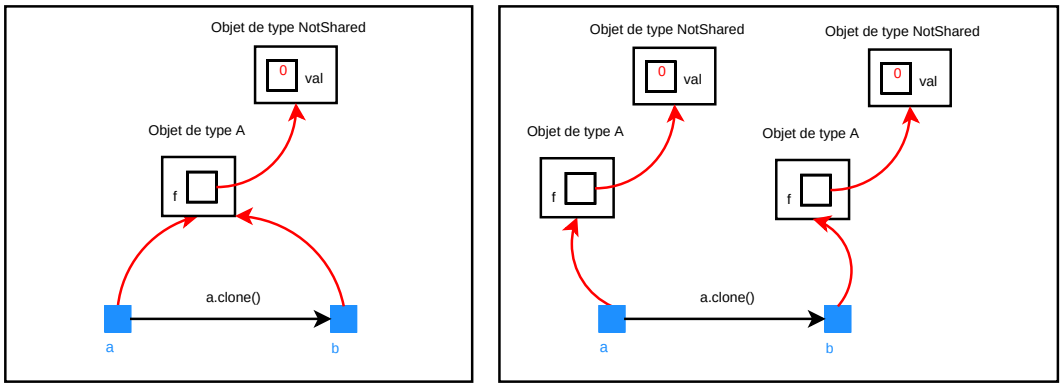
montre ainsi qu'un appel à la méthode `clone()` ne garantit pas d'obtenir des objets différents.

```
class NotShared implements Cloneable{
  int val = 0;
  public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }
}


class A implements Cloneable{
  NotShared f;
  public Object clone() throws CloneNotSupportedException {
    A a = (A) super.clone();
    a.f = (NotShared) f.clone();
    return a;
  }
}

class B extends A{
  public Object clone() throws CloneNotSupportedException{
    return this;
  }
}
```

Les figures 4(a) et 4(b) montrent le résultat d'une opération `clone()` sur un objet respectivement de type `B` et de type `A`. L'exécution d'un test `a==b` (teste l'égalité des deux références) sur ces figures donnerait respectivement les résultats `true` et `false`.



(a) Clonage d'un objet de type B (b) Clonage d'un objet de type A

FIGURE 4 – Clonage

À noter dans cet exemple, la nécessité de redéfinir la méthode `clone()` avec une visibilité `public` dans la classe `NotShared`, sans quoi l'appel `f.clone()` dans la classe `A` n'aurait pas été possible.

4.1.9.1 Risques

En passant une référence à un objet au lieu d'une copie stockée (par exemple dans un champ privé), il est alors possible de donner accès à ce champ en dehors de la classe.

4.1.9.2 Recommandations

Lors d'un échange d'objets avec un code externe, il faut privilégier la copie (complète ou partielle) d'objets afin d'éviter de partager des références à des zones mémoires avec des données à protéger.

4.1.10 Objets immuables

Les objets et les tableaux sont dits *mutables*, parce qu'il est possible de modifier leur état lors de l'exécution, en affectant une nouvelle valeur aux champs (pour les objets) ou en remplaçant un élément par un autre dans un tableau.

Cependant, certaines classes fournies par la bibliothèque standard de Java sont immuables, c'est-à-dire que la valeur des objets instanciant ces classes est fixée à la création de l'objet et ne change plus en cours d'exécution. Par exemple, on peut citer les classes servant de *wrappers* aux types primitifs : `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character` et `Boolean`. Parmi les classes immuables, on peut également citer la classe `String` qui correspond à une chaîne de caractères Unicode.

Le caractère immuable n'est pas une propriété intrinsèque de certaines classes de la bibliothèque standard, mais il est obtenu en implémentant les classes d'une certaine manière. Il est même considéré comme une bonne pratique de génie logiciel Java [34] de rendre une classe d'objets immuable quand c'est possible. Pour cela, il suffit de suivre les règles suivantes :

- rendre tous les champs `final` et `private` ;
- rendre la classe `final` pour empêcher qu'elle soit étendue ;
- ne pas fournir des méthodes qui modifieraient l'état de l'objet ;
- ne pas fournir les références des champs qui sont typés avec des classes mutables, mais fournir des copies profondes de ces champs.

Un avantage de cette approche est qu'il n'est pas possible de créer des *data races* entre *threads* sur des objets immuables (voir section 6.1).

Pour finir, nous devons tout de même préciser deux points. Le premier est qu'au sens strict, `String` n'est pas vraiment immuable. En effet, cette classe définit un champ `hash` qui n'est pas `final` et qui sert à stocker la valeur de hachage de l'objet quand sa méthode `int hashCode()`

est appelée. Avant l'appel à cette méthode, ce champ reçoit la valeur 0 et après le premier appel, il prend définitivement la valeur calculée par cette méthode. Comme le calcul de la valeur de hachage peut être lourd si la chaîne de caractères est longue et que cette méthode n'est pas forcément appelée, l'utilisation de ce champ constitue une optimisation. D'un côté, cela évite de calculer la valeur si ce n'est pas nécessaire, et d'un autre côté, cela évite de recalculer à chaque fois cette valeur. Il faut toutefois noter que la chaîne de caractères ne peut pas être modifiée (excepté dans le cas évoqué sur le point suivant).

Le second point concerne l'utilisation des mécanismes de réflexion. Le caractère immuable d'une classe est obtenue en verrouillant l'accès aux champs d'une classe ou en empêchant que ceux-ci puissent être modifiés. Cependant, les mécanismes de réflexion permettent d'accéder aux champs quels que soient les modificateurs qui y sont apposés et de les modifier. Le seul moyen pour se prémunir de l'utilisation de la réflexion est dans l'utilisation d'un *Security Manager* et d'empêcher les classes non sûres (*untrusted*) d'utiliser ces mécanismes.

4.1.10.1 Risques

Un effet indésirable de l'immuabilité des objets est que des données confidentielles comme les mots de passe (qui sont souvent stockés comme des chaînes de caractères) ne peuvent pas être effacées de la mémoire par le programme qui les utilise.

4.1.10.2 Recommandations

Privilégier des classes mutables pour les champs contenant des données confidentielles. Par exemple, pour le cas des mots de passe, il vaut mieux utiliser la classe `StringBuffer` que la classe `String`, car elle permet une mise à zéro de son contenu.

4.1.11 Sérialisation

Java offre la possibilité d'obtenir une représentation concrète sous forme d'une suite d'octets d'un objet, notamment pour pouvoir communiquer l'objet à travers un réseau ou pour améliorer la persistance d'une application. On dit alors que l'objet a été sérialisé. Pour que cette opération soit possible, il faut que la classe de l'objet soit déclarée *sérialisable*, c'est-à-dire il faut qu'elle implémente l'interface vide `java.io.Serializable`.

La classe `java.io.ObjectOutputStream` contient une méthode `writeObject` qui permet de créer une représentation binaire d'un objet. Par défaut, la sérialisation d'un objet contient l'état interne de l'objet, c'est-à-dire la valeur de l'ensemble de ces champs non statiques, qu'ils soient `public`, `protected`, (*package*) ou `private`. Pour récupérer cet état interne, la classe

`java.io.ObjectOutputStream` utilise les mécanismes de réflexion (cf. section 4.1.12), ce qui lui permet d'accéder aux champs non `public`.

La classe `java.io.ObjectInputStream` propose une méthode `readObject` qui permet de récupérer une représentation interne de l'objet sérialisé à partir d'une suite d'octets. Plus précisément, `readObject()` rend un objet de classe `java.lang.Object` qui doit être ensuite transtypé (par un *cast*) vers sa propre classe. Si le flux d'octets n'est pas conforme à ce que la méthode `readObject` attend, alors une exception est levée à l'exécution (soit `java.io.ClassNotFoundException`, soit `java.io.IOException`). De la même manière que pour le mécanisme de sérialisation par défaut, la désérialisation utilise les mécanismes de réflexion pour l'écriture des données récupérées dans la suite d'octets dans les champs de l'objet.

L'application du modificateur `transient` à un champ indique, aux mécanismes de (dé)sérialisation par défaut des classes `ObjectOutputStream` et `ObjectInputStream`, que le champ ne doit pas être (dé)sérialisé.

Cependant, il est possible de définir partiellement ou intégralement les algorithmes de (dé)sérialisation. Pour cela, il suffit de définir les méthodes suivantes [9] dans la classe devant être sérialisée :

```
private void writeObject(java.io.ObjectOutputStream out)
    throws IOException;
private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

Par exemple, si un champ, dont l'accès à la valeur est soumis à un contrôle d'accès, doit faire partie de la sérialisation d'un objet, il est préférable de rendre ce champ `transient`, et de définir, dans la classe devant être sérialisée, la méthode `writeObject` dans laquelle le contrôle d'accès est effectué avant de sérialiser cet objet dans la représentation binaire (cf figure 5).

De la même manière, puisque le mécanisme de désérialisation par défaut n'effectue pas de vérifications sur les invariants d'un objet reconstruit à partir d'un flux binaire (par exemple, si un objet représente bien une date ou une heure valide), il est préférable de définir la méthode `readObject` pour effectuer ces vérifications. Comme règle générale, une classe doit effectuer les mêmes vérifications au sein de sa méthode `readObject` que celles qui sont effectuées dans les constructeurs. Pour un exemple, voir section 5 ("Serialization and Deserialization") de [47].

Il faut également noter que, si une partie des champs a été sérialisée manuellement dans la méthode `writeObject` de la classe sérialisable, on doit absolument définir la méthode `readObject` pour les désérialiser de manière symétrique (cf. figure 5).

Comme pour d'autres formes d'échange d'objets (voir section 4.1.9), il est conseillé de ne pas récupérer des références à des objets provenant des zones mémoires inconnues, mais plutôt de créer des copies locales des objets.

9. Au sein de ces méthodes, le mot-clé `transient` n'empêche pas un champ d'être (dé)sérialisé.

```
class A implements java.io.Serializable{
  /** Attribut dont l'accès n'est pas contrôlé */
  private String str;
  /** Attribut dont l'accès est contrôlé */
  private transient int valAC;
  /** Numéro de version de la classe */
  private static final long serialVersionUID  = 42L;

  ... // Reste du code

  private void writeObject(java.io.ObjectOutputStream out)
                    throws IOException, AccessControlException{
    // Vérification du contrôle d'accès pour les champs dont l'accès
    // est contrôlé. Si l'accès n'est pas autorisé, renvoie de
    // AccessControlException
    securityCheckPermissions();

    // Sérialisation des champs non static et non transient (ex: str)
    out.defaultWriteObject();
    // Sérialisation manuelle de valAC (champ contrôlé)
    out.writeInt(valAC);
  }

  private void readObject(ObjectInputStream in)
                    throws IOException, ClassNotFoundException,
                           AccessControlException{
    // Vérification des permissions de création d'un objet de cette
    // classe. Si l'accès n'est pas autorisé, renvoie de
    // AccessControlException
    securityCheckPermissions();

    // Désérialisation des champs non static et non transient (ex: str)
    in.defaultReadObject();
    // Désérialisation de valAC sérialisé manuellement
    valAC = in.readInt();

    // Vérifications des invariants de la classe
    checkInvariants();
  }
}
```

FIGURE 5 – Exemple de classe implémentant `java.io.Serializable`

Il est également recommandé de définir, dans toutes les classes sérialisables, une constante `serialVersionUID` (dont la signature est celle présentée dans l'exemple de la figure 5). Cette constante, utilisée par les mécanismes de sérialisation par défaut, indique le numéro de version de la classe et doit être incrémentée au moins à chaque modification faite sur les signatures des champs de la classe. Ce champ a été introduit pour pouvoir garder une compatibilité des mécanismes de (dé)sérialisation sur différentes versions de la même classe. Ainsi, au moment d'une désérialisation, selon la version de la classe qui a été sérialisée, il sera possible d'identifier ce qui peut et ce qui doit être récupéré dans la suite d'octets. Si, par exemple, entre la version qui a été sérialisée et la version courante de la classe, des champs ont été ajoutés, il ne faudra pas essayer de les lire dans la suite d'octets et donner à ces champs une valeur par défaut.

Enfin, si l'on veut absolument maîtriser l'ensemble de la sérialisation sans utiliser les mécanismes de réflexion, il est possible d'implémenter l'interface `java.io.Externalizable` qui elle-même étend `java.io.Serializable`. En implémentant cette interface, la seule information sérialisée par le mécanisme de sérialisation par défaut est le nom de la classe. Le reste de la sérialisation doit être effectué manuellement en implémentant la méthode suivante :

```
public void writeExternal(ObjectOutput out) throws IOException
```

En ce qui concerne la désérialisation, le mécanisme de désérialisation par défaut désérialise le nom de la classe, ce qui lui permet d'invoquer le constructeur sans paramètre de la classe et ensuite d'appeler la méthode suivante dans laquelle l'algorithme de désérialisation doit être défini manuellement. **La classe doit absolument disposer d'un constructeur `public` sans argument.**

```
public void readExternal(ObjectInput in) throws IOException,
                                                 ClassNotFoundException
```

4.1.11.1 Risques

La sérialisation d'un objet permet d'accéder à ses données (même les champs déclarés `private`) à travers sa représentation comme une suite d'octets.

La sérialisation d'un objet peut rendre accessible des données « système » sur des ressources qui sont par ailleurs inaccessibles (par exemple une référence à un objet).

La désérialisation permet à un attaquant de créer des objets par un moyen qui n'utilise pas les constructeurs de classes et qui, en conséquence, contourne les contrôles effectués dans les constructeurs. De plus, la version par défaut de `readObject` ne vérifie pas que la suite d'octets correspond à la sérialisation d'un objet valable. Un attaquant peut ainsi forger une suite d'octets qui mène à la création d'objets partiellement initialisés.

La désérialisation peut ne contrôler la classe réellement désérialisée qu'*a posteriori*. En effet, le mécanisme de désérialisation tel qu'il est conçu est prévu pour renvoyer un objet quel-

conque dont le type est déterminé lors de la désérialisation elle-même. Après coup, l'objet retourné est *casté* dans le type que l'on pense récupérer, mais l'objet peut être de type différent et lors de sa désérialisation, le code de la méthode `readObject` a pu être exécuté et ainsi effectuer des opérations malicieuses.

4.1.11.2 Recommandations

- La possibilité de rendre une classe sérialisable doit être utilisée avec parcimonie et doit être accompagnée d'une analyse des parties d'un objet à sérialiser.
- Éviter que des champs avec des données confidentielles ou « système » apparaissent dans la sérialisation d'un objet, en leur donnant le modificateur `transient`.
- Éviter que des champs privés dans un objet désérialisé ne contiennent des références fournies par la sérialisation d'un objet. Pour ces champs, il convient de cloner les objets référencés.
- Répliquer les vérifications de contrôle d'accès (`checkPermission`) aux champs confidentiels dans la méthode `writeObject`.
- Traiter tout objet obtenu par désérialisation comme non-fiable. Répliquer les vérifications effectuées dans les constructeurs de la classe concernant les permissions d'accès et les invariants de classe dans la méthode `readObject`.

4.1.12 Programmation réflexive

La programmation réflexive est une discipline de programmation puissante et délicate qui permet à une application d'accéder et potentiellement de modifier son code lors de son exécution. La programmation réflexive sous Java permet de faire de l'introspection (analyser l'état de ces objets), de modifier l'état interne des objets (modifier la valeur des champs des objets) et d'invoquer des méthodes découvertes dynamiquement sur des objets. Si l'on dispose des permissions de sécurité ou s'il n'y a pas de `Security Manager`, il est également possible de modifier la valeur des champs utilisant les modificateurs `private, protected` et (*package*), ou d'appeler des méthodes avec ces modificateurs.

Lors de l'exécution, Java garde une représentation de chaque classe comme un objet de classe `java.lang.Class`. Via des méthodes disponibles dans les classes `java.lang.Object`, `java.lang.Class` et dans toutes celles du *package* `java.lang.reflect`, une application peut entre autre :

- obtenir une classe, soit en récupérant celle d'un objet chargé (par un appel comme `o.getClass()`), soit en chargeant une classe par la méthode `Class.forname` qui prend en paramètre le nom d'une classe à charger et rend comme résultat l'objet de type `java.lang.Class` qui porte ce nom ou une exception `ClassNotFoundException` si la classe n'a pas été trouvée par le *class loader* ;

- à partir d'un objet de type `java.lang.Class`, récupérer les informations de la classe :
  - le nom, le *package*, les modificateurs apposés sur la classe, les interfaces implémentées et la super-classe étendue (seulement les interfaces implémentées et la super-classe héritée directement par la classe et non l'ensemble de la hiérarchie de classe),
  - la liste des champs publics (de type `java.lang.reflect.Field`) par la méthode `getFields()` ou l'ensemble des champs (que que soit leur visibilité) en utilisant la méthode `getDeclaredFields()`. Pour utiliser cette dernière, il faudra les permissions de sécurité pour contourner les contrôles de visibilité,
  - de la même manière que pour les champs, la liste des méthodes (de type `java.lang.reflect.Method`) par les méthodes `getMethods()` et `getDeclaredMethods()`,
  - de la même manière que pour les champs, la liste des constructeurs (de type `java.lang.reflect.Constructor`) par les méthodes `getConstructors()` et `getDeclaredConstructors()`,
  - depuis Java 1.5, la liste des annotations (de type `java.lang.Annotation`) ;
- obtenir les modificateurs de visibilité des membres d'une classe et des classes elle-mêmes par la méthode `getModifiers()` ;
- suspendre les contrôles liés aux modificateurs de visibilité en utilisant la méthode `setAccessible` accessible depuis chacun des membres. Cette méthode ne peut être utilisée que si l'on dispose des permissions de sécurité ou s'il n'y a pas de *Security Manager* ;
- à partir des objets de type `Field` et `Method`, obtenir les noms des membres d'une classe, le type des champs, la liste des paramètres d'une méthode et leur type, etc. ;
- instancier de nouveaux objets soit en utilisant la méthode `Class.newInstance()` qui va appeler le constructeur `public` et sans argument de la classe, soit à partir des objets de type `java.lang.reflect.Constructor` en appelant leur méthode `newInstance(Object... initargs)` ;
- obtenir et modifier la valeur d'un champ avec les méthodes de type `java.lang.reflect.Field.get*` et `java.lang.reflect.Field.set*` ;
- exécuter les méthodes à partir des objets de type `java.lang.reflect.Method` en utilisant la méthode `invoke(Object obj, Object... args)`.

Il faut également noter que, sans *Security Manager* ou en ayant les permissions d'accès, il est possible d'utiliser directement les classes du *package* `sun.reflect` qui fournissent les moyens de contourner les politiques de sécurité. De plus, selon les implémentations et la documentation de la bibliothèque standard, les vérifications faites par le vérificateur de *bytecode* ne s'appliquent à aucune classe héritant de la classe `sun.reflect.MagicAccessorImpl`. En l'absence de *Security Manager* ou en ayant les permissions d'accès, il est possible de créer une classe héritant de la classe `sun.reflect.MagicAccessorImpl` en déclarant que cette classe fait partie du *package* `sun.reflect` (pour contourner la visibilité (*package*) de la classe `sun.reflect.MagicAccessorImpl`). Pour plus d'information concernant la mise en place d'une politique de sécurité, il faut se référer à la section 6.2.2.

Enfin, il faut également noter que, depuis Java 1.5, les classes du *package* `java.lang.instrument` permettent d'instrumenter des classes déjà chargées par la JVM en y ajoutant du bytecode. Ces classes ont été rajoutées pour permettre de faire par exemple du débogage ou du *profiling*. Selon la documentation de ces classes, il est seulement possible d'ajouter du bytecode et non de modifier ou de retirer celui qui est déjà en place. Pour plus de détails, voir la section 9.3.1.

4.1.12.1 Risques

La programmation réflexive offre moins de vérifications statiques que la programmation standard. Par exemple, il est possible de charger la classe fournie comme argument au programme avec le code suivant `cl = Class.forName(args[0])` et ensuite de créer une instance de cette classe avec l'appel `cl.newInstance()`. Garantir le bon typage d'un tel code nécessite de faire des vérifications dynamiques lors de l'exécution.

Un autre problème est que l'accès aux membres d'une classe par des méthodes réflexives peut contourner le contrôle d'accès effectué par inspection de pile parce que seule l'identité de l'appelant immédiat est vérifiée. Par exemple, un appel d'une méthode réflexive `java.lang.reflect.Field.set*` sur un objet `Field` va permettre de modifier la valeur d'un champ si l'appelant possède les droits d'effectuer cette modification, même si l'objet a été fourni par un code malveillant qui ne possède pas ces droits. Pour cette raison, il est deconseillé d'appeler des méthodes réflexives sur des objets de provenance inconnue (voir section 6-4 de [47]).

4.1.12.2 Recommandations

La programmation réflexive ne permet pas d'utiliser la vérification statique offerte par la programmation sans réflexion et doit donc être limitée au maximum. Le livre de Bloch [34] explique les inconvénients de la programmation réflexive d'un point de vue du génie logiciel et propose des techniques de programmation qui permettent de l'éviter.

Ne pas appeler des méthodes réflexives sur des instances d'objets créées à partir de classes dont la provenance n'est pas sûre.

Mettre en place un *Security Manager* dès le début de l'application pour interdire de passer outre les contrôles liés à la visibilité d'une classe ou d'un membre de classe, que ce soit pour des classes dont la provenance est sûre ou non.

4.1.13 Le modificateur `strictfp`

Les calculs flottants sont plus précis et effectués sur une plus grande plage de valeurs au niveau matériel que ce que requiert la spécification de Java. Il est donc possible d'obtenir des résultats différents selon les plates-formes matérielles. Le modificateur `strictfp` a été introduit pour être strictement conforme à la spécification Java et donc obtenir les mêmes résultats sur les calculs flottants quelle que soit la plate-forme matérielle d'exécution. Si une expression sur des flottants est considérée *FP-strict*, cela signifie que tous les résultats sur ces expressions doivent être ceux « prévus » dans la norme IEEE 754 sur l'arithmétique sur des flottants à simple ou double précision.

Le modificateur `strictfp` peut être appliqué sur des classes, des interfaces et des méthodes. Lorsque le modificateur est appliqué sur une classe, toutes les expressions concernant des caculs sur des flottants (`float` ou `double`) de chacune de ses méthodes ou de ses classes internes seront *FP-strict*. Il en est de même pour les expressions incluses dans les initialisations de variables, dans les constructeurs et dans les blocs statiques. Si le modificateur est appliqué sur une méthode, toutes les expressions concernant des calculs sur des flottants seront *FP-strict*. Enfin, si le modificateur est appliqué sur une interface, toutes les expressions concernant des calculs sur des flottants et toutes les classes déclarées dans l'interface [10] seront *FP-strict*. Par défaut, toutes les expressions constantes ou les annotations sont considérées implicitement comme étant *FP-strict*.

Par contre, toutes les expressions concernant des calculs sur des flottants qui ne sont pas constantes et qui ne sont pas au sein d'une classe, interface ou méthode utilisant le modificateur `strictfp` ne seront pas considérées *FP-strict* ; les résultats pourront être plus ou moins précis selon les architectures matérielles sur lesquelles ces expressions seront calculées.

La bibliothèque standard dispose de deux classes pour les opérations mathématiques : `java.lang.Math` et `java.lang.StrictMath`. La seconde classe fournit le support pour des calculs d'expressions *FP-strict*. En utilisant la classe `java.lang.StrictMath`, on s'assure les mêmes résultats quelle que soit la plate-forme matérielle d'exécution. Quant à la classe `java.lang.Math`, elle peut fournir des services plus rapides et plus précis. Cependant, il faut noter que `java.lang.Math`, pour plusieurs de ces méthodes, appelle les méthodes de la classe `java.lang.StrictMath`.

4.1.13.1 Risques

Nous n'avons pas connaissance de risque de sécurité particulier concernant ce modificateur, mais il nous paraissait important de souligner que, sans ce modificateur, les résultats pouvaient ne pas être les mêmes selon la plate-forme matérielle d'exécution.

10. Tout comme pour une classe, il est possible de déclarer une classe interne dans une interface, mais par contre, celle-ci sera forcément `public`.

4.1.13.2 Recommandations

- Utiliser les types flottants quand cela est réellement nécessaire.
- Utiliser le modificateur `strictfp` sur toutes les classes, interfaces et méthodes nécessitant une précision identique quelle que soit la plate-forme matérielle d'exécution.

4.2 Typage du langage Java

Cette section a pour vocation de donner un survol du mécanisme de vérification de types (ou *type checking*) en Java lors de la compilation. Le typage est un composant essentiel de l'architecture de sécurité en Java : sans typage, les mécanismes comme les modificateurs de visibilité et la protection de membres privés peuvent facilement être contournés. Le typage offre une encapsulation des données qui interdit des accès contournés comme par exemple l'arithmétique de pointeurs. Il faut cependant rappeler que les vérifications faites par le compilateur donnent peu de garanties sur la sécurité d'une application. En effet, le code exécuté par la JVM a pu être modifié après compilation. De plus, il faut noter que la machine virtuelle effectue une vérification de *bytecode*, dont entre autres une vérification de typage. Nous prêtons une attention particulière aux mécanismes du langage qui génèrent des vérifications dynamiques qui peuvent entraîner des exceptions lors de l'exécution.

Le langage Java opère avec deux sortes de types : les types primitifs et les types références. Il existe trois sortes de types références : les classes, les tableaux et les interfaces. Depuis la dernière version de Java, il est également possible de définir des types paramétrés et des types énumérés.

4.2.1 Types primitifs

Les types primitifs de Java se divisent entre les types numériques et les booléens. Les types numériques sont ensuite divisés entre les types entiers (`byte`, `short`, `int`, et `long`), les types flottants (`float` et `double`) et le type caractère (`char`).

4.2.2 Types références : classes et tableaux

Un objet est une instance d'une classe ou un tableau. Une référence est soit un pointeur vers un objet soit la valeur `null`.

La notion de sous-classe joue un rôle essentiel lors du typage. Si une classe `B` est une sous-classe de la classe `A`, alors les objets de classe `B` peuvent être affectés à des variables et à des

champs de type `A` et peuvent être utilisés comme argument à des méthodes ayant un paramètre déclaré de type `A`. En conséquence, la vérification de type (concernant les opérations d'affectation, d'accès aux champs, d'appels de méthodes, etc.) doit prendre en compte la hiérarchie de classes.

Le transtypage (ou *cast*) du langage Java permet de modifier le type d'un objet d'un type `B` vers un type `A` en vérifiant dynamiquement si `B` est une sous-classe de `A`. Statiquement, le typage de cette instruction ne peut être garantie, mais nécessitera des vérifications lors de l'exécution et retournera une exception `ClassCastException` si la vérification échoue.

Le typage de tableaux entraîne également des contrôles dynamiques lors de l'exécution. Les règles de typage de Java permettent d'affecter un tableau de classe `B[]` à une variable `T` de type `A[]` si `B` est une sous-classe de `A`. Si, plus loin dans le code, une instruction affecte un objet de classe `A` dans un des éléments du tableau `T` (`T[1] = new A();`) la vérification de typage sera incapable de déterminer s'il y a effectivement un tableau d'éléments de type A dans T. En conséquence, la vérification statique laisse passer cette instruction, mais génère une vérification dynamique qui lève une exception si l'élément affecté à `t[1]` n'est pas un sous-type de la classe des éléments du tableau stocké dans `T` (ce qui est le cas dans l'exemple).

4.2.3 Types références : interfaces

Les interfaces font partie de la hiérarchie de classes et peuvent donc être utilisées comme type pour des variables, des champs, etc. Contrairement aux classes, une classe `C` peut implanter plusieurs interfaces `I,J` qui seront à considérer comme des super-classes de la classe et qui permettent d'affecter un objet de classe `C` à une variable de type `I` ou `J`. La vérification de types devient alors plus complexe parce qu'un objet peut avoir plusieurs types (on parle aussi de *types d'intersection*). Au lieu de faire cette vérification statiquement, la vérification traite les interfaces comme des objets de type `java.lang.Object` et vérifie dynamiquement le bon typage de l'utilisation des opérations sur les interfaces. En effet, même si le compilateur vérifie bien qu'une classe définit l'ensemble des méthodes des interfaces qu'elle implémente, la JVM, et plus particulièrement le *bytecode verifier*, ne fait pas une telle vérification. La JVM effectue des vérifications durant l'exécution au moment de l'appel de ces méthodes et si celles-ci n'existent pas, une exception `AbstractMethodError` est levée (voir section 5.5.8). Par contre, si une classe ne définit pas une méthode d'une des interfaces qu'elle implémente, et que cette méthode n'est pas appelée, il n'y aura aucune levée d'exception.

4.2.4 Types paramétrés

Depuis la version 1.5, le langage Java permet l'utilisation de classes paramétrées (types génériques ou *generics* en anglais). Les classes paramétrées permettent d'écrire du code générique pouvant être instancié par différentes classes. L'exemple suivant permet de représenter des listes

contenant des objets de type quelconque. Dans la classe `MyList<A>`, `A` est le paramètre représentant la classe qui peut être utilisée pour déclarer des éléments (champs, variables,...). À partir d'une telle définition paramétrée, il est possible de créer des listes pour une classe quelconque comme on le fait pour la classe `Point` dans la méthode `main` en remplaçant le paramètre par le type voulu.

```
  class MyList<A>{A element; MyList<A> next;}

  class Point{int x; int y;}

  public class Test{
    public static void main(String[] args){
      MyList<Point> list = new MyList<Point>();
      list.element=new Point();
      list.element.x=0;
    }
}
```

Toutefois, la notion de classe paramétrée n'est pas indispensable. Il est tout à fait possible d'écrire un programme similaire en utilisant des *casts* explicites, comme le montre l'exemple ci-dessous. C'est d'ailleurs ce que fait un compilateur Java qui supprime la notion de classe paramétrée de cette manière. Cela signifie que la quasi-totalité des vérifications de typage, pour un type paramétré, aura lieu de manière dynamique durant l'exécution.

```
class MyList{Object element; MyList next;}

class Point{ int x; int y;}

public class Test{
  public static void main(String[] args){
    MyList list = new MyList();
    list.element = new Point();
    ((Point) list.element).x=0;
  }
}
```

La notion de classe paramétrée n'est en fait que du sucre syntaxique. En particulier, si la classe `B` est une sous classe de `A`, le système de type ne considère pas `MyList<B>` comme une sous classe de `MyList<A>` et le code suivant est rejeté :

```
  MyList<A> l = new MyList<B>();
```

À cet effet, le langage propose l'utilisation d'un type *joker*. L'exemple ci-dessous déclare `l` comme une variable pouvant accepter des listes d'éléments dont le type est une sous-classe de `A`, ce qui est le cas de `B`. Ce code est accepté :

```
  MyList<? extends A> l = new MyList<B>();
```

Il est également possible de ne pas imposer de restriction sur le type des éléments ou d'imposer que ce type soit une super-classe d'une autre comme dans les exemples ci-dessous.

```
MyList<?> l = new MyList<B>();
MyList<? super B> l = new MyList<A>();
```

Dans tous les cas, l'utilisation des types paramétrés ne fournit au niveau du langage que du sucre syntaxique. Au niveau du *bytecode*, les types paramétrés n'existent plus en tant que type paramétré. Ce mécanisme ne présente donc aucun risque supplémentaire et ne fournit qu'une simplification de l'écriture au niveau du langage en termes d'utilisation de *cast*.

4.2.5 Types enumérés

Depuis Java 1.5, il est possible de définir des types énumérés (*enumerated types*). La définition d'un type énuméré se fait en utilisant le mot-clé `enum` de la manière suivante :

```
public enum Jour {
  LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE
}
```

La définition d'une énumération se fait en listant l'ensemble des instances possibles de cette énumération. Dans l'exemple précédent, le type énuméré `Jour` liste l'ensemble des jours de la semaine. Il est également possible d'attacher une ou plusieurs valeurs à chacune des valeurs de l'énumération, comme le montre l'exemple suivant :

```
public enum Jour {
  LUNDI(1), MARDI(2), MERCREDI(3), JEUDI(4), VENDREDI(5),
  SAMEDI(6), DIMANCHE(7);

  // Numéro du jour dans la semaine
  private final int val;

  private Jour(int val){
    this.val = val;
  }
}
```

Afin de pouvoir ajouter ces valeurs à celles de l'énumération, il faudra par contre ajouter manuellement un constructeur auquel on ne pourra appliquer que les modificateurs de visibilité `private` ou (*package*).

Comme pour les types paramétrés (cf. section 4.2.4), l'utilisation d'un type énuméré en utilisant le mot-clé `enum` ne fournit au niveau langage que du sucre syntaxique. Il serait tout à fait possible de définir le type énuméré précédent sans utiliser le mot-clé `enum` de la manière suivante :

```
public final class Jour extends java.lang.Enum{
  public final static Jour LUNDI = new Jour(1);
  public final static Jour MARDI = new Jour(2);
  public final static Jour MERCREDI = new Jour(3);
  public final static Jour JEUDI = new Jour(4);
  public final static Jour VENDREDI = new Jour(5);
  public final static Jour SAMEDI = new Jour(6);
  public final static Jour DIMANCHE = new Jour(7);

  // Numéro du jour dans la semaine
  private final int val;

  private Jour(int val){
    this.val = val;
  }
}
```

Un type énuméré est transformé par le compilateur Java en une classe qui étend la classe `java.lang.Enum` et à laquelle on applique le modificateur `final`. Il est donc impossible d'hériter d'un type énuméré et, comme le montre le code précédent, il est impossible qu'un type énuméré hérite d'une autre classe, puisque Java n'autorise pas l'héritage multiple et que le type énuméré hérite forcément de la classe `java.lang.Enum`. Par contre, un type énuméré peut implémenter des interfaces. Chacune des valeurs de l'énumération est convertie en une instance du type énuméré utilisant les modificateurs `public final static`.

Enfin, il faut noter que deux méthodes statiques sont ajoutées automatiquement à chaque énumération :

- `values()` qui retourne un tableau contenant la liste des valeurs du type énuméré ;
- `valueOf(String val)` qui retourne la valeur du type énuméré correspondant à la chaîne de caractères passée en paramètre.

Remarque 2 *Un objet de type énuméré ne peut pas être cloné puisque la classe `java.lang.Enum` redéfinit la méthode `clone` en y ajoutant le modificateur `final` (pour empêcher que la méthode soit redéfinie) et en générant une exception de type `CloneNotSupportedException`. L'objectif est de conserver le statut de singleton d'une énumération.*

4.3 Sémantique

La sémantique formelle d'un langage de programmation a pour but de décrire de façon mathématique le comportement attendu d'un programme écrit dans ce langage. Un telle description formelle s'adresse à plusieurs publics. Les premiers sont les développeurs de programme qui

ont besoin de comprendre, sans ambiguïté, le comportement attendu des programmes qu'ils écrivent. Les seconds sont les développeurs des plates-formes d'exécution du langage (compilateur, machine virtuelle, vérificateur statique) qui utilisent cette sémantique comme une spécification haut niveau de leur plate-forme. Une description entièrement formelle de la sémantique d'un langage réaliste est cependant difficile à donner et les sémantiques existantes se contentent généralement d'une description informelle en langue naturelle.

La sémantique du langage Java est officiellement donnée par l'ouvrage *The Java Language Specification* de Gosling, Joy, Steele et Bracha [33]. Bien qu'un tel document ait avant tout le mérite d'exister et d'être régulièrement mis à jour, il s'agit d'une sémantique *informelle* (en anglais) qui est parfois imprécise.

La communauté scientifique s'est particulièrement intéressée à la formalisation du langage Java et plus particulièrement à son langage *bytecode*. Ces travaux ont permis, d'étudier en profondeur le système de type du langage, de renforcer certains passages trop faibles de la spécification officielle (par exemple concernant l'utilisation des *threads* [48]) et de renforcer la confiance globale dans le langage. Cependant, le travail de formalisation sur le papier est apparu comme difficile à passer à l'échelle pour un tel langage. L'utilisation d'assistants de preuve est donc de plus en plus courante pour spécifier de manière cohérente et vérifier mécaniquement les preuves réalisées.

Dans ce domaine, la référence actuelle est l'ouvrage de Stärk, Schmid et Börger [44] qui propose une formalisation (à base d'*abstract state machines*) du langage source et *bytecode* au moyen de cinq couches : un noyau impératif, une couche inter-procédurales avec une notion de classe, une couche objet, une couche ajoutant les exceptions et enfin une dernière couche pour la gestion des *threads*. L'ouvrage aborde la formalisation du vérificateur de *bytecode* et d'un compilateur non-optimisé entre chaque couche. Les auteurs semblent avoir décelé plusieurs erreurs dans la spécification officielle. Celles-ci concernent essentiellement la notion de *definite assignment* pour l'initialisation des variables locales et des champs finaux. Dans les premières versions de Java, le compilateur pouvait générer des programmes *bytecode* rejetés par le vérificateur de *bytecode*, car la vérification de *bytecode* était trop restrictive. Le problème est principalement lié à l'utilisation de sous-routines (voir section 5.2.10).

Une autre activité de formalisation importante a été menée par Drossopoulou et Eisenbach [43] qui ont modélisé une large partie du langage et démontré la correction de son système de type. Les définitions et les preuves de ces travaux ont été faites à la main. Syme [18] a ensuite formalisé une partie de ces travaux dans son propre démonstrateur de théorème *DECLARE*. Syme rapporte que le simple encodage des définitions dans un format mécanisé a soulevé de l'ordre de 40 erreurs. Plus important, des erreurs non triviales ont été ainsi détectées dans les preuves de Drossopoulou et Eisenbach.

Un large travail de formalisation a directement été développé par Nipkow et Klein [29] avec l'assistant à la preuve Isabelle/HOL. Les auteurs réalisent une sémantique opérationnelle grand pas et petit pas [11] pour un langage similaire à Java, baptisé Jinja. Les deux sémantiques sont

11. Dans ce contexte précis, une sémantique petit pas décrit formellement l'effet sur la mémoire d'une étape de

prouvées équivalentes. Un système de type ainsi qu'une analyse d'initialisation des variables locales (pour assurer que toutes les variables locales de chaque méthode sont écrites avant d'être lues) sont définis et prouvés corrects vis-à-vis de la sémantique petit pas. Les auteurs définissent aussi un compilateur vers un langage *bytecode* pour machine virtuelle. Ce langage est muni d'une sémantique opérationnelle petit pas, d'un système de type et d'un vérificateur de *bytecode*, tous prouvés corrects vis-à-vis de la sémantique. Enfin, une preuve atteste que le compilateur conserve la sémantique et le caractère typable des programmes. Un tel travail laisse de côté plusieurs aspects du langage (interfaces, tableaux, initialisation d'objets), mais il démontre la maturité suffisante des assistants de preuve et plus généralement, des travaux de recherche dans le domaine, pour permettre d'attaquer de tels objectifs sur **tout** Java.

Nous verrons dans la section 5.4, une description détaillée de la sémantique du langage *bytecode* Java. La sémantique du langage Java ne fera pas l'objet de davantage de développement formel ici, pour les raisons suivantes :

- La formalisation du langage *bytecode* a fait l'objet de plusieurs travaux depuis ces dernières années et est maintenant bien comprise ;
- La sémantique du langage Java n'est pas aussi critique que la sémantique du *bytecode*. Les mécanismes de vérification de la JVM constituent un dernier « rempart » dans l'architecture de sécurité Java. Ce sont donc avant tout eux qui doivent être corrects. Cette correction ne peut être établie que vis-à-vis d'une sémantique formelle du langage *bytecode*.

la machine virtuelle, alors qu'une sémantique grand pas modélise l'exécution globale d'une méthode.

5 CARACTÉRISTIQUES ET PROPRIÉTÉS DU *BYTECODE* JAVA

Malgré les apparences (par exemple, l'utilisation d'une pile ou le stockage sous forme binaire), le *bytecode* Java peut difficilement être considéré comme un langage bas niveau. Les *instructions* du *bytecode* sont très proches des *expressions* du langage source. La différence principale porte sur le fait que le *bytecode* est un langage non structuré. Au niveau *bytecode*, une méthode est une suite d'instructions pouvant contenir des instructions de saut (par exemple, pour la compilation d'une boucle `while`). Ceci représente toutefois une différence mineure, car les adresses de saut sont connues à la compilation (et non calculées dynamiquement comme en assembleur où il est possible de sauter à une adresse mémoire contenue dans un registre). De plus, les types sont conservés, ce qui autorise l'utilisation d'un *bytecode verifier*, qui permet de conserver les propriétés de sûreté du langage source.

Mise à part l'utilisation d'une représentation binaire plutôt que textuelle, la structure générale d'un programme Java, au niveau *bytecode*, est similaire à celle d'un programme Java au niveau source : un ensemble de classes [12], chaque classe comportant des déclarations de champs et méthodes. À un fichier source `.java` correspondent un ou plusieurs fichiers `.class`. Un fichier `.class` est une suite d'octets décrivant exactement une classe du fichier source (la compilation d'un fichier source génère autant de fichiers `.class` qu'il y a de classes définies dans ce fichier). La description précise du format binaire d'un fichier `.class` sort du cadre de cette présentation. Une description exhaustive de celui-ci est donnée par la spécification officielle de la machine virtuelle. On se bornera ici à donner les éléments nécessaires à la compréhension des instructions du *bytecode*. Afin de donner des exemples concrets, on s'appuiera sur l'outil en ligne de commande `javap` qui permet d'afficher de manière lisible le contenu d'un fichier `.class` en utilisant `javap -c filename`. L'exemple ci-dessous présente le source d'une méthode (à gauche) et le *bytecode* correspondant affiché par `javap` (à droite).

```
int i = 0;                          0: iconst_0
while (true){                       1: istore_1
  i++;                              2: iinc  1, 1
}                                   5: goto  2
```

Sans décrire les instructions de ce *bytecode*, qui le seront par la suite, on peut noter l'illustration de l'aspect non structuré du langage par l'utilisation du `goto`. Les numéros apparaissant à gauche des instructions dans le *bytecode* correspondent au décalage (en octets) de l'instruction par rapport au début de la méthode dans le fichier `.class` et non à un numéro de ligne.

Un fichier `.class` contient de nombreuses informations, en plus de la définition d'une classe, de ses champs et de ses méthodes. En particulier, un fichier `.class` possède une table appelée *constant pool* qui contient des constantes de type chaîne de caractères, des noms de

12. Plus précisément, un ensemble de classes et d'interfaces. Les interfaces jouent le même rôle que dans le langage Java source.

classe, champs et méthodes et d'autres constantes auxquelles le contenu du fichier fait référence. Dans l'exemple ci-dessous (affichage de `javap`) la position 2 décrit un élément de type *nom de classe* dont *la valeur* est en position 13. L'indirection représente une forme de typage du *constant pool*. Le vérificateur de *bytecode* vérifie la cohérence (l'indice référencé existe bien dans le *constant pool*) et le bon typage du *constant pool*.

```
(...)
const #2    =   class    #13 ;
(...)
const #13   =   Asciz    Test ;
```

Au niveau du *bytecode*, `javap` affiche, en tant que commentaire, le contenu du *constant pool* référencé (après l'indirection) par l'instruction. Par exemple, pour le *constant pool* ci-dessus, l'allocation d'un objet de type `Test` affichée par `javap` donne l'instruction et le commentaire suivants `0: new #2; //class Test`. Les exemples de cette section ne montreront que ces commentaires pour référencer de manière implicite le contenu du *constant pool*.

Beaucoup d'autres informations sont contenues dans un fichier `.class`. L'énumération de ces informations sort du cadre de cette présentation. Pour une présentation exhaustive, on se réferera à la spécification de la machine virtuelle. Au besoin, certains éléments seront évoqués lorsque ceux-ci seront nécessaires à la compréhension du *bytecode*.

Les instructions du *bytecode* Java manipulent divers éléments de la JVM (voir figure 6) :

1. la pile d'opérandes, locale à chaque méthode ;
2. les variables locales qui sont des registres eux aussi locaux à chaque méthode ;
3. la pile d'appels qui mémorise les différents contextes d'appels successifs (*frames*) constitués du point d'appel (nom de la méthode appelante et index de l'instruction d'appel dans le tableau d'opcodes de la méthode), de la pile d'opérandes (1) et des variables locales (2).
4. le tas, global, qui contient les objets et les tableaux tous alloués dynamiquement ;
5. les champs statiques, globaux eux aussi (en fonction de leur visibilité) ;

Au cours de l'exécution d'un programme par la machine virtuelle, chaque *thread* manipule sa propre pile d'appels et son propre contexte courant (contenu d'ailleurs par sa pile d'appels). Le tas et les champs statiques sont partagés. L'allocation en mémoire de ces différents éléments est décrite plus précisément dans [14].

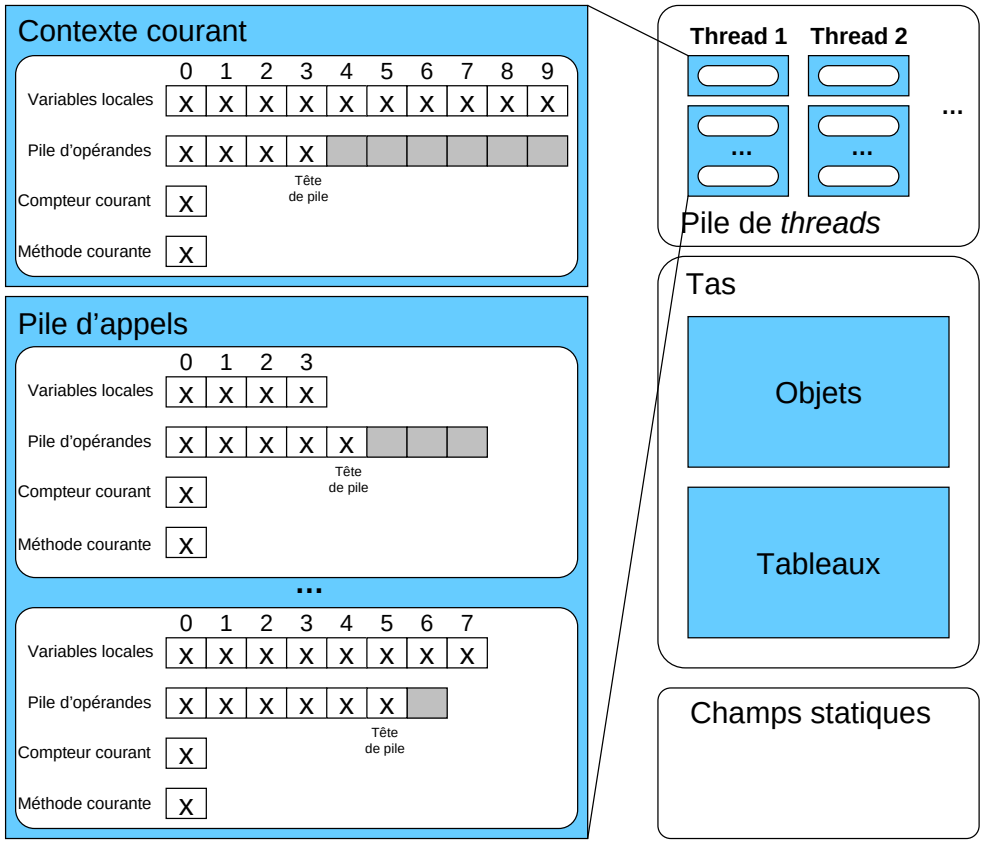


FIGURE 6 – Éléments mémoires manipulés par un programme *bytecode* Java.

5.1 Typage du langage Bytecode Java

Les valeurs manipulées dans les piles d'opérandes, les variables locales, les tableaux ou les champs appartiennent à deux catégories distinctes : les références et les valeurs numériques de type primitif.

5.1.1 Types primitifs

Les types primitifs sont moins nombreux que pour le langage source. Seuls figurent `int`, `float`, `long` et `double`. Les types `boolean`, `byte`, `short` et `char` du langage Java sont absents, ils sont représentés au niveau *bytecode* par des `int`, sauf pour les contenus de tableaux (un tableau étant une référence, voir les types références).

5.1.2 Types références

Il n'y a que deux catégories de types référence : les objets et les tableaux. Une référence à un objet admet pour type sa propre classe (qu'il a acquis lors de son allocation) ou une de ses superclasses (directes ou indirectes). Les tableaux sont compatibles avec le type `java.lang.Object`, mais ont aussi leurs propres types tableaux : `int[]`, `short[]`, `char[]`, `byte[]`, `boolean[]`, `float[]`, `long[]`, `double[]` et *C*`[]` avec *C* un type référence quelconque. Les types `short[]`, `char[]`, `byte[]` et `boolean[]` permettent d'assurer un typage précis des contenus de tableaux. Enfin le type interface, présent au niveau source, est absent au niveau *bytecode*. Les types interfaces sont en effet systématiquement identifiés avec le type `java.lang.Object`.

5.1.3 Autres types

La vérification de *bytecode* manipule trois types supplémentaires pour des raisons internes. Ces types ne sont pas directement apparents dans la syntaxe des programmes au niveau du *bytecode*. Contrairement aux autres types, ces types ne correspondent à aucun type du langage Java. Ces types sont :

- *ReturnAddress*. Ce type est utilisé dans la JVM par les instructions gérant les sous-routines (`jsr`, `jsr_w` et `ret`). Une valeur de type *ReturnAddress* est un pointeur sur un *opcode* de la JVM. Une valeur de ce type ne peut pas être modifiée par le programme ; la seule chose qui peut être faite est de l'enregistrer au moyen d'une instruction `astore` dans les variables locales ;
- *UninitializedThis*. Ce type est utilisé pour indiquer, dans un constructeur, le statut de l'objet courant. Il indique que l'objet n'a pas encore fini l'exécution de son constructeur et il aura ce type/statut jusqu'à la fin de l'exécution du constructeur parent (`super(...)`) où il reçoit alors le type référence correspondant à la classe instanciée par l'objet ;
- *UninitializedOffset*. Ce type est utilisé, dans une méthode, pour indiquer le statut d'un objet venant d'être alloué et sur lequel le constructeur n'a pas encore été appelé. Ce type interne disparaîtra dès que le constructeur correspondant à l'objet alloué aura terminé normalement (sans levée d'exception).

5.2 Présentation du jeu d'instructions

Une instruction de la machine virtuelle Java est constituée d'un *opcode* (codé sur un octet) suivi de zéro ou plusieurs paramètres. Dans un fichier `.class`, une instruction est donc une suite d'octets de longueur 1 ou plus (l'*opcode* suivi des paramètres). C'est ce qui explique que les décalages (positions des instructions) obtenus avec la commande `javap` ne soient pas

consécutifs. Le langage comporte environ 200 *opcodes* différents. En termes de fonctionnalité, le nombre d'opérations possibles est beaucoup moins élevé. Plusieurs *opcodes* sont dupliqués,

- soit pour distinguer une même opération sur différents types de données. Un *opcode* peut être précédé d'un préfixe précisant le type des valeurs sur lesquelles il opère : `s` pour un short, `b` pour un byte, `l` pour long, `d` pour double, `f` pour float et `a` pour une valeur de type référence. Dans le cas d'une valeur d'un tableau, l'*opcode* peut également être précédé par un préfixe précisant le type des valeurs référencées. Par exemple, pour un *opcode* manipulant un tableau d'entiers, on aura un préfixe `ia` ;
- soit pour représenter des cas particuliers, fréquemment utilisés. Ce type de codage permet à la fois de coder une instruction sur moins d'octets et d'améliorer les performances de l'opération de décodage des instructions par la machine virtuelle. Par exemple, l'instruction `iconst_0` empile l'entier 0 sur la pile et n'occupe qu'un octet. Cette instruction est équivalente à l'instruction `iconst 0` qui occupe deux octets (un pour l'opcode et un pour le paramètre).

Nous présentons rapidement le jeu d'instructions de la machine virtuelle Java en les regroupant par catégories de fonctionnalité. Celles-ci sont énumérées dans le tableau 7. La description qui suit le tableau reste assez imprécise dans le sens où on ne donne pas, pour chaque instruction, les paramètres de l'opcode ou la liste des valeurs dépilées ou empilées. Une description exacte de ces informations est donnée dans la spécification de la machine virtuelle. Dans cette présentation, on cherche simplement à donner un aperçu de l'expressivité de la machine virtuelle.

(A)	`bipush, sipush, ldc, ldc_w, ldc2_w, aconst_null, iconst_m1, iconst_N, lconst_N, fconst_N, dconst_N`
(B)	`pop, pop2, dup, dup2, dup_x1, dup2_x1, dup_x2, dup2_x2, swap`
(C)	`getfield, putfield`
(D)	`getstatic, putstatic`
(E)	`iload, iload_N, lload, lload_N, fload, fload_N, dload, dload_N, aload, aload_N`
(F)	`istore, istore_N, lstore, lstore_N, fstore, fstore_N, dstore, dstore_N, astore, astore_N`
(G)	`baload, saload, iaload, laload, faload, daload, caload, aaload`
(H)	`bastore, sastore, iastore, lastore, fastore, dastore, castore, aastore`
(I)	`i2b, i2s, i2l, i2f, i2d, l2i, l2f, l2d, f2i, f2l, f2d, d2i, d2l, d2f`
(J)	`lcmp, fcmpl, fcmpg, dcmpl, dcmpg`
(K)	`iinc`
(L)	`new`
(M)	`nop`

(N)	`multinewarray, newarray`
(O)	`arraylength`
(P)	`instanceof, checkcast`
(Q)	`ineg, lneg, fneg, dneg, iadd, ladd, fadd, dadd, isub, lsub, fsub, dsub, imul, lmul, fmul, dmul, idiv, ldiv, fdiv, ddiv, irem, lrem, frem, drem`
(R)	`ixor, lxor, ior, lor, iand, land, iushr, lushr, ishr, lshr, ishl, lshl`
(S)	`ifeq, iflt, ifle, ifne, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq, if_acmpne`
(T)	`tableswitch, lookupswitch`
(U)	`goto, goto_w`
(V)	`invokevirtual, invokestatic, invokespecial, invokeinterface`
(W)	`ireturn, lreturn, freturn, dreturn, areturn, return`
(X)	`athrow`
(Y)	`ret, jsr, jsr_w`
(Z)	`monitorenter, monitorexit`

TABLE 7: Les différentes familles d'instructions du *bytecode* Java

5.2.1 Manipulation exclusive de la pile d'opérandes

On distingue deux catégories d'instructions permettant de manipuler la pile d'opérandes. La première catégorie (A) correspond aux instructions qui empilent une constante sur la pile. Les *opcodes* `bipush` et `sipush` empilent une valeur, respectivement de types `byte` et `short` [13], sur la pile. Les *opcodes* `ldc`, `ldc_w` et `ldc2_w` prennent en paramètre une position dans le *constant pool* et empilent la valeur obtenue à cette position du *constant pool* (Cette valeur a pu être calculée, si l'entrée référencée dans le *constant pool* présente une ou plusieurs indirections dans le *constant pool*). Les *opcodes* `ldc_w` et `ldc2_w` sont des versions de `ldc` correspondant respectivement à un mode d'adressage et à un type de contenu différents. La constante `null` est empilée par l'*opcode* `aconst_null`.

13. nous avons précisé précédemment que les types `short` et `byte` n'étaient pas présents au niveau du *bytecode*. Ici, ces types sont seulement présents à titre d'optimisation de la taille du fichier `.class`. Quand ces instructions seront exécutées en mémoire, ce sera bien un `int` qui sera placé sur la pile.

Finalement, les derniers *opcodes* de cette classe sont des versions dupliquées des précédents. Par exemple, l'*opcode* `iconst_m1` code sur un seul octect l'instruction qui empile la valeur `-1`. La notation `iconst_N` représente en fait les *opcodes* `iconst_0`, ..., `iconst_5` qui empilent respectivement les entiers $0, \ldots, 5$. Les notations `lconst_N`, `fconst_N` et `dconst_N` ont la même signification pour les types associés à leurs préfixes. Un exemple d'utilisation du *constant pool* est le chargement sur la pile d'une référence à une chaine de caractères connue à la compilation. Cette opération est réalisée par l'instruction `ldc #2` si 2 est la position dans le *constant pool* de la chaîne de caractères.

La seconde catégorie (B) correspond aux instructions qui manipulent (suppriment, dupliquent ou permutent) les éléments de la pile.

5.2.2 Manipulation des variables locales

Au niveau *bytecode*, une variable est représentée par un entier (ou position). À l'entrée d'une méthode d'instances prenant deux paramètres `int x, float y` au niveau source, celle-ci utilisera au niveau *bytecode* les variables locales 0, 1 et 2. Les paramètres `x` et `y` correspondent aux positions 1 et 2, la position 0 étant réservée à l'adresse de l'objet sur laquelle la méthode courante a été appelée (mot-clé `this` au niveau source). Pour une méthode de classe, les paramètres auraient les positions 0 et 1, car il n'y a pas besoin de l'adresse d'un objet. Cependant, il faut noter que ces positions sont celles définies à l'entrée de la méthode, rien n'empêche le *bytecode* de modifier les positions de chacune de ces variables locales. Par exemple, ce n'est pas parce qu'une instruction écrirait à la position 0, que celle-ci changerait l'adresse de l'objet sur lequel la méthode a été appelée.

Les instructions des catégories (E) et (F) permettent de manipuler ces variables locales. Une instruction de la catégorie (E) charge le contenu d'une variable sur la pile (par exemple `iload 1` empile le contenu de la position 1 et ce contenu est un entier). Une instruction de la catégorie (F) dépile la valeur en tête de pile et la charge dans une variable locale (par exemple `istore 1` stocke la valeur en tête de pile dans la variable locale d'indice 1 et cette valeur est de type entier). Les autres instructions de ces deux catégories ont la même signification pour les autres types. Les instructions suffixées par `N` représentent plusieurs instructions où `N` représente cette fois l'indice de la variable locale.

5.2.3 Instructions de contrôle

Les catégories (M), (S), (T) et (U) regroupent les instructions de contrôle. La catégorie (M) contient uniquement l'instruction `nop` dont l'exécution consiste à passer à l'instruction suivante.

Une instruction `goto offset` (de la catégorie (U)) réalise un saut inconditionnel vers une autre instruction de la même méthode. Le paramètre `offset` représente un décalage (un entier

positif ou négatif) entre l'instruction courante et celle vers laquelle le programme doit sauter. L'instruction `goto_w offset` a la même sémantique que `goto offset` à ceci près que `offset` représente un décalage sur 32 bits au lieu de 16 bits dans le premier cas. Une instruction `goto`, comme toutes les instructions de saut (hormis les invocations de méthodes), ne peut sauter qu'à une instruction se trouvant dans la même méthode. Cela implique que le décalage ne peut pas être de plus de 65536 octets qui est la taille maximale du code d'une méthode.

Les instructions de la catégorie (S) réalisent un saut conditionnel en fonction des valeurs entières en tête de pile. Par exemple, l'instruction `ifeq offset` teste si la valeur en tête de pile est 0. Si c'est le cas, l'exécution est poursuivie au point de programme obtenu à partir du décalage `offset` par rapport à la position de l'instruction `ifeq`, sinon elle est poursuivie au point de programme suivant l'instruction `ifeq offset`.

Pour réaliser un saut en fonction de valeurs numériques de type `float`, `double` ou `long`, il est nécessaire d'utiliser les opérations de comparaison de la catégorie (J), qui placent 0 sur la pile si le résultat de la comparaison est `false` et 1 si c'est `true`, et de combiner cette valeur avec `ifeq offset` pour tester l'égalité à zéro de la tête de pile. On renvoie à la spécification de la machine virtuelle pour la description exhaustive de ces différentes instructions.

Les instructions de la catégorie (T) sont des instructions de taille variable. L'instruction `lookupswith` lit une valeur entière sur la pile et réalise un saut en fonction de cette valeur. Intuitivement, cette instruction est le pendant de `switch` au niveau source. Son paramètre (de taille variable) est une table de paires (valeur, décalage vers le point de programme). Le saut est réalisé au point de programme correspondant (dans cette table) à la valeur lue sur la pile. Un point de programme par défaut est choisi lorsqu'aucune valeur ne correspond.

L'instruction `tableswitch` peut être également utilisée pour représenter le `switch` du langage source. Intuitivement, cette dernière est une version optimisée de `lookupswitch` qui permet d'éviter de tester toutes les entrées de la table lorque la valeur testée n'y est pas. Pour cela, cette instruction dispose, parmi ces paramètres, de bornes inférieures et supérieures sur les valeurs de sa table.

Grâce à la vérification de *bytecode*, un programme vérifié possède la propriété que tous les sauts ont pour cible une instruction valide de la méthode courante. La cible potentielle de ce saut est directe (n'est pas calculée à l'exécution).

5.2.4 Appels et retours de méthodes

Les instructions des catégories (V) et (W) gèrent respectivement les appels et retours de méthodes. Les quatres instructions de la catégorie (V) correspondent à différents types d'invocation de méthodes. Ces quatres instructions prennent comme paramètre une position du *constant pool* contenant une description textuelle de la méthode utilisée pour la résolution dynamique. L'information placée dans le *constant pool* permet en particulier de déterminer le

nombre de paramètres de la méthode et donc le nombre de valeurs à dépiler et à placer dans les variables locales de la nouvelle *frame*. L'instruction `invokevirtual` est la plus couramment utilisée. Elle correspond à l'appel d'une méthode sur l'objet placé en tête de pile. Comme mentionné précédemment, dans ce cas, la variable locale 0 reçoit l'adresse de l'objet en question. L'instruction `invokestatic` correspond à l'appel d'une méthode statique. Dans ce cas, seuls les paramètres de la méthode sont sur la pile (il n'y a pas d'objet sur lequel effectuer l'appel). L'instruction `invokespecial` est un cas particulier de `invokevirtual` utilisée pour l'appel de méthodes de super-classes, de méthodes privées et de méthodes d'initialisation. L'instruction `invokeinterface` est utilisée pour l'appel de méthodes d'interface. Dans le cas de méthodes natives, la gestion des paramètres lors de l'appel dépend de l'implémentation, la spécification n'imposant aucune restriction.

Les instructions de la catégorie (W) terminent une méthode et retournent si nécessaire une valeur (la tête de pile) à la méthode appelante. Les différentes instructions sont préfixées par le type de la valeur retournée. L'absence de préfixe correspond à une méthode dont le type de retour est `void` et qui ne retourne donc pas de valeur.

5.2.5 Opérations arithmétiques et booléennes

Les instructions des catégories (K), (Q) et (R) manipulent toutes des valeurs numériques de types `int`. On y trouve les opérations classiques de manipulations de valeurs numériques (la description complète de ces instructions peut être trouvée dans la spécification de la machine virtuelle).

5.2.6 Manipulations d'objets

Les instructions des catégories (C) et (D) permettent de manipuler les champs d'instance (C) ou de classe (D). Dans le cas des instructions de la catégorie (C), l'objet dont le champ est lu ou écrit est celui positionné en tête de pile. L'instruction `new` de la catégorie (L) permet d'allouer un nouvel objet dans le tas. L'adresse de cet objet est placée sur la pile.

5.2.7 Manipulations de tableaux

Les instructions des catégories (G) et (H) permettent d'accéder aux éléments d'un tableau (respectivement en lecture et en écriture). Comme pour les instructions des catégories (E) et (F), le préfixe indique le type de ces éléments. En revanche, contrairement aux instructions portant sur la gestion des variables, celles-ci utilisent deux paramètres placées en tête de pile, l'adresse du tableau et l'index de l'élément. Les instructions de la catégorie (N) permettent la création

de tableaux. L'instruction `newarray` crée un nouveau tableau de dimension 1. L'instruction `multinewarray` permet de créer un nouveau tableau à plusieurs dimensions. Dans les deux cas, les dimensions sont lues sur la pile. L'instruction `arraylength` de la catégorie (O) permet de connaître la taille du tableau dont l'adresse est sur la pile. Il est à noter que tout accès à un indice d'un tableau est accompagné de manière implicite d'une vérification des bornes. Ce dernier point est détaillé dans la section 5.5.3.

5.2.8 Manipulations de type

La spécification officielle de la JVM n'impose aucun choix *a priori* concernant la localisation de l'information du type d'un objet. Cependant, puisque qu'une référence mémoire doit avoir la même taille qu'une valeur de type `int`, il est peu probable qu'une implémentation de JVM place l'information de classe au niveau de la référence elle-même. L'information est donc plus probablement placée sur les objets eux-mêmes. Il s'agit de toute façon d'une information immuable. En effet, durant l'exécution, la classe d'un objet (et de toutes les références qui pointent vers lui) ne change jamais.

Les instructions de la catégorie (P) permettent de tester le type des objets. L'instruction `instanceof` va comparer l'objet qui se trouve en tête de pile avec la classe dont l'indice dans le *constant pool* est passé en paramètre de l'instruction. Si la classe de l'objet en tête de pile est la même (ou une sous-classe) que celle passée en paramètre de l'instruction, alors l'entier 1 est placé sur la pile, sinon c'est l'entier 0.

L'instruction `checkcast` vérifie si la classe de l'objet en tête de pile est la même (ou une sous-classe) de la classe dont l'indice dans le *constant pool* est passé en paramètre de l'instruction `checkcast`. Si ce n'est pas le cas, une exception `java.lang.ClassCastException` est levée. Le type de l'objet n'est pas modifié, mais le type passé en paramètre de l'instruction `checkcast` est utilisé lors de la vérification de *bytecode*.

5.2.9 Exceptions

Au niveau *bytecode*, les blocs `try`/`catch` d'une méthode sont remplacés par une *table d'exceptions*. Chaque méthode possède sa propre table d'exceptions (éventuellement vide). Une table d'exceptions est une liste de descriptions de *handlers* (blocs `try`/`catch`), chaque description donnant la portée, la cible et le type de l'exception rattrapée par le *handler*.

```
public static void main(String[] args){
  int[] array = new int[10];
  try{int x = array[12];}
  catch (ArrayIndexOutOfBoundsException exc){
    System.out.println(exc);
  }
```

```
}

public static void main(java.lang.String[]);
Code:
 0:   bipush  10
 2:   newarray int
 4:   astore_1
 5:   aload_1
 6:   bipush  12
 8:   iaload
 9:   istore_2
 10:  goto    21
 13:  astore_2
 14:  getstatic       #16; //Field java/lang/System.out:
                           // Ljava/io/PrintStream;
 17:  aload_2
 18:  invokevirtual   #22; //Method java/io/PrintStream.println:
                           //(Ljava/lang/Object;)V
 21:  return
Exception table:
 from   to  target type
   5    10    13   Class java/lang/ArrayIndexOutOfBoundsException
```

Dans l'exemple précédent, le code source essayant de copier le contenu de la douzième case du tableau dans la variable `x` correspond aux lignes [5...10] du *bytecode*. La description du bloc `try/catch` englobant donne donc une portée allant de la ligne 5 à la ligne 10. La cible (ici la ligne 13) correspond au point de programme auquel l'exécution doit reprendre si une exception de type `ArrayIndexOutOfBoundsException` est levée par le code compris entre les lignes 5 à 10. L'intruction `athrow` de la catégorie (X) lance une exception (correspondant à l'objet en tête de pile).

5.2.10 Sous-routines

Les instructions de la catégorie (Y) sont des instructions de contrôle particulières permettant la programmation de sous-routines. Les instructions `jsr` et `jsr_w` prennent respectivement en paramètre un *offset* sur 2 et 4 octets, sautent à l'adresse ainsi indiquée contenant une sous-routine et stockent l'adresse de retour sur la pile d'opérandes. Cette adresse ne peut pas être modifiée, elle peut et doit seulement être stockée (par une instruction `astore`) dans une variable locale afin d'être réutilisée par l'instruction `ret` dont le paramètre est l'indice de la variable locale.

Historiquement, une motivation pour l'introduction de ces instructions était le codage du `finally` qui n'existe pas au niveau *bytecode*. Freund [45] montre que le dépliage de sous-routine augmente seulement de 0,02% la taille du code (pour tout le JDK 1.1.2). Les compi-

lateurs courants ne les utilisent plus. Pour le code uniquement disponible au format `.class`, il est possible d'appliquer un algorithme de dépliage permettant de supprimer l'usage de ces instructions et de déplier (*inlining*) le code correspondant. À partir de la version de *bytecode* 51.0 (ce qui devrait correspondre à Java 1.7), les instructions de cette catégorie ne devront plus apparaître pour être conforme à la spécification [33].

5.2.11 Synchronisation

À proprement parler, les seules instructions du *bytecode* liées à la gestion des *threads* sont les instructions de la catégorie (Z). Les instructions `monitorenter` et `monitorexit` permettent respectivement de prendre et de relâcher un verrou. Bien que la notion de *thread* fasse partie intégrante du langage Java, les opérations de création de *threads*, de synchronisation par variables de condition ne sont que des invocations de méthodes natives au sein de la classe `java.lang.Thread`. Ces opérations n'ont aucune instruction particulière au niveau *bytecode*. Au niveau source, la prise de verrou est structurée (bloc ou méthode `synchronized`). Cette structuration est perdue au niveau *bytecode*. La spécification du langage Java demande à ce que les verrous pris par une méthode soient relâchés par cette même méthode avant qu'elle ne termine (dans le cas contraire, une exception est levée). Les compilateurs essaient toutefois d'éviter ce cas de figure en ajoutant un handler portant sur la portion de code protégée par le verrou qui en cas de levée d'une exception relâche le verrou et relance l'exception.

5.3 Vérification de *bytecode*

Les instructions du langage *bytecode* sont typées (hormis celles de la catégorie (B) qui sont polymorphes). Chaque instruction ne doit donc être exécutée qu'avec des opérandes de types adéquats. Cette restriction est assurée par le système de type et le vérificateur de *bytecode* qui rejettent les programmes qui n'adhèrent pas à cette politique de typage.

Les garanties intrinsèques assurées par ce typage sont les suivantes :

- garantie sur la taille de la pile d'opérandes : chaque instruction doit avoir suffisamment d'opérandes sur la pile. Pour éviter le débordement de pile, le typage assure en fait que, pour tout point de programme, toutes les piles d'opérandes atteignant ce point aient la même taille ;
- pas d'arithmétique de pointeurs : les opérations arithmétiques n'ont lieu que sur des opérandes de type numérique. Aucune opération ne permet de transtyper (*cast*) un pointeur en valeur numérique ;
- pas de *forge* de pointeurs : les opérations d'accès à la mémoire dynamique (lecture/écriture de champs, de tableaux) n'ont lieu que sur des opérandes de type référence. Aucune opération ne permet de transtyper (*cast*) une valeur numérique en référence.

Les éléments non traités par ce typage sont les suivantes :

- pas de type booléen : les booléens sont traités comme des entiers. À la différence du langage Java source, il est donc possible de faire de l'arithmétique avec les booléens ;
- pas de types paramétrés : pour des raisons de compatibilité arrière avec les bibliothèques existantes (distribuées sous forme de *bytecode*), le langage *bytecode* n'a pas été muni de types paramétrés. Contrairement au niveau source, le système de type du *bytecode* n'assure pas statiquement que les manipulations de structures paramétrées sont sûres. Il s'appuie sur des *casts* dynamiques pour éviter toute faille. L'exemple suivant illustre ce point.

  ```
  A foo(Vector<A> v) {
    return v.firstElement();
  }
  ```

  Cette méthode récupère le premier élément d'un vecteur d'élément de classe `A`. Aucun *cast* n'est nécessaire, car le compilateur assure statiquement que `v.firstElement();` aura toujours un type compatible avec `A`. Néanmoins, un *cast* est inséré par le compilateur au niveau *bytecode*.

  ```
  A foo(java.util.Vector);
    Code:
     0: aload_1
     1: invokevirtual #2;
       //Method java/util/Vector.firstElement :
       //()Ljava/lang/Object;
     4: checkcast #3; //class A
     7: areturn
  }
  ```

  Sans cela, le vérificateur de *bytecode* n'aurait aucun moyen de comprendre que le résultat de l'appel virtuel est toujours compatible avec la classe `A` et rejetterait le programme ;
- pas de types énumérés : pour des raisons de compatibilité arrière avec les bibliothèques existantes (distribuées sous forme de *bytecode*), le langage *bytecode* n'a pas été muni de types énumérés. Un type énuméré est transformé en une classe `final` qui hérite de la classe `java.lang.Enum`. Pour plus de détails sur cette transformation, voir la section 4.2.5 ;
- pas de variable `this` : par convention, lors de l'appel d'une méthode non statique, l'objet sur lequel est appelé la méthode (la variable `this` au niveau source) est placé dans la variable locale 0. Contrairement au niveau source où, on ne peut écrire `this = o`, rien n'empêche, au niveau *bytecode*, d'écrire dans la variable 0. Par contre, ce n'est pas parce qu'on écrit dans la variable 0 que l'on réécrit `this`. De plus, la portée de la réécriture de la variable 0 n'ést que local à l'exécution d'une méthode. Dès qu'un autre appel de méthode a lieu, la variable `this` correspondant à cet appel de méthode sera placée dans la variable 0 pour l'exécution de cette méthode. Enfin, rien n'empêche non plus dans le *bytecode* d'une méthode d'enregistrer la variable source `this` dans une autre variable locale.

– pas de classes internes : cette notion est absente du langage *bytecode*. Les classes internes sont transformées lors de la compilation en des classes standards. Cela pose certains problèmes de sécurité vis-à-vis des champs privés de la classe englobante manipulés dans les classes internes. Lorsqu'un de ces champs privés est utilisé dans une classe interne, il ne peut plus rester totalement privé pour être encore accessible depuis la classe interne une fois compilée. Le compilateur a alors recours à une transformation qui élargit la visibilité du champ privé. Cette transformation s'appuie, soit sur une transformation du modificateur de visibilité de `private` à une visibilité au sein du même *package*, soit sur l'introduction d'une nouvelle méthode visible au sein du même *package*, pour lire le champ qui reste privé. C'est cette dernière technique qui est utilisée dans l'exemple suivant où une classe interne `In` d'une classe `A` accède à son champ statique privé `f` dans son constructeur.

```
class A {
  private static int f;
   class In { public int g = f; }
}
```

Le compilateur génère deux classes `A.class` et `A$In.class`.

```
class A extends java.lang.Object{
  private static int f;

  [...]

  static int access$000();
    Code:
     0: getstatic #1; //Field f:I
     3: ireturn
}

class A$In extends java.lang.Object{
  public int g;

  final A this$0;

  A$In(A);
  Code:
    0:  aload_0
    1:  aload_1
    2:  putfield  #1; //Field this$0:LA;
    5:  aload_0
    6:  invokespecial #2; //Method java/lang/Object."<init>":()V
    9:  aload_0
    10: invokestatic  #3; //Method A.access$000:()I
    13: putfield  #4; //Field g:I
    16: return
}
```

Pour accéder au champ statique `f`, le constructeur de la classe `A$In` utilise la méthode statique `access$000` insérée par le compilateur dans la classe `A`. Cette méthode étant visible dans le *package*, elle pourrait être appelée depuis la classe d'un attaquant ayant réussi à s'intégrer au *package* de `A`.

En plus de ces propriétés de typage, le vérificateur de *bytecode* assure les propriétés suivantes :

- le compteur de programme pointe, à tout moment de l'exécution, sur une instruction valide de la méthode en cours ;
- initialisation des variables locales : la lecture d'une variable locale est toujours précédée d'une écriture. Cela évite de pouvoir accéder à un contenu non-initialisé de la mémoire. Le même type de propriété est assuré pour les autres emplacements mémoires de la JVM par un mécanisme dynamique. Les champs (statiques ou d'instance) et les cellules des tableaux sont initialisés avec une valeur par défaut compatible avec leurs types (`null` pour une référence et `0` ou `0.0` pour un numérique) lors de la création de l'objet ou de la classe correspondante ;
- initialisation d'objet : quand une instance d'une classe `A` est créée, elle ne peut pas être utilisée tant qu'un constructeur de la même classe `A` n'aura pas été appelé sur l'objet. Plus précisément, chaque constructeur de classe, à l'exception de celui de la classe `java.lang.Object`, doit appeler un autre constructeur de la même classe ou de sa super-classe directe avant d'accéder à un des champs de l'objet en cours de construction. Cette propriété n'empêche cependant pas qu'un objet « échappe » avant la fin de sa construction, *i.e.* qu'il soit accessible par d'autres *threads* (à travers un champ statique par exemple), ou qu'il apparaisse comme paramètre `this` d'une méthode virtuelle ;
- les classes (respectivement les méthodes) déclarées `final` ne sont pas héritées (respectivement redéfinies dans une sous-classe). Aucune vérification n'est par contre effectuée vis-à-vis de **la déclaration `final` des champs**. Seul le compilateur assure la bonne utilisation de cette notion, au niveau source. Toutefois, à l'exécution de l'instruction `putfield`, une vérification dynamique est tout de même effectuée pour vérifier que le champ `final` ne peut être écrit qu'au sein d'une méthode de la classe dans laquelle le champ est déclaré ;
- la manipulation des sous-routines (par les instructions (Y)) nécessite plusieurs vérifications, car ces instructions manipulent des adresses de saut sur la pile d'opérandes et dans les variables locales. Ce trait du langage étant inutile (voir Section 5.2.10), il est préférable de simplement interdire l'usage de telles instructions (ce qui sera le cas pour des classes compatibles Java 1.7). On a alors l'assurance qu'aucune adresse de saut ne sera manipulée et que tous les sauts intra-procéduraux, hormis les exceptions potentielles vers des *handlers*, sont alors apparents dans la syntaxe du programme.

Depuis Java 1.6, le vérificateur de *bytecode* utilise une technique de vérification en deux temps, inspirée des concepts du code porteur de preuve (PCC : *Proof Carrying Code*, présenté dans la Section 8.4). L'algorithme standard est basé sur une analyse de flot de données qui doit itérer pour trouver la solution d'un ensemble d'équations récursives. Pour éviter ce calcul itératif lors du chargement d'un programme, il est possible d'attacher une solution au programme (lors de la compilation) et de seulement vérifier qu'il s'agit bien d'une solution en réalisant une seule

passe. Aucune confiance n'a besoin d'être accordée a priori à cette solution puisqu'elle est vérifiée. En pratique, cette solution est seulement partiellement transmise, car une partie peut être reconstruite sans itération. L'information transmise prend la forme de *stackmaps* qui sont des annotations de type pour les variables locales et la pile d'opérandes, attachées à certains points de programme. Cette technique sépare donc l'inférence de type de la vérification de type. La technique avait initialement été introduite pour Java CLDC afin de réduire le temps de calcul et l'occupation mémoire de la vérification de *bytecode* embarqué sur les petits systèmes portables. Son utilisation dans Java SE semble essentiellement provenir d'un besoin toujours plus grand d'efficacité dans le démarrage d'une application, mais aussi de la volonté d'utiliser des algorithmes simples dans la base de confiance de l'architecture Java. Le vérificateur de *bytecode* formant une part primordiale dans cette base, il est ainsi allégé de tout son moteur d'inférence. Pour être plus précis, pour le *bytecode* compatible Java 1.5 (version de *bytecode* 49.0), seul le mécanisme d'inférence est utilisé ; pour le *bytecode* compatible Java 1.6 (version de *bytecode* 50.0), le mécanisme de vérification par stackmaps est utilisé et s'il échoue, alors le mécanisme d'inférence est utilisé ; enfin pour les versions de *bytecode* supérieures à 50.0, seule la vérification par stackmaps devra être utilisée.

5.4 Sémantique

La sémantique du langage *bytecode* Java a fait l'objet de plusieurs travaux de formalisation. Là encore, si les premiers travaux étaient uniquement manuscrits[6, 46, 51], les plus récents utilisent un assistant de preuve[29, 16]. Bertelsen [6] fut un des premiers à tenter de formaliser la sémantique du langage. Ces travaux soulevèrent plusieurs problèmes dans la première édition de la spécification officielle. L'une des spécifications les plus complètes est celle de Cohen [8] qui donna une spécification détaillée d'un fragment du langage incluant notamment le chargement de classe. Cette spécification a la particularité d'être entièrement écrite dans l'assistant de preuve ACL2 et d'être exécutable. La plupart des formalisations sémantiques suivantes eurent pour but de formaliser le vérificateur de *bytecode* et prouver sa correction vis-à-vis d'une sémantique formelle du langage.

La sémantique du langage *bytecode* est suffisamment précise pour être décrite de façon mathématique. Les meilleures références sur le sujet sont les travaux de Stärk, Schmid and Börger [44] et ceux de Freund et Mitchell [24]. Dans ces derniers travaux, la sémantique prend la forme d'une relation entre les états mémoires d'une JVM, modélisant une étape élémentaire pour l'exécution d'une instruction. Nous commençons d'abord par introduire la notation pour définir des règles sémantiques. Un état mémoire est de la forme

$$(< M, pc, s, l >:: cs, h)$$

avec $< M, pc, s, l >:: cs$ la pile d'appels dont le sommet $< M, pc, s, l >$ représente le contexte courant et cs la pile d'appels. Le contexte courant est formé d'une méthode M, d'un point de programme pc, d'une pile d'opérandes s et d'un ensemble de variables locales l ; et avec h le tas. Ce contexte se trouve en tête de la pile d'appels $< M, pc, s, l >:: cs$.

Nous donnons ici un exemple de règle sémantique pour une instruction très simple comme `iload`.

$$\frac{instrAt(M,pc) = \texttt{iload}\ x \quad ValidVar(x) \quad IsInt(l(x))}{(< M,pc,s,l >:: cs,h) \rightarrow (< M,pc+1,l(x) :: s,l >:: cs,h)}$$

Pour simplifier, nous ne présentons qu'une version séquentielle, il n'y a donc qu'une pile d'appels. Une telle règle se lit de la manière suivante : il existe une transition de la machine virtuelle à partir d'un tel état mémoire si toutes les conditions suivantes (celles spécifiées au dessus de la barre horizontale) sont assurées :

1. l'instruction courante est de la forme `iload` x. Dans le cas contraire, c'est une autre règle sémantique qui devra s'appliquer ;
2. le nom de variable x est *valide*, c'est-à-dire que son numéro est inférieur au nombre maximum de variables locales déclarées dans la méthode M ;
3. la valeur $l(x)$ associée à la variable x représente un entier.

Ces deux dernières conditions sont toujours assurées lorsqu'on rencontre une instruction `iload` x. C'est une garantie du vérificateur de *bytecode*. Il n'est donc pas nécessaire de spécifier un comportement en dehors de ces cas. Sous ces conditions, la transition s'effectue vers un état $(< M,pc+1,l(x) :: s,l >:: cs,h)$ où le point de programme $pc+1$ représente le successeur de pc et la pile d'opérandes est maintenant constituée de la valeur de la variable x. Cet état est placé au sommet de la pile d'appels cs.

Les assistants de preuve comme Coq ou Isabelle permettent de spécifier de façon plus mécanique l'exemple précédent. Pour cet exemple précis, la relation de transition prendrait la forme suivante dans le langage de spécification de Coq.

```
Inductive step : state -> state -> Prop :=
... (* autres cas *) ...
| step_iload : forall M pc s l cs h x,
    instrAt (M,pc) = iload x ->
    ValidVar x ->
    IsInt (l x) ->
    step ((M,pc,s,l)::cs,h) ((M,next pc,(l x)::s,l)::cs,h).
```

La relation de transition est ici définie comme une relation inductive entre deux états mémoires de type `state` (`Prop` désigne le type associé aux propriétés logiques). Une telle définition se compose d'un ensemble de règles décrivant tous les cas où la relation a lieu. Le cas que nous faisons apparaître ici est facilement mis en relation avec la description mathématique précédemment donnée. Nous illustrons ici le fait que, au moins pour le cas de la sémantique d'une JVM, la mécanisation dans un assistant de preuve reste relativement facile d'accès, tout en apportant une cohérence très forte sur l'ensemble des définitions posées. Cette cohérence ne met pas à l'abri des mauvaises interprétations de la spécification officielle, mais elle permet de lever toutes les ambiguïtés inévitables des spécifications informelles.

5.5 Vérifications dynamiques

Un certain nombre de propriétés intrinsèques du langage *bytecode* et du langage source ne sont pas assurées statiquement, ni par le compilateur, ni par le vérificateur de *bytecode*. Elles sont néanmoins assurées dynamiquement grâce à une exécution dite défensive qui vérifie la légalité des actions avant de les effectuer. Cette approche a l'avantage d'être généralement plus simple à mettre en oeuvre, car les mécanismes associés sont plus simples que leurs homologues statiques. Cependant, elle retarde la détection des erreurs jusqu'au moment de l'exécution et ralentit l'exécution des programmes. Au niveau du langage, chaque violation d'une de ces propriétés occasionne la levée d'une exception dont la classe hérite de `RuntimeException`. Dans cette partie, nous présentons ces différentes exceptions et les propriétés intrinsèques qui leurs sont associées. Toutes les exceptions listées dans cette partie correspondent aux réponses de la JVM à des vérifications faites implicitement (dans le sens où elles n'apparaissent pas dans le *bytecode*).

5.5.1 `NullPointerException`

Toutes les instructions du langage qui manipulent une référence ont généralement besoin de la déréférencer. Une référence est soit égale à une adresse mémoire dans le tas, soit à la constante `null`. Dans le premier cas, les propriétés intrinsèques du langage assurent que l'adresse correspond toujours à un objet (ou un tableau) valide. Dans le deuxième cas, une exception de classe `NullPointerException` est levée.

5.5.2 `ArrayStoreException`

Le système de type des tableaux Java ne permet pas d'assurer statiquement que les objets stockés dans un tableau auront un type compatible avec son type déclaré.

```
class A {}
class B extends A {}
class Array {
  public static void main(String[] args) {
    A[] t = new B[10]; // affectation acceptée
                       // car B[] est un sous-type de A[]
    t[0] = new B();    // affectation correcte
    t[1] = new A();    // affectation incorrecte :
    // Cette dernière ligne génère une erreur à l'exécution
  }
}
```

Dans l'exemple précédent, le tableau `t` est déclaré statiquement de type `A[]`, mais il est instancié dynamiquement avec le type `B[]`. Cela est accepté grâce à la règle de sous-typage Java qui dit que `B[]` est un sous-type de `A[]` si `B` est un sous-type de `A`. L'affectation `t[0] = new B();` est autorisée, puisque le type instancié du tableau est `B[]`. Par contre, l'affectation suivante `t[1] = new A();` est refusée, car il est interdit de mettre un élément de type `A` dans un tableau de type `B[]` (`A` n'est pas une sous-classe de `B`), même si statiquement le tableau a été déclaré de type `A[]`. Cette erreur n'est détectée ni à la compilation, ni par le vérificateur de *bytecode*. Elle est détectée dynamiquement par la JVM, qui lance une exception `java.lang.ArrayStoreException: A`. **Le type instancié prévaut sur le type statiquement déclaré.**

Cette règle de sous-typage a un intérêt pratique pour le programmeur. Sans elle, l'opération JCodeA[] t = new B[10] précédente nécessiterait une copie explicite de la forme :

```
A[] a = new A[10];
B[] t = new B[10];
for (int i=0;i<b.length;i++){
  a[i]=t[i];
}
```

Par contre, après une telle copie, il est possible d'ajouter, dans le tableau `a`, des éléments de type `A`, par l'opération `a[3] = new A();`.

Cette règle de sous-typage, dite *covariante* [37], est néanmoins responsable des limitations du vérificateur de *bytecode*. Il est à noter qu'elle n'a pas été reprise pour les types génériques Java. C'est ainsi que dans l'exemple suivant, l'affectation est proscrite par le système de type Java.

```
import java.util.ArrayList;
class A {}
class B extends A {}
class List {
  public static void main(String[] args) {
    ArrayList<A> t = new ArrayList<B>(); // erreur a la compilation
  }
}
```

5.5.3 ArrayIndexOutOfBoundsException

Les tableaux sont alloués dynamiquement dans le tas avec une taille déterminée à l'exécution. En mémoire, un tableau est constitué de trois informations :

1. *n*, la taille déclarée du tableau (qui n'évolue plus après création),
2. *t*, le type déclaré des éléments (qui ne change plus après création),
3. les *n* éléments dont les types sont compatibles avec *t* (c'est-à-dire *t* ou une sous-classe de *t*).

Il est important de faire la distinction entre les 3 notions suivantes :

- le type déclaré d'une variable tableau. Sur l'exemple suivant, c'est la classe `A` ;
- le type déclaré des éléments du tableau (ce qui correspond à *t* précédemment introduit). C'est le type utilisé au moment de la création du tableau. Sur l'exemple suivant, c'est la classe `B` ;
- le type réel des éléments du tableau. C'est le type des objets placés dans les « cases » du tableau et qui doit correspondre impérativement à *t* ou à un de ses sous-types. Dans l'exemple suivant, cela correspond aux classes `B` et `C`.

Dans l'exemple suivant, `C` hérite de `B` qui lui-même hérite de `A`.

```
A[] tab = new B[10];
tab[0] = new B();
tab[1] = new C();
```

L'information sur la taille est utilisée à chaque accès, en écriture ou en lecture, pour vérifier si l'accès a bien lieu dans les bornes $[0, n-1]$. Dans le cas contraire, une exception de classe `ArrayIndexOutOfBoundsException` est lancée. Cette vérification n'est pas faite explicitement dans le *bytecode*, mais elle est implicite à l'exécution d'une des instructions des catégories (G) et (H).

5.5.4 NegativeArraySizeException

L'instruction `newarray t` nécessite un entier *n* en tête de pile. La JVM alloue alors un tableau de taille *n* de type `t[]` en initialisant les *n* éléments avec la valeur par défaut pour le type `t`. Si l'entier *n* est strictement négatif, une exception de classe `NegativeArraySizeException` est lancée. Par contre, si l'entier est égal à `0`, l'allocation du tableau crée un objet tableau de taille `0`, mais tout accès à un des indices du tableau conduira à une exception de type `ArrayIndexOutOfBoundsException`.

5.5.5 ArithmeticException

L'exception `ArithmeticException` est utilisée lorsqu'une division par zéro ou une opération de modulo par zéro est tentée. Les instructions concernées sont `idiv`, `ldiv`, `irem` et `lrem`. Les dépassements arithmétiques (*overflow*) ne lèvent pas d'exception. Aucune opération flottante ne lève d'exception. Une division flottante par zero retournera soit un `NaN` (Not a Number) dans le cas où le dividende est égal à `0f`, soit un `Float.POSITIVE_INFINITY` ou un `Float.NEGATIVE_INFINITY` dans les cas où le dividende est respectivement un flottant positif ou négatif.

5.5.6 `ClassCastException`

L'exception `ClassCastException` est lancée par l'instruction `checkcast C`, si la référence en tête de pile est nulle ou si le type de la référence n'est pas `C` ou une sous-classe de `C`.

5.5.7 `IncompatibleClassChangeError`

Lors de l'appel virtuel d'une méthode déclarée dans une interface, le système de type standard ne permet pas de prédire si la classe de l'objet sur lequel l'appel est effectué, implémentera effectivement l'interface concernée. La JVM utilise par conséquent une vérification dynamique et lance une exception de classe `IncompatibleClassChangeError` en cas d'échec.

5.5.8 Exceptions liées à la résolution dynamique

La nature profondément dynamique du langage implique qu'il n'est pas possible de prédire avant l'exécution si une classe mentionnée dans un programme bénéficie des éléments attendus ou tout simplement si elle existe. Ce n'est qu'au moment où la première instruction manipulant cette classe est exécutée que la vérification est effectuée. Cette phase est appelée *résolution*. Les cas d'erreurs possibles sont les suivants :

- si aucune classe ne correspond au nom annoncé, la JVM lance une exception de type `NoClassDefFoundError` ;
- si la résolution d'un nom de classe `C` (suite à une instruction `new C`) aboutit à une interface ou une classe abstraite, la JVM lance une exception `InstantiationError` ;
- lors de la lecture ou l'écriture d'un champ par une des instructions des catégories (C) et (D), si le champ n'existe pas, la JVM lance une exception `NoSuchFieldError` ;
- lors de l'accès (ou de l'écriture) à un champ statique, s'il y a une exception lors de l'initialisation de la classe (initialisation de ses champs statiques), alors la JVM lance une exception `ExceptionInInitializerError` ;
- lors de l'accès à un champ non statique par une instruction `getstatic` ou `putstatic`, ou inversement lors de l'accès à un champ statique par une instruction `getfield` ou `putfield`, la JVM lance une exception `IncompatibleClassChangeError` ;
- un appel de méthode dont la résolution aboutit à une méthode absente ou abstraite provoque une exception `NoSuchMethodError` ou `AbstractMethodError` ;
- lors de l'accès à un champ, à une méthode ou à une classe, si l'élément est inaccessible (de par ses modificateurs de visibilité par exemple), la JVM lance une exception `IllegalAccessError` ;
- un appel de méthode native dont la résolution aboutit à une méthode native introuvable provoque une exception `UnsatisfiedLinkError`.

5.5.9 `IllegalMonitorStateException`

La JVM vérifie dynamiquement qu'une méthode respecte un bon parenthésage des instructions `monitorenter`/ `monitorexit` pour la prise et le relâchement de verrous.

5.5.10 Exceptions asynchrones

Toutes les exceptions précédentes sont lancées pour un type bien précis d'actions (et donc un nombre restreint d'instructions). Les exceptions dites asynchrones peuvent, par opposition, être lancées à tout moment de l'exécution d'un programme. Il s'agit des exceptions suivantes :

- `InternalError` : erreur interne liée au logiciel implémentant la JVM, au système d'exploitation sous-jacent ou à une faute matérielle.
- `OutOfMemoryError` : la mémoire physique ou virtuelle n'a pas la capacité nécessaire pour allouer un nouvel emplacement.
- `StackOverflowError` : un processus nécessite d'étendre sa pile d'appels et la capacité mémoire actuelle ne le permet pas.
- `UnknownError` : une exception ou une erreur a eu lieu, mais la JVM est incapable de donner sa nature exacte.

On voit que la spécification officielle est assez lâche sur ces points afin de faciliter le travail d'implémentation d'une JVM.

6 MÉCANISMES DE SÉCURITÉ OFFERTS PAR LES CLASSES DE BASE DU JRE

Les classes de la bibliothèque standard du JRE constituent, avec le langage Java et la JVM, une partie importante de l'environnement d'exécution standard de Java. Elles implémentent notamment un certain nombre de mécanismes de sécurité mis en œuvre nativement par la plate-forme d'exécution ou qui peuvent être utilisés par un développeur pour implémenter des fonctions de sécurité dans les applications Java. Les mécanismes qui sont abordés dans ce rapport sont les suivants :

- les mécanismes de programmation concurrente (gestion des *threads*), présentés en section 6.1 ;
- les mécanismes de contrôle d'accès, présentés en section 6.2 ;
- le mécanisme de chargement de classes, présentés en section 6.3 ;
- les mécanismes cryptographiques, présentés en section 6.4.

6.1 Programmation concurrente

6.1.1 *Thread* Java

6.1.1.1 Notion de thread

Depuis la toute première version du langage Java, celui-ci inclut la notion de *thread*. Un *thread* représente un fil d'exécution d'instructions *bytecode* Java. Chaque *thread* s'exécute indépendamment des autres *threads* se trouvant dans la JVM, et on ne peut présupposer d'aucun ordonnancement entre les instructions s'exécutant dans les différents *threads*.

La zone de mémoire (le tas) où sont alloués les objets et les tableaux (champs statiques ou non) est partagée par tous les *threads* d'une instance de la JVM. Certaines données ne sont pas stockées dans cette zone et par conséquent ne sont pas partagées entre les threads [14], comme par exemple les variables locales et les paramètres des méthodes.

14. Le fait que les données soient ou non dans la zone de mémoire partagée n'a rien à voir avec le fait que les *threads* soient créés à partir de la même classe, de la même instance d'objet, ou de classes différentes.

6.1.1.2 *Spécificités des* threads *Java*

Depuis la version J2SE 5.0 et la JSR 133 [48], le langage Java propose une description de son fonctionnement dans un environnement *multi-threads* multi-processeurs, et ce indépendamment de l'implémentation des *threads* par l'OS sous-jacent. Toutefois, les JVM modernes s'appuient sur les services de *threading* fournis par l'OS pour implémenter les *threads* Java. Ceci a pour conséquence que l'entrelacement des *threads* Java va jusqu'au code assembleur de la JVM (mode d'exécution interprété) ou au code assembleur généré par la JVM (mode d'exécution *JIT*).

Le modèle mémoire Java et les problématiques de synchronisation afférentes sont décrits en section 6.1.3.

6.1.1.3 *Introduction à la problématique de synchronisation*

Un programme non correctement synchronisé peut avoir un comportement ne correspondant pas au résultat escompté par le développeur. En effet, pour des raisons d'efficacité, un programme peut être l'objet d'optimisations effectuées par :

- le compilateur Java ;
- un éventuel compilateur *JIT* ;
- le processeur natif lors de l'exécution des instructions cibles.

Ces optimisations peuvent modifier le comportement des programmes mal synchronisés. En effet, en l'absence de synchronisation explicite, ces optimisations supposent une exécution séquentielle du code modifié. Parmi ces optimisations, on trouve en particulier, le réordonnancement d'instructions (cf. section 6.1.1.4) et l'utilisation de mémoire cache (cf. section 6.1.1.5).

6.1.1.4 *Réordonnancement des instructions*

Le réordanncement des instructions est une optimisation essentielle pour les processeurs modernes. Cette technique permet en particulier de limiter les interruptions du *pipeline* pour cause de dépendance entre instructions. Cependant, dans un environnement *multi-threads*, cette optmisation peut poser des problèmes si l'on n'y porte pas attention.

Considérons l'exemple suivant dans lequel deux *threads* exécutent chacun le code suivant, où `L1` et `L2` sont des variables locales à chacun des *threads*, et `A` et `B` des variables globales avec intialement `A == B == 0` :

Thread 1	*Thread 2*
`L2 = A`	`L1 = B`
`B = 1`	`A = 2`

Un développeur s'attend à ce que les instructions de chacun des *threads* s'exécutent dans l'ordre dans lequel elles ont été définies. Le tableau suivant présente ainsi les différentes possibilités d'ordre d'exécution des instructions, en fonction de l'ordonnancement des *threads* :

Entrelacement 1	Entrelacement 2	Entrelacement 3	Entrelacement 4	Entrelacement 5

`L2 = A`
`B = 1`
`L1 = B`
`A = 2`
	`L2 = A`
	`L1 = B`
	`B = 1`
	`A = 2`
		`L1 = B`
		`L2 = A`
		`B = 1`
		`A = 2`
			`L1 = B`
			`L2 = A`
			`A = 2`
			`B = 1`
				`L1 = B`
				`A = 2`
				`L2 = A`
				`B = 1`

`L1 = 1` `et L2 = 0`	`L1 = 0` `et L2 = 0`	`L1 = 0` `et L2 = 0`	`L1 = 0` `et L2 = 0`	`L1 = 0` `et L2 = 2`

La dernière ligne du tableau précédent présente les valeurs de `L1` et `L2`. Il est donc impossible d'avoir à la fois `L1` égale à `1` et `L2` égale à `2`.

Considérons maintenant le réordonnancement suivant du code de *Thread* 1 (valide selon le modèle mémoire de Java) :

```
B = 1
L2 = A
```

On ne peut rien trouver à redire sur cette optimisation d'un point de vue séquentiel, car localement il n'y a aucune dépendance entre la variable globale `B` et la variable locale `L2`. Cependant, l'exécution de ce nouveau code en parallèle de *Thread* 2 rend possible l'obtention du résultat (`L1 = 1` et `L2 = 2`) en utilisant l'ordonnancement suivant des instructions :

```
B = 1
L1 = B
A = 2
L2 = A
```

Cette optimisation est permise, car il n'y a pas de synchronisation entre les deux *threads* sur les accès aux variables globales (*data race*).

Conclusion : l'utilisation de primitives de synchronisation permet de désactiver localement ce type d'optimisations, ce qui a pour effet de rétablir la cohérence de l'exécution. Par exemple, deux instructions séparées par une synchronisation ne seront pas réordonnées. Les accès concurrents à la mémoire partagée doivent toujours être synchronisés.

6.1.1.5 Problématique de la mémoire cache dans un environnement multi-cœur

Dans un environnement multi-cœur, chaque cœur dispose de sa propre mémoire cache (il en est de même pour les processeurs dans un un environnement multi-processeurs). Comme le réordonnancement d'instructions, la gestion de la mémoire cache repose sur des raisonnements locaux en l'absence de synchronisation explicite. Cette gestion locale de la mémoire cache (si celle-ci est de type *write-back* [15]) peut entrainer des problèmes de cohérence tels que l'inversion des écritures comme présentée dans l'exemple suivant :

(Initialement `A = B = 0`)

Cœur 1	*Cœur 2*
`L1 = A + B`	`A = 21` `B = 2 * A`

(Quelle est la valeur de L1 ?)

La figure 7 représente une exécution du programme précédent dans laquelle le cœur 1 voit l'écriture de la variable globale `B` avant celle de `A`. Par conséquent, la réponse à la question précédente est `42`, alors qu'on pouvait s'attendre à obtenir soit `0`, soit `21`, soit `63` en fonction de l'ordre des exécutions. Il faut noter que l'optimisation présentée dans la section précédente ne peut être appliquée sur cet exemple, puisqu'il y a une dépendance entre la valeur de `A` et celle de `B`.

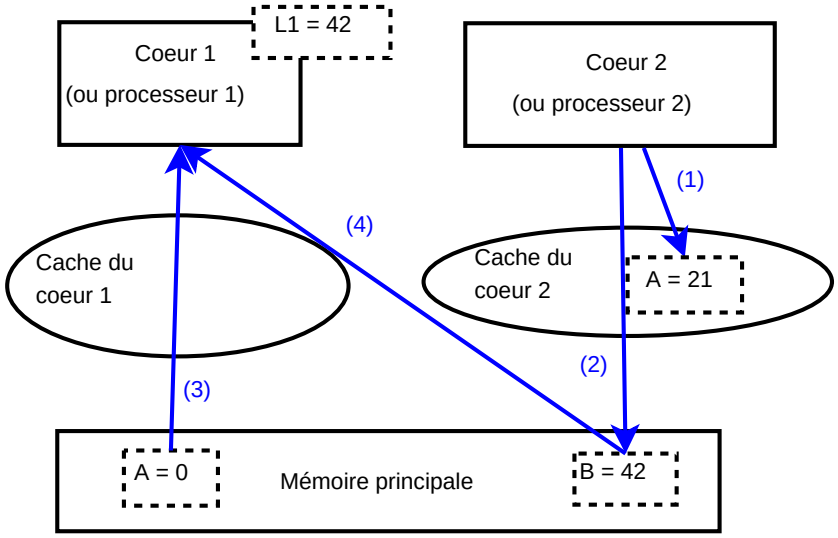


FIGURE 7 – Incohérence liée à un cache de type *write-back*

Le problème vient ici du fait que, en l'absence de synchronisation, chaque cœur maintient une « vue locale » de la mémoire principale, et que les données écrites par un *thread* s'exécutant sur un cœur donné peuvent ne pas être recopiées immédiatement en mémoire principale.

15. Un cache de type *write-back* supporte les modifications de valeurs dans le cache avec mise à jour différée de la mémoire globale.

Conclusion : l'utilisation de primitives de synchronisation a pour effet d'invalider la mémoire cache et donc de forcer l'écriture en mémoire principale.

6.1.1.6 Manipulation des threads

Les *threads* Java se présentent sous la forme d'instances de la classe `java.lang.Thread`. Un *thread* est démarré en appelant la méthode `start()` de l'objet de type *Thread*. Le code du *thread* à proprement parler peut être défini de deux manières différentes :

- soit en héritant de la classe `java.lang.Thread` et en définissant une méthode `run()` ;
- soit en passant, au constructeur de `Java.lang.Thread`, une instance d'une classe implémentant l'interface `java.lang.Runnable`, celle-ci ayant défini une méthode `run()`.

Dans tous les cas, c'est le code de la méthode `run()` qui est appelé au démarrage du *thread* (soit celle du *thread* lui-même, soit celle de l'objet passé à son constructeur). Il est intéressant de noter qu'il n'existe aucun mécanisme sûr pour arrêter ou suspendre un *thread* depuis un autre *thread* (les méthodes qui permettaient de réaliser ces opérations sont déclarées `deprecated`). En revanche, il est possible d'attendre que le *thread* se termine en utilisant la méthode `join()`.

6.1.2 Synchronisation et notification

6.1.2.1 Système de verrouillage initial

La synchronisation du langage Java est basée sur la manipulation d'un verrou intrinsèque à chaque objet, appelé « moniteur ». Deux opérations sont possibles sur un moniteur : l'acquisition et la libération.

La manipulation des moniteurs remplit deux objectifs dans le langage Java :

1. synchroniser les accès aux données devant être partagées par plusieurs *threads* ;
2. assurer l'atomicité de sections critiques.

La relation *happens-before* (définie par le modèle mémoire Java, voir 6.1.3) entre l'opération de libération d'un moniteur et toute opération d'acquisition du même moniteur garantit que le code exécuté après l'acquisition du moniteur verra les opérations effectuées par le code exécuté avant son acquisition.

Le langage Java tel que spécifié en version 1.0 fournissait deux moyens de réaliser ces opérations d'aquisition et de libération d'un moniteur par l'intermédiaire du mot-clé `synchronized` :

1. les méthodes `synchronized` ;

2. les blocs de code `synchronized`.

Une fois qu'un *thread* Java est entré dans une méthode `synchronized` d'un objet `o`, aucune autre méthode `synchronized` de cet objet `o` ne peut s'exécuter dans le contexte d'un *thread* différent. Tous les autres *threads* qui essaient d'entrer dans une méthode `synchronized` du même objet se retrouvent suspendus jusqu'à ce que le *thread* `t` sorte de la méthode `synchronized`. Conceptuellement, le moniteur est acquis à l'entrée de la méthode et relâché en sortie. Dans le cas d'une méthode `synchronized static`, le verrou est placé sur l'objet de type `Class` représentant la classe de l'objet.

```
public class SynchronizedCounter {
  private int c = 0;

  public synchronized void increment() {
      c++;
  }

  public synchronized void decrement() {
      c--;
  }

  public synchronized int value() {
      return c;
  }
}
```

Cependant, ce niveau de verrouillage est assez grossier : il est en effet impossible par exemple de protéger une donnée membre en particulier. Dans le but de permettre un verrouillage plus fin, on entoure un bloc de code par le mot-clé `synchronized`, comme présenté dans l'exemple ci-dessous :

```
public void addName(String name) {
  synchronized(this) {
      lastName = name;
      nameCount++;
  }
  nameList.add(name);
}
```

Dans ce cas, le moniteur est acquis en entrée du bloc `synchronized` et libéré en sortie.

Une autre technique permettant de synchroniser l'accès à une variable partagée est l'utilisation du modificateur `volatile`. Dans ce cas,

- la valeur de cette variable ne sera jamais placée dans la mémoire cache (écriture directe en mémoire) ;
- les accès à cette variable se comporteront comme s'ils se trouvaient à l'intérieur d'un bloc `synchronized`.

6.1.2.2 Notifications

Java inclut un mécanisme de notifications *inter-threads*. Ce mécanisme (bien que le nom ne soit apparu que récemment dans J2SE 5.0) est couramment appelé « variables de conditions ». Il permet à un *thread* de se mettre en attente d'un événement qui sera envoyé en temps utile par un autre *thread*.

Ce mécanisme est implémenté par les méthodes `wait()`, `notify()` et `notifyAll()` de la classe `java.lang.Object`. La méthode `wait()` suspend l'exécution du *thread* courant jusqu'à ce qu'un autre *thread* appelle la méthode `notify()` ou `notifyAll()` de l'objet sur lequel la méthode `wait()` a été appelée. Plusieurs *threads* peuvent être en attente sur un même objet. `notifyAll()` permet de tous les réveiller, alors que `notify()` ne réveille que l'un d'entre eux. Ni le *thread* choisi dans le cas du `notify()`, ni l'ordre de réveil des *threads* dans le cas du `notifyAll()` ne sont déterministes.

Il est à noter que le langage Java, comme tout autre mécanisme de *thread* basé sur POSIX, autorise un comportement contre-intuitif au niveau de la méthode `wait()`. En effet, celle-ci peut parfois rendre la main sans qu'aucun *thread* n'ait appelé `notify()` pour cet objet [16] (*spurious wakeup*). Pour gérer ceci, le développeur **doit** utiliser `wait()` de la façon suivante :

```
synchronized(obj) {
  while (<condition fausse>)
      obj.wait(timeout);
  ... // Réalise l'action correspondante
}
```

6.1.2.3 Nouveautés J2SE 5.0

L'environnement J2SE 5.0 ajoute de nouveaux éléments en ce qui concerne la programmation concurrente. Ces éléments sont des éléments de synchronisation de haut niveau qui se trouvent dans le nouveau *package* `java.util.concurrent` et ses *sous-packages*. On y trouve en particulier :

- l'interface `Condition` fournit un point d'entrée pour implémenter des mécanismes fonctionnellement proches (mais étendus) de `wait()` et `notify()`. Cette interface est implémentée par exemple par la classe `AbstractQueuedSynchronizer.ConditionObject` qui est définie dans le *package* `java.util.concurrent.locks` ;
- l'interface `Lock` fournit un point d'entrée pour implémenter des verrous offrant des opérations plus étendue que celles offertes par le mot-clé `synchronized`. Ces opérations permettent entre autres une structuration et le support de l'aggrégation de

16. Il s'agit en fait d'un détail technique d'implémentation des JVM qui a été rémonté jusque dans la spécification, probablement dû au fait que la fonction utilisée sous UNIX pour implémenter le `wait()` peut rendre la main de façon prématurée.

variables de conditions. Toute classe implémentant cette interface doit, entre autres, définir les méthodes `tryLock()` (tentative non bloquante de prendre un verrou), et `lockInterruptibly()` (tentative de prendre un verrou qui peut être interrompue). Par exemple, la classe `ReentrantLock` implémente cette interface ;

- l'interface `ReadWriteLock` fournit un point d'entrée pour implémenter des verrrous différenciant le verrouillage pour la lecture de celui pour l'écriture. Par exemple, la classe `ReentrantReadWriteLock` implémente cette interface ;
- le *package* `java.util.concurrent.atomic` fournit des classes permettant de réaliser des opérations *test and set* et *get and set* (sans poser de verrou) sur des variables partagées ;
- le *package* `java.util.concurrent` étend les collections Java par un ensemble de structures facilitant la manipulation, en parallèle, de larges quantités de données et réduisant le besoin de synchronization explicite de la part du développeur. On peut citer entre autres les `ConcurrentHashMap` et `BlockingQueue`.

Ces nouveautés suivent une tendance des langages de développement à usage généraliste à faciliter la vie du développeur de programmes concurrents, et de proposer un verrouillage plus fin afin d'améliorer les performances. *Il est donc vivement recommandé de s'en servir.*

Il est à noter que ces nouveautés ne sont pas uniquement des ajouts dans la biliothèque de classes. En effet, elles ont nécessité des modifications dans la JVM (support des `ReadWriteLock` par exemple).

6.1.3 Modèle mémoire Java

6.1.3.1 Introduction

Le modèle mémoire Java a été entièrement revu avec la JSR 133 [48]. Ces travaux ont abouti à la réécriture complète du chapitre 17 de la spécification du langage Java [33] : *Threads and Locks*.

Le modèle mémoire Java s'attache à décrire, de manière formelle, les comportements qu'un programme Java utilisant différents fils d'exécution est autorisé à avoir, et non comment la gestion des *threads* doit être implémentée, le but étant de laisser une marge de manœuvre pour l'implémentation des *threads* sur une JVM particulière.

Dans un programme utilisant plusieurs *threads*, chaque *thread* Java est exécuté comme s'il était le seul à s'exécuter dans la JVM, lorsque l'ordonnanceur (*scheduler*) des *threads* lui « donne la main ». Le modèle mémoire précise, pour chaque lecture en mémoire, les écritures visibles par celle-ci. Intuitivement, le modèle mémoire détermine les limites des optimisations (voir sections 6.1.1.4 et 6.1.1.5).

6.1.3.2 Problématique de modèle mémoire

Un modèle mémoire met en jeu plusieurs entités (dans notre cas des *threads*) qui sont amenées à partager des données (dans notre cas une zone mémoire partagée).

Bien évidemment le développeur Java aimerait que tous les *threads* aient en permanence la même vision de la mémoire partagée, sans contrainte sur la manière dont il doit programmer. Cependant, un tel modèle d'exécution serait extrêmement coûteux en temps d'exécution, puisqu'il supposerait une synchronisation permanente des processeurs (dans un environnement SMP par exemple). Le modèle mémoire définit un ensemble de règles qui, si elles sont respectées, permettent au développeur d'avoir l'impression que les modifications sont instantanément propagées, tout en permettant d'avoir une implémentation avec de bonnes performances.

Nous n'aborderons pas ici en détail les différents modèles mémoire qui existent, car ceci sortirait du cadre de cette étude [17]. On notera cependant que le modèle mémoire le plus simple qu'on puisse définir est le modèle de « cohérence forte » (*strict consistency*), dans lequel chaque opération de lecture sur un élément réalisé par un *thread* retourne la dernière écriture qui a eu lieu sur cet élément.

6.1.3.3 Programme bien synchronisé en Java

***Data race* dans le modèle mémoire Java** Quand un programme contient deux accès mémoire conflictuels [18] qui ne sont pas ordonnés par la relation *happens-before*, on dit que le programme contient une *data-race*. Un programme Java est dit *correctement synchronisé* si et seulement si toutes les exécutions séquentiellement cohérentes ne contiennent pas de *data-races*.

La notion de cohérence séquentielle évoquée ci-dessus exprime le fait que l'ensemble des actions réalisées par un programme parallèle peut être séquentialisé tout en conservant l'ordre des instructions induit par le programme.

Relation *happens-before* La relation *happens-before* est définie comme suit :

1. Une libération de moniteur (sortie de bloc ou de méthode `synchronized`) induit une relation *happens-before* avec toute acquisition ultérieure du même moniteur (entrée de bloc ou de méthode `synchronized`) ;
2. Une écriture dans un champ `volatile` induit une relation *happens-before* avec toute lecture ultérieure de ce même champ ;
3. Un appel à la méthode `start()` d'un *thread* induit une relation *happens-before* avec

17. Les modèles mémoire sont toujours un sujet d'étude d'aujourd'hui.
18. les deux accès concernent la même adresse et au moins l'un d'entre eux est une écriture.

toute action dans le *thread* démarré ;

4. Toutes les actions dans un *thread* induisent une relation *happens-before* avec n'importe quel autre *thread* qui retourne de l'appel d'une méthode `join()` sur ce *thread* ;
5. Toute action d'un *thread* induit une relation *happens-before* avec les actions ultérieures de ce même *thread*.

De plus, les méthodes de toutes les classes de `java.util.concurrent` et de ses *sous-packages* étendent cette définition aux objets de synchronisation de haut niveau. On se reportera à l'URL suivante pour de plus amples détails : `http://java.sun.com/javase/6/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility`.

6.1.4 Conclusions concernant la programmation concurrente

6.1.4.1 Bénéfices de la programmation concurrente

De manière générale, la programmation concurrente peut apporter plusieurs avantages au développeur Java :

- elle permet de traiter de manière plus naturelle des processus intrinsèquement concurrents ;
- dans le cas de calculs avec des algorithmes intrinsèquement parallélisables, elle permet de tirer partie de la puissance de plusieurs files d'exécutions (plusieurs processeurs ou processeur à coeurs multiples). Toutes les JVMS industrielles sont en effet capables de répartir l'exécution des *threads* sur plusieurs processeurs ;
- dans le cas particulier des applications graphiques, c'est le seul outil à la disposition du développeur pour ne pas bloquer l'interface homme-machine si des traitements longs doivent être effectués (traitements algorithmiques par exemple).

À la différence de langages naturellement conçus pour faciliter la programmation concurrente et la modélisation des processus concurrents tels que Erlang, Java reste très proche dans son esprit de ce que l'on peut trouver avec les bibliothèques d'extensions pour la gestion des *threads* en C et C++. Par conséquent, les *threads* Java ne ressortent pas comme un moyen de gérer les processus concurrents, mais plus comme un moyen de gérer des entrées sorties sur de multiples fichiers ou *sockets* en parallèle. En effet Java, jusqu'à sa version 1.4 et l'ajout des entrées sorties asynchrones dans le *package* `java.nio`, ne permettait pas d'être en attente de lecture ou d'écriture sur plusieurs fichiers en parallèle sans utiliser un *thread* par fichier.

6.1.4.2 Risques

L'écriture d'un programme concurrent correct en Java impose de fortes contraintes sur le développeur Java. En effet, celui-ci doit gérer manuellement la synchronisation des différents *threads* en ce qui concerne les accès aux zones mémoires partagées.

Les risques liés à la programmation concurrente sont principalement des erreurs liées à une synchronisation incorrecte, qui peuvent ensuite déboucher sur des problèmes de sécurité au sens large. En particulier, l'entrelacement des actions réalisées par les *threads* peut briser les invariants sur des portions de code non atomique.

6.1.4.3 Recommandations

La programmation concurrente ne devrait être utilisée que si l'une des conditions suivantes est vraie :

- un gain indispensable en termes de performance est clairement identifié ;
- la modélisation du concept à traiter se traite naturellement sous forme de processus concurrents ;
- les traitements algorithmiques sont trop longs et génèrent un blocage de l'interface homme-machine inacceptable pour l'utilisateur final.

Si la programmation concurrente s'avère indispensable, les recommandations suivantes s'appliquent :

- les échanges *inter-threads* doivent être clairement définis et maîtrisés. Certaines bonnes pratiques peuvent aider à maîtriser les échanges entre *threads*. En particulier, les mécanismes issus du *package* `java.util.concurrent`, tels que ceux implémentant l'interface `java.util.concurrent.Future`, permettent de faciliter l'implémentation d'une tâche de fond effectuant un traitement lourd [19] ;
- chaque appel à une méthode `wait()` sur un objet ou sur une variable de condition doit être suivi d'un test vérifiant si la condition attendue est effectivement avérée ;
- l'utilisation de la méthode `notify()` est déconseillée. Il est conseillé d'appeler la méthode `notifyAll()` ;
- les méthodes obsolètes (ou *deprecated*) de la classe `java.lang.Thread` ne doivent pas être utilisées ;
- utiliser des blocs de code atomique lorsque des invariants doivent être localement assurés.

19. Pour plus de détails, on pourra se référer à la documentation de Sun sur `http://java.sun.com/javase/6/docs/api/java/util/concurrent/Future.html`, ou à l'exemple de code 3 du document [13].

6.2 Contrôle d'accès et authentification

Le contrôle d'accès consiste à spécifier puis mettre en œuvre une politique de sécurité définissant des règles sur l'utilisation des ressources (généralement appelées « objets » [20]) par les sujets ou entités actives du système (les sujets pouvant être considérés comme des objets particuliers). Ce type de mécanisme est généralement implémenté au niveau du système d'exploitation où les objets désignent les conteneurs d'information ou les services proposés par l'OS (par exemple, les fichiers, les IPC, etc.) et les sujets désignent les processus s'exécutant pour le compte d'un utilisateur. L'utilisation des ressources est caractérisée en termes d'opérations réalisées sur les objets.

Les travaux du domaine distinguent deux types de contrôle d'accès :

- le contrôle d'accès discrétionnaire (DAC) où les règles peuvent être définies et modifiées par le sujet propriétaire (généralement le créateur) de l'objet ;
- le contrôle d'accès obligatoire (MAC) où les règles sont définies par un administrateur de sécurité dans un contexte spécifique et s'appliquent obligatoirement aux sujets non privilégiés qui ne peuvent les modifier.

Le premier cas correspond aux droits UNIX traditionnels ou aux mécanismes d'ACL. Le second cas est implémenté dans des *frameworks* de sécurité comme SELinux ou les Trusted Extensions de Solaris.

Il est également possible de distinguer les différentes formes de contrôle d'accès en fonction du type de sujet que l'on souhaite contrôler. En effet, le contrôle d'accès permet potentiellement de restreindre les privilèges de deux types de sujets :

- les utilisateurs « physiques » qui accèdent aux services en se connectant localement ou à distance sur le système ;
- les programmes informatiques qui s'exécutent sur le système, éventuellement pour le compte d'utilisateurs physiques.

Certains mécanismes de contrôle d'accès ne font pas de distinctions entre ces deux types de sujets (les droits d'accès UNIX par exemple ne distinguent pas explicitement les différentes applications s'exécutant pour le compte d'un même utilisateur physique). D'autres, en revanche, prennent en compte ces différents types de sujet ou s'adressent à un type particulier. Ainsi, le contrôle d'accès par rôle (RBAC) s'intéresse principalement aux utilisateurs physiques. Le contrôle d'accès par historique d'exécution (HBAC) s'intéresse quant à lui principalement aux différentes applications exécutées. La distinction entre ces deux types de sujets est importante, car elle implique deux échelles de confiance :

- la confiance dans les utilisateurs physiques qui dépend des aspects organisationnels de la politique de sécurité ;

20. Cette dénomination est *a priori* indépendante de la notion d'objet utilisée dans la POO, par exemple dans Java.

– la confiance dans les programmes informatiques exécutés par les utilisateurs pour accéder à l'information qui dépend de l'origine ou des propriétés intrinsèques des applications.

Dans tous les cas, le contrôle d'accès s'appuie sur un mécanisme d'authentification qui permet de s'assurer de l'identité du sujet (ou de la provenance du code source).

La plate-forme d'exécution Java repose sur différents mécanismes de contrôle d'accès :

1. les mécanismes proposés par l'OS sur lequel s'exécute l'application Java ;
2. le mécanisme de contrôle d'accès par historique d'exécution (HBAC) du JPSA (Java Platform Security Architecture) ;
3. le mécanisme de contrôle d'accès orienté utilisateur de l'API JAAS (Java Authentication and Authorization Services).

Le premier type de mécanismes n'est pas propre à la plate-forme Java. Les processus de la plate-forme d'exécution Java sont en effet soumis au mécanisme de contrôle d'accès fourni par l'OS. Toutefois, ce type de mécanismes ne peut distinguer les accès réalisés par la plate-forme d'exécution pour ses besoins propres (par exemple, lecture d'un fichier de configuration de la JVM) de ceux réalisés par l'application Java. La section 6.2.1 présente à titre d'exemple les interactions entre la plate-forme d'exécution fournie par Sun et les mécanismes de contrôle d'accès standards de Linux.

Le second type de mécanismes, présenté en section 6.2.2, est mis en œuvre nativement par la plate-forme d'exécution Java. Il permet de restreindre les accès des méthodes des classes Java en fonction du type ou de la provenance des classes Java. Ce mécanisme permet également, en collaboration avec le mécanisme de chargement de classes présenté en section 6.3, de cloisonner les différentes classes Java suivant leur type.

Le troisième type de mécanismes, présenté en section 6.2.3, correspond à un ensemble de primitives qui peuvent être utilisées par les développeurs d'applications Java afin de contrôler les accès réalisés par l'application en fonction des utilisateurs de l'application.

6.2.1 Contrôle d'accès fourni par l'OS

Le système d'exploitation implémente généralement un mécanisme de contrôle d'accès discrétionnaire (DAC). Il peut également comprendre des mécanismes de contrôle mandataire (MAC) ou basés sur les rôles (RBAC). Ce type de mécanismes permet de restreindre les accès aux ressources du système natif pour les composants de la plate-forme d'exécution Java ainsi que pour les différentes applications Java. En ce qui concerne ces dernières, deux modèles d'exécution peuvent être distingués :

1. les applications Java s'exécutent au sein de différents processus du système d'exploitation. Lorsqu'une JVM est utilisée, chaque application s'exécute dans une instance distincte de la JVM ;

2. les applications Java s'exécutent au sein d'un unique processus du système d'exploitation et partage une instance commune de la JVM.

La spécification du langage Java et de l'API de la bibliothèque standard n'apporte aucune précision sur l'interface entre les applications Java et le système de contrôle d'accès éventuellement fourni par l'OS. Chaque implémentation de la plate-forme d'exécution adopte sa propre stratégie. Toutefois, le premier modèle est le plus courant pour le type d'applications visé par cette étude. Le deuxième modèle est principalement utilisé dans trois cas de figure qui sortent du périmètre de cette étude :

- dans les applications de type *applet* exécutées à partir d'un navigateur ;
- dans les applications embarquées (J2ME), pour des raisons d'optimisation (une seule instance de la JVM est présente en mémoire) ;
- dans les serveurs d'application (J2EE) qui peuvent exécuter plusieurs « applications » Java.

Cette étude repose donc sur l'hypothèse que le contrôle d'accès fourni par l'OS peut distinguer les différentes applications Java, exécutées dans des instances distinctes de la JVM (c'est notamment le cas pour des applications exécutées sur le JRE de Sun sous Linux). L'analyse des mécanismes de contrôle d'accès fournis par l'OS (qui est propre à chaque instance de la plate-forme d'exécution native) n'entre pas dans le périmètre de cette étude. Toutefois, il paraît pertinent d'étudier les interactions entre la plate-forme d'exécution Java et les mécanismes de sécurité fournis par l'OS (mécanismes « standards » et mécanismes de durcissement). Ces interactions étant propres à l'implémentation du modèle d'exécution, elles seront présentées plus en détail dans le « Rapport sur les modèles d'exécution de Java » [14] et dans le rapport « Comparatif des JVM » [11]. À titre d'exemple, nous présentons succinctement par la suite l'interaction entre le mécanisme de contrôle d'accès discrétionnaire de Linux et les applications Java exécutées à partir du JRE de Sun.

La configuration des droits relatifs à la plate-forme Java dépend en théorie de chaque distribution Linux. En pratique, la configuration par défaut est la suivante (il s'agit notamment de la configuration adoptée pour le *package* officiel du JRE de Sun pour la distribution Debian) :

- la plupart des fichiers de la plate-forme d'exécution (situés dans le répertoire `/usr/lib/jvm/<java-version>` pour la distribution Debian) sont la propriété de l'utilisateur *root*. Il s'agit de fichiers exécutables (dont la commande `java`), de bibliothèques natives partagées (dont la JVM), de bibliothèques Java (dont l'archive `rt.jar` comprenant la plupart des classes de la bibliothèque standard) et de fichiers de configuration. Ces fichiers ne sont modifiables que par leur propriétaire (*root*). Ils sont en revanche lisibles par l'ensemble des utilisateurs du système. Les exécutables peuvent être exécutés par l'ensemble des utilisateurs ;
- la configuration globale du JRE peut être adaptée pour chaque utilisateur du système disposant d'un répertoire utilisateur via des fichiers de configuration situés dans le répertoire de l'utilisateur ;
- la configuration des droits pour les fichiers des applications et les bibliothèques installées par l'utilisateur sont à la discrétion de l'utilisateur.

Par défaut, les applications Java s'exécutent sous l'identité de l'utilisateur qui a exécuté la commande `java`. Il s'agit généralement de l'identité d'un utilisateur physique qui s'est connecté sous une session de l'OS. Il peut également s'agir d'une identité *ad hoc* pour les applications de type *daemon*. Le contrôle d'accès de l'OS permet de cloisonner les différentes applications Java dès lors qu'il peut distinguer les différents processus qui correspondent à ces applications. Par exemple, le contrôle d'accès discrétionnaire de Linux distingue les différentes applications Java dès lors qu'elles sont exécutées sous des identités (utilisateur ou groupe) différentes. Dans tous les cas, une forme limitée de cloisonnement est assurée par le mécanisme de gestion des processus puisque les applications correspondent à différents processus (elles ont donc des espaces mémoire virtuels distincts). Toutefois, celles-ci peuvent communiquer via les mécanismes de type *Inter Process Communication*. Il convient donc de restreindre les accès à ce type de mécanisme, comme pour toutes applications s'exécutant sur l'OS. Cependant, certains mécanismes de contrôle d'accès (notamment SELinux) s'appuient sur les fichiers exécutables pour distinguer les applications. Or, dans un environnement Java SE standard pour Linux, les différentes applications Java correspondent au même fichier exécutable (la commande `java`). Il s'agit donc d'une limite qui peut être contournée de différentes manières :

- il est possible de fournir avec chaque application un lanceur adapté à l'application. Ce lanceur est un exécutable (développé par exemple en C/C++) qui remplace la commande `java`, charge la JVM (qui est une bibliothèque partagée) et exécute le chargeur de classes initial sur la classe principale de l'application [21]. Les différentes applications peuvent alors être distinguées. Ce type d'approche nécessite un effort de développement supplémentaire. Toutefois, il s'agit d'une solution générique qui permet de s'adapter aux différents systèmes de contrôle d'accès ;
- il est parfois possible d'exécuter un même exécutable sous différents contextes de sécurité à l'aide d'une commande adéquate (par exemple, `runcon` pour SELinux). Toutefois, cette solution nécessite de préciser ce contexte explicitement lors de chaque exécution de l'application. Par exemple, il n'est pas possible d'utiliser le mécanisme de changement de domaine automatique (transition de domaine) de SELinux.

6.2.2 Java Plateform Security Architecture

La bibliothèque standard de Java propose un mécanisme de contrôle d'accès fourni par un ensemble de classes regroupées au sein de la *Java Plateform Security Architecture* [22]. Ces classes sont principalement situées dans le *package* `java.security`. Ce mécanisme permet de contrôler les accès des différentes classes Java (donc des différentes applications) en fonction du code exécuté. En revanche, ce mécanisme s'appuie uniquement sur le code et ne peut donc distinguer différentes instances d'une même classe ou différentes occurences d'une même application.

21. Ce programme s'appuie sur l'API JNI qui permet « d'embarquer » la JVM au sein d'une application native.
22. `http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html`

Concrètement, les intérêts du mécanisme de contrôle d'accès de Java, dans le cadre de JAVASEC, sont les suivants :

- il permet d'appliquer des restrictions différentes en fonction de l'application Java exécutée, notamment lorsque le mécanisme de contrôle d'accès de l'OS ne peut distinguer ces applications (par exemple lorsqu'elles sont exécutées sous la même identité, au sens du contrôle d'accès de l'OS [23]) ;
- il permet de restreindre les droits de l'application Java par rapport à ceux de la JVM (il est par exemple envisageable de permettre à la JVM de lire les fichiers de configuration dont elle a besoin, mais d'en interdire l'accès à l'application Java qu'elle exécute) ;
- il permet également, au sein d'une même application Java, d'adapter les privilèges des différents blocs fonctionnels afin d'appliquer le principe de minimum de privilèges. Ceci permet notamment de limiter l'impact de l'exploitation d'éventuelles vulnérabilités ou de classes malicieuses (ou piégées) fournies par une bibliothèque tierce ;
- outre l'accès aux ressources natives, ce mécanisme permet de restreindre l'utilisation de certains mécanismes ou méthodes Java dont l'utilisation peut s'avérer dangereuse pour la sécurité et qu'il est le seul à pouvoir contrôler : la réflexion (qui permet notamment, en l'absence de restriction du contrôle d'accès, de contourner les règles de visibilité), le chargement de classes (qui peut permettre de piéger l'application), le chargement de bibliothèques natives, etc. Le mécanisme doit également restreindre les accès aux classes participant à la mise en œuvre du contrôle (les chargeurs de classes, le gestionnaire de sécurité, etc.) afin qu'une classe vulnérable, piégée ou malicieuse ne puisse modifier son comportement. Cette forme d'auto-protection est nécessaire du fait que le mécanisme est quasi-entièrement implémenté en Java, la JVM n'offrant qu'un support minimal.

Si, dans la plupart des cas, le développeur doit se contenter de spécifier la politique de contrôle d'accès, il peut parfois être amené à implémenter des points de contrôle ou à modifier des classes. Dans tous les cas, la mise en place d'un tel contrôle nécessite un effort de structuration du code lors de la conception afin de pouvoir distinguer clairement les différents blocs fonctionnels et déterminer pour chacun d'eux le minimum de privilèges requis. Cet effort relève toutefois des bonnes pratiques de programmation. Dans le cadre de la conception d'applications de sécurité à haut niveau de confiance, cet effort paraît nécessaire. Les recommandations du rapport [13] vont dans ce sens.

6.2.2.1 Description de l'architecture

Un des buts initiaux de Java était de pouvoir exécuter du code distant, téléchargé sur un réseau informatique (notamment sous forme d'*applet*). La plate-forme d'exécution comprend donc, dès les premières versions de Java, un mécanisme permettant de contrôler les accès des

23. Toutefois, le mécanisme de contrôle d'accès de Java ne peut distinguer différentes exécutions d'une même application Java puisqu'il repose sur le code de l'application.

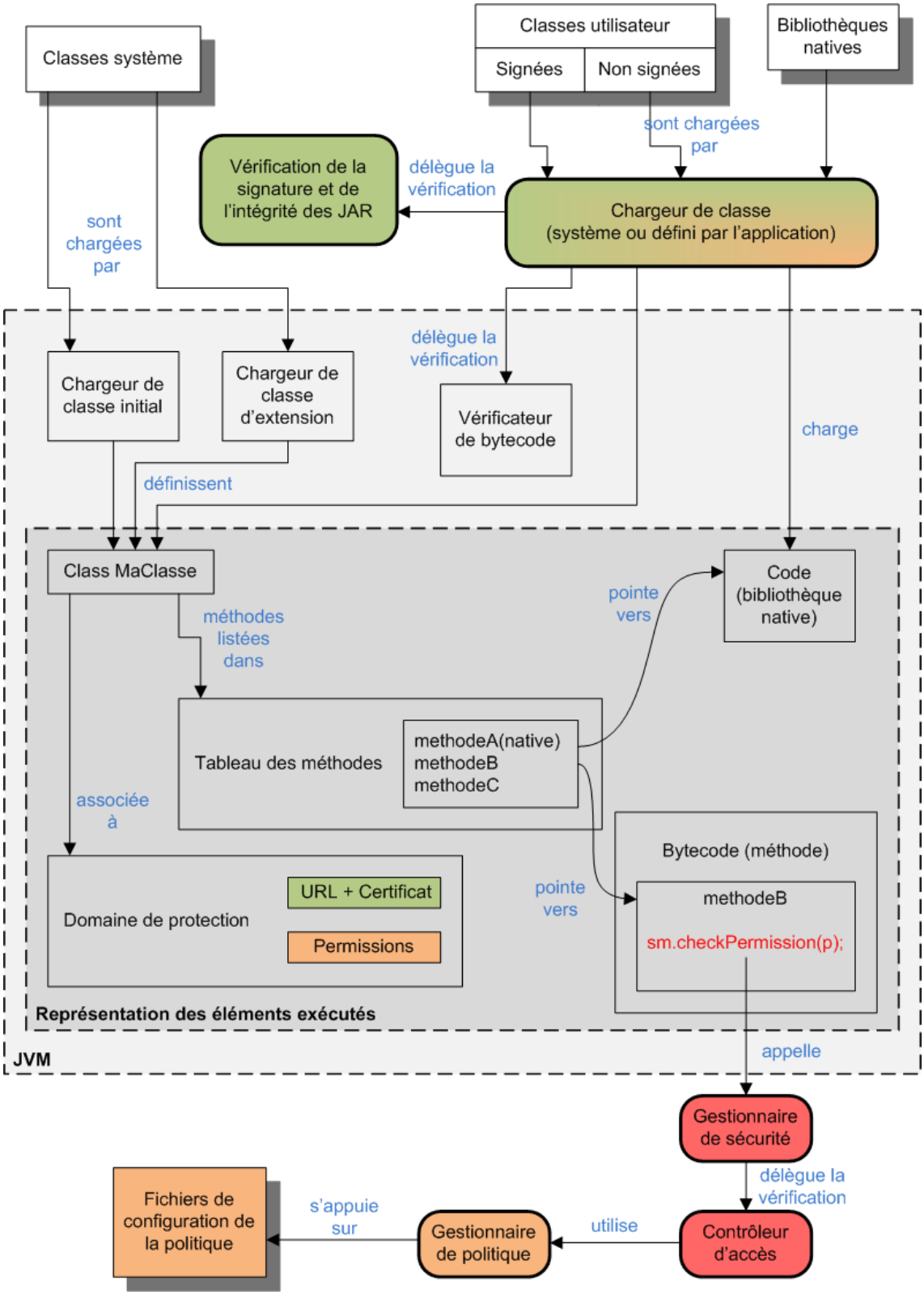


FIGURE 8 – Nouvelle architecture générique du système de contrôle d'accès de Java.

classes Java aux ressources de la plate-forme native (fichiers, *sockets*, etc.) ainsi qu'aux méthodes sensibles (introspection, paramétrage du contrôle d'accès, etc.). Ce contrôle s'effectue en fonction de l'origine de la classe. Initialement (version 1.0 et 1.1 du JDK), ce mécanisme ne distinguait que deux types d'applications :

- les applications de confiance, dont les classes sont présentes « localement » sur le système ou dont le fournisseur est de confiance (depuis le JDK 1.1 via le mécanisme de signature des *applets*) ;
- les *applets* téléchargées sur le réseau (Internet) dont le fournisseur n'a pu être identifié ou n'est pas un fournisseur de confiance.

Ce mécanisme simple permettait de se prémunir, ou tout du moins de limiter l'impact, des *applets* hostiles ou vulnérables téléchargées depuis Internet. Il ne fait en revanche aucune distinction entre les différentes classes « locales ». De manière générale, ce mécanisme souffrait d'un manque de souplesse et de polyvalence. Il a donc été profondément remanié dans la version 2 (1.2) du JDK. Cette nouvelle implémentation, illustrée par la figure 8, a par la suite fait l'objet de quelques modifications et ajouts dans les versions successives du JDK, mais l'architecture globale a été maintenue. Trois types de composants peuvent être distingués :

- les composants qui participent à l'identification et l'authentification des classes (identifiés en vert sur la figure 8) ;
- les composants qui permettent de spécifier la politique de contrôle d'accès (identifiés en orange sur la figure 8) ;
- les composants qui appliquent les règles de contrôle d'accès (identifiés en rouge sur la figure 8).

Le contrôle d'accès standard de la plate-forme d'exécution Java est de type « orienté code ». Il vise à restreindre les accès des méthodes des classes Java en fonction de l'origine du code exécuté. Cela suppose dans un premier temps d'identifier (et dans l'idéal d'authentifier) le code exécuté. La granularité retenue par les concepteurs de Java pour l'identification du code correspond à la classe. Différentes méthodes définies au sein d'une même classe se verront donc attribuer des règles de contrôle d'accès identiques. De plus, seul le code est identifié. Aussi, les différentes instances d'une même classe sont traitées de la même manière. L'identification des classes est réalisée par un composant qui n'est pas dédié au contrôle d'accès : le chargeur de classes. Ce composant est décrit plus en détail en section 6.3. Il s'appuie, pour authentifier le code, sur le mécanisme de signature d'archives JAR, décrit en section 6.2.5. L'« identité » d'une classe correspond en fait à son origine, identifiée à la fois en termes d'URL et de certificats des fournisseurs de la classe (si celle-ci est distribuée à l'aide d'une archive JAR signée). Cette origine est liée par un domaine de protection à l'instance de la classe `java.lang.Class` représentant la classe chargée. Les notions de domaine de protection et d'origine de code sont présentées plus en détail annexe 10.1, page 204.

La spécification de la politique de contrôle d'accès repose principalement sur un composant de gestion de la politique [24]. Lorsque les mécanismes par défaut sont utilisés, ce composant est

24. Le chargeur de classes peut en partie définir la politique de contrôle comme évoqué en section 6.3.

implémenté par une instance de la classe `java.security.Policy`. Celle-ci s'appuie elle-même sur des fichiers textes permettant à un administrateur de spécifier la politique. Une politique de sécurité Java repose sur l'attribution de permissions à un ensemble de classes identifiées par une origine de code.

La mise en œuvre du contrôle s'appuie sur deux composants : le gestionnaire de sécurité qui, par défaut, délègue les vérifications au contrôleur d'accès qui implémente l'algorithme de contrôle d'accès [25]. Les vérifications sont initiées par des points de contrôles qui doivent être explicitement spécifiés dans le code des bibliothèques et des applications Java. La bibliothèque standard implémente un certain nombre de points de contrôle (notamment lors de l'accès aux ressources natives). Le développeur d'applications peut (doit, s'il implémente des méthodes critiques, notamment des méthodes natives) implémenter ses propres points de contrôle.

Cette architecture vise notamment à séparer la spécification de la politique de la mise en œuvre du contrôle d'accès. Les différents éléments de spécification et de mise en œuvre sont pour la plupart configurables et modifiables (le développeur d'application Java peut modifier l'implémentation des différents éléments du mécanisme de contrôle d'accès de Java en utilisant la notion d'héritage). Cette souplesse permet de s'adapter facilement aux différents besoins de contrôle d'accès de l'utilisateur. Elle peut cependant mener à des déviations par rapport au comportement des implémentations de référence. De plus, la relative complexité de l'architecture peut mener à des failles de sécurité lors du déploiement ou de la modification des mécanismes natifs.

6.2.2.2 Mise en œuvre du contrôle

Les principales étapes de la mise en œuvre du contrôle d'accès peuvent être résumées ainsi :

1. le chargeur de classes détermine le fichier de la classe à charger et fournit le *bytecode* à la JVM sous la forme d'un tableau d'octets. Ce *bytecode* est normalement vérifié par le vérificateur de *bytecode* (en pratique ce n'est pas toujours le cas pour les classes de la bibliothèque standard) ;
2. la signature de la classe est vérifiée (si la classe est signée) et l'origine du code est identifiée (construction d'un objet `CodeSource`) ;
3. éventuellement, le chargeur de classes spécifie un ensemble de permissions statiques qui s'appliquent à la classe [26] ;
4. une instance de domaine de protection correspondant est associée à la classe (cette instance est éventuellement créée, si aucune classe précédemment chargée n'est liée à ce domaine de protection) et l'objet `Class` représentant la classe chargée est créé ;

25. En toute rigueur celui-ci délègue le contrôle à une instance de la classe `java.security.AccessControlContext` représentant le contexte d'exécution pour lequel les vérifications sont effectuées.

26. Ces permissions sont en général spécifiées explicitement dans le code du chargeur de classes. La politique de contrôle d'accès est alors modifiée, sans que cela apparaisse explicitement dans le fichier de spécification de cette politique (seules des autorisations supplémentaires peuvent avoir été ajoutées)

5. lors de l'exécution d'une méthode entraînant une vérification d'accès, le mécanisme de vérification prend en compte les permissions (statiques et dynamiques) des différentes classes dont les méthodes appartiennent à la chaîne d'appels du fil d'exécution courant. Cette vérification donne lieu à une exception de type `java.java.lang.SecurityException` si l'accès doit être refusé.

La phase de vérification à proprement parler s'appuie sur différents éléments :

- les **points de contrôle** insérés par exemple dans les méthodes de la bibliothèque standard permettant d'accéder aux ressources de la plate-forme native ou constituant des méthodes « critiques » (telles que les méthodes permettant la programmation réflexive ou celles permettant de modifier le mécanisme de contrôle d'accès) ;
- le **gestionnaire de sécurité** qui centralise les demandes de vérification émises par les différents points de contrôle et qui, dans l'implémentation utilisée par défaut, délègue la vérification au contrôleur d'accès ;
- le **contrôleur d'accès** qui implémente l'algorithme de vérification des permissions ;
- les **actions privilégiées** qui permettent au développeur de soustraire certaines méthodes de l'algorithme de contrôle d'accès afin d'augmenter temporairement leurs privilèges (ce type d'éléments peut être assimilé à l'appel système `su` sous UNIX).

Points de contrôle Les points de contrôle doivent être implémentés explicitement par les développeurs des bibliothèques et des applications Java. Il s'agit d'appels aux méthodes de vérification du gestionnaire de sécurité décrites dans le paragraphe suivant [27]. Les classes de la bibliothèque standard implémentent des points de contrôle permettant notamment de protéger l'accès aux ressources natives (accès aux fichiers, au réseau, etc.) via les méthodes offertes par la bibliothèque standard. Lorsqu'un développeur d'une bibliothèque ou d'une application implémente ses propres méthodes d'accès aux ressources natives (qui ne font pas appels aux méthodes de la bibliothèque standard), il doit donc implémenter ses propres points de contrôle.

L'implémentation d'un point de contrôle est réalisée en deux étapes :

1. obtenir une référence sur le gestionnaire de sécurité de l'instance de la JVM exécutant l'application (appel à la méthode `getSecurityManager` de la classe `java.lang.System`) ;
2. si l'application est surveillée par un gestionnaire de sécurité (c'est-à-dire si la référence est non nulle [28]), appeller une méthode de vérification du gestionnaire de sécurité en lui passant en paramètre les permissions minimales nécessaires pour effectuer l'action qui fait l'objet du contrôle.

27. En toute rigueur, il est possible d'appeler directement les méthodes du contrôleur d'accès. Dans la majorité des cas, ceci a peu d'impact sur le processus de contrôle puisque le gestionnaire de sécurité délègue les vérifications au contrôleur d'accès. Toutefois, il n'est pas recommandé de contourner le gestionnaire de sécurité puique celui-ci peut insérer des contrôles supplémentaires ou modifier la mise en œuvre du contrôle.

28. En pratique, pour des applications de sécurité, un gestionnaire de sécurité devrait être installé systématiquement pour une utilisation en production. Le rapport [13] fournit une recommandation dans ce sens.

L'exemple suivant illustre le développement d'un point de contrôle pour la vérification lors de la lecture d'un fichier du système d'exploitation.

```
SecurityManager security = System.getSecurityManager();
if (security != null) {
  security.check(new FilePermission(name, "read"));
}
```

Gestionnaire de sécurité Le gestionnaire de sécurité, implémenté par une instance de la classe `java.lang.SecurityManager` (ou d'une classe héritant de cette classe) centralise les demandes de vérifications puisque les différents points de contrôle font appel à ses méthodes [29]. Depuis la version 2 de Java, les vérifications à proprement parler sont déléguées au contrôleur d'accès. Ce comportement est implémenté par la classe `java.lang.SecurityManager`, qui n'est pas une classe abstraite. Toutefois, cette classe n'est pas déclarée *final* et le comportement par défaut peut être modifié par un développeur qui implémente et utilise une classe héritant de `java.lang.SecurityManager`.

L'implémentation initiale du contrôle d'accès, dans les versions antérieures à la version 2 de Java, s'appuyaient sur différentes méthodes de la classe `java.lang.SecurityManager`. Ces méthodes dépendent du type d'accès (par exemple, `checkRead(FileDescriptor fd)`, `checkWrite(FileDescriptor fd)`, etc.). Dans la nouvelle architecture, l'ensemble des vérifications fait appel à la méthode `checkPermission( Permission perm)` qui repose sur le concept des permissions. Les autres méthodes sont toujours présentes par souci de compatibilité, mais leur implémentation fait appel à la méthode `checkPermission` et il est maintenant recommandé d'utiliser exclusivement cette méthode.

Une seule instance de gestionnaire de sécurité est active à un instant donné sur une instance d'une JVM. L'instance active est celle prise en compte par le mécanisme de contrôle d'accès. Elle est mise en place à l'aide de la méthode `setSecurityManager(final SecurityManager s)` de la classe `java.lang.System`. N'importe quelle méthode d'une classe de l'application Java peut appeler cette méthode, pour peu qu'elle dispose des privilèges nécessaires [30], l'installation d'un gestionnaire de sécurité ne nécessitant pas de privilège si au préalable aucun gestionnaire n'est actif. Le gestionnaire de sécurité par défaut peut également être activé lors du lancement d'applications « locales » à l'aide d'un paramètre : `java -Djava.security.manager`. Par défaut (pour l'implémentation de Sun du JRE), le paramétrage du gestionnaire de sécurité est le suivant :

- les applications « locales » Java, exécutées à l'aide de la commande `java`, n'utilisent pas de gestionnaire de sécurité (dans ce cas, la méthode `getSecurityManager()` de la classe `java.lang.System` renvoie `null`) ;
- les *applets* exécutées à l'aide de la commande `appletviewer` ou à l'aide du plug-in Java du navigateur Web utilisent obligatoirement le gestionnaire de sécurité par défaut

29. En pratique, certaines méthodes de la bibliothèque standard font appels directement au contrôleur d'accès, mais ceci n'est pas conseillé, sauf lorsque cela est nécessaire (par exemple, pour utiliser un contexte différent du contexte par défaut)

30. Permission de type `RuntimePermission("setSecurityManager")`.

qui met en œuvre par défaut la politique de bac-à-sable implémentée « en dur » dans les versions antérieures de Java.

Il est donc nécessaire d'activer explicitement le gestionnaire de sécurité afin de surveiller les applications locales. Ceci peut être réalisé par l'administrateur ou l'utilisateur en passant un paramètre à la commande `java`. Le développeur peut également installer explicitement le gestionnaire de sécurité lors du démarrage de l'application.

Si l'accès est autorisé, l'exécution continue dans la méthode où est implémenté le point de contrôle. Si l'accès doit être interdit, c'est-à-dire si le code exécuté ne possède pas les permissions suffisantes, une exception de type `java.lang.SecurityException` est levée. Il s'agit d'un type d'exception héritant de `java.lang.RuntimeException`, qui ne nécessite donc pas d'être interceptée ni transmise explicitement (présence de `throw` dans la signature de la méthode). Il est toutefois possible d'intercepter ce type d'exception bien que cela ne soit pas nécessaire (ni souhaité dans certains cas). En effet, dans la philosophie Java, les exceptions de type *runtime* sont dédiées aux erreurs qu'il n'est pas toujours possible de recouvrir, à l'inverse des *checked exception* [34], qui elles nécessitent un traitement (par défaut, par le code de la méthode qui a généré l'exception ou, s'il refuse ce « contrat », par le code d'une des méthodes appelantes). Cependant, dans le cas précis des exceptions de sécurité, le développeur peut dans certains cas être amené à rattraper ce type d'exception afin de modifier le comportement de l'application en cas d'accès refusé (tentative d'accès à une autre ressource).

Contrôleur d'accès Le contrôleur d'accès, implémenté par la classe `java.security.AccessControler`, applique l'algorithme de vérification des accès. Il s'appuie pour cela sur les domaines de protections associés à chaque classe, sur le gestionnaire de la politique de sécurité (une implémentation de la classe `java.security.Policy`) qui permet d'obtenir les permissions dynamiques associées à chaque domaine et sur le contexte d'exécution (une implémentation de la classe `java.security.AccessControlContext`). Contrairement au gestionnaire de sécurité, la classe `java.security.AccessControler` est déclarée *final*. De plus, cette classe n'a pas vocation à être instanciée (ses constructeurs étant déclarés *private*). L'algorithme de contrôle d'accès n'est donc pas modifiable directement (il est toutefois possible de modifier l'algorithme en utilisant une implémentation du gestionnaire de sécurité qui ne délègue pas les vérifications au contrôleur d'accès). La mise en œuvre du contrôle d'accès se fait via l'appel des méthodes statiques du contrôleur d'accès, qui sont donc accessibles directement depuis n'importe quelle classe (à l'inverse du gestionnaire de sécurité).

La vérification est réalisée par l'appel à la méthode statique `checkPermission(Permission perm)`. Cette méthode est généralement appelée par l'instance du gestionnaire de sécurité qui surveille le système. La permission passée en paramètre correspond aux droits nécessaires pour réaliser l'accès contrôlé. L'algorithme de contrôle d'accès consiste donc à vérifier que l'ensemble du code qui a conduit à l'accès contrôlé possède au moins la permission requise. En première approche, l'algorithme réalise dans la plupart des cas les opérations suivantes :

1. il détermine la pile des appels des méthodes pour le *thread* courant (celui qui tente d'accéder à la ressource contrôlée). Le bas de la pile est constitué par la méthode qui a initié le *thread* (`main` pour le *thread* principal de l'application, `run` pour les autres

threads). Le sommet est constitué par l'appel à la méthode `checkPermission` de la classe `java.security.AccessControler` ;

2. il détermine, pour chaque méthode de la pile d'appels, la classe qui définit la méthode [31] ainsi que son domaine de protection associé ;
3. il analyse chaque domaine de protection en partant du sommet de la pile et vérifie que chaque domaine de protection implique [32] la permission nécessaire pour réaliser l'accès controlé.

Les deux premières étapes de l'algorithme sont implémentées par la JVM. Celle-ci doit donc fournir une interface permettant aux méthodes Java qui mettent en œuvre le contrôle d'accès d'obtenir la liste des domaines de protection correspondant à un contexte d'exécution donné. Cette interface est fournie par la méthode native privée `getStackAccessControlContext()` de la classe `java.security.AccessController` qui est appellée par différentes méthodes du contrôleur d'accès. Cette méthode retourne un contexte d'exécution (instance de la classe `java.security.AccessControlContext`). C'est cette classe qui implémente réellement la dernière étape de l'algorithme au sein de la méthode `checkPermission`.

Concrètement, cet algorithme s'assure que toutes les méthodes du *thread* courant (celui qui réalise l'accès controlé) possèdent les permissions nécessaires pour réaliser l'accès. Soit D_i le domaine de protection associé à la méthode i de la pile d'appel du *thread* courant (qui comprend N méthodes). Soit P_i l'ensemble des permissions impliquées par le domaine D_i : $P_i = \{p / D_i \rightarrow p\}$. L'algorithme simplifié de vérification de la permission p_v consiste donc à s'assurer que la propriété suivante est vérifiée :

$$p_v \in \left\{ \bigcap_{i=1}^{N} P_i \right\}$$

Cette règle est très restrictive puisque non seulement la méthode réalisant l'accès doit posséder la permission, mais également l'ensemble des méthodes de la pile d'appels. Il est parfois nécessaire de la contourner en partie afin d'exprimer certaines restrictions moins strictes. En effet, dans certains cas, il parait souhaitable de restreindre l'accès direct à une ressource sensible à du code de confiance (dont on maîtrise le développement, qui a été audité, etc.) et d'en interdire l'accès direct au code distant ou fourni par une application tierce. Toutefois, il peut être souhaitable que ce code tierce accède indirectement à la ressource, en utilisant les primitives fournies par le code de confiance (qui doit filtrer les paramètres et les données retournées par ses primitives). Deux mécanismes permettent de modifier cet algorithme de vérification :

– l'utilisation d'un contexte de contrôle d'accès différent du contexte courant lors de la vérification des permissions ;

31. La classe qui définit la méthode est celle où le code de la méthode est spécifié. Ainsi, si une classe `B` hérite d'une classe `A` et que `B` ne redéfinit pas la méthode `m` de `A`, alors les appels à `A.m` et `B.m` sont toujours associés aux permissions de `A` (la classe qui définit la méthode). En revanche, si `B` redéfinit la méthode `m`, alors l'appel de `B.m` sera associé aux permissions de `B`.

32. Un domaine D « implique » une permission p (noté $D \rightarrow p$) si une des permissions explicitement attachées au domaine implique la permission p ($\exists p_i \in D$ telle que $p_i \rightarrow p$).

– les actions privilégiées, implémentées par des instances d'une classe héritant de l'interface `java.security.PrivilegedAction<T>`, permettent de soustraire certaines méthodes privilégiées de l'algorithme de vérification des permissions.

Le premier mécanisme est principalement utilisé lors de la création de nouveaux *threads*. Certaines vérifications nécessitent alors de prendre en compte non seulement le contexte du nouveau *thread* créé, mais également celui du *thread* père qui l'a créé. En dehors de ce cas particulier, ce mécanisme est rarement utilisé. Il n'est donc pas détaillé dans ce document. La description et la mise en œuvre de ce mécanisme sont notamment explicitées dans l'ouvrage *Inside Java 2 Platform Security* [35]. Il convient toutefois de noter que ce mécanisme doit être utilisé avec prudence. En effet, il nécessite que le code Java sauvegarde un contexte afin de l'utiliser par la suite (à l'inverse de la procédure par défaut où le contexte est fourni directement par la JVM). Il est donc nécessaire de limiter la visibilité et les accès à ce contexte afin d'éviter qu'un code vulnérable, malveillant ou piégé puisse le modifier. Le problème ne se pose que pour les points de contrôle qui font explicitement appel à ce type de mécanisme (la majorité des points de contrôle de la bibliothèque standard utilisant la procédure par défaut, où le contexte est fourni directement par la JVM). De plus, l'utilisation de ce mécanisme nécessite un développement particulier du point de contrôle : il n'est pas possible pour un code Java de modifier le contexte utilisé par défaut (fourni par la JVM) sans modifier le code de la méthode implémentant le point d'accès.

Le second mécanisme est en revanche plus couramment utilisé puisqu'il permet notamment d'appliquer le principe de minimum de privilèges. Il est décrit plus en détail dans le paragraphe suivant.

Actions privilégiées Les actions privilégiées permettent de répondre aux besoins suivants : certaines méthodes peuvent être amenées à accéder « indirectement » à certaines ressources (en appelant d'autres méthodes), mais ne doivent pas accéder directement à ces ressources (leur code n'est pas de confiance). Par exemple, une *applet* est généralement considérée comme non sûre et ne possède pas les droits lui permettant d'accéder aux ressources de la plateforme d'exécution native (par exemple, les fichiers du système d'exploitation). Toutefois, cette *applet* peut faire appel aux services fournis par une bibliothèque « locale » (la bibliothèque standard ou une bibliothèque tierce) qui nécessite d'accéder aux ressources natives. Par exemple, l'*applet* peut jouer des fichiers audio en s'appuyant sur une bibliothèque adéquate : le code de l'*applet* n'accède pas directement aux fichiers audio, mais il fait appel au code de la bibliothèque qui lui doit pouvoir accéder à ces fichiers. Suivant l'algorithme de contrôle d'accès, l'ensemble du code (celui de l'*applet* et celui de la bibliothèque) doit posséder les permissions afin d'accéder à ces fichiers. Plutôt que de modifier la politique et d'octroyer les permissions à l'ensemble du code, les actions privilégiées permettent de ne prendre en compte temporairement que le code de la bibliothèque lors de la vérification. Ainsi, seule la bibliothèque nécessite les permissions de lecture des fichiers audio.

Les actions privilégiées permettent donc d'appliquer le principe de minimum de privilèges. Toutefois, cela suppose de faire confiance au code privilégié : les méthodes exécutées de la sorte doivent filtrer correctement leurs paramètres et s'assurer que le code appelant n'obtient

pas un contrôle important sur la ressource accédée (ce qui reviendrait à permettre au code non privilégié d'accéder directement à la ressource contrôlée). D'après l'exemple précédent, les méthodes de la bibliothèque appelées par le code de l'*applet* doivent s'assurer que ce dernier ne puisse spécifier n'importe quel type de fichier et surtout, ces méthodes ne doivent pas retourner directement une référence sur le fichier ouvert. En effet, pour des raisons de performance (l'algorithme de contrôle d'accès de Java est relativement lent), les points de contrôle sont généralement implémentés sur les méthodes permettant d'obtenir une référence sur la ressource et non sur celles effectuant les accès. C'est notamment le cas pour les fichiers où le point de contrôle est implémenté à l'ouverture du fichier et non pour chaque lecture ou écriture. Typiquement, ces méthodes publiques de l'API qui implémentent les points de contrôle assurent elles-mêmes l'interface avec la plate-forme native, via des méthodes natives et privées, afin de jouer les fichiers audio.

Le passage en mode privilégié nécessite de définir et d'instancier une classe implémentant l'interface `java.security.PrivilegedAction<T>`. Cette interface possède une seule méthode, `run`, qui est exécutée lors du passage en mode privilégié. Cette méthode, ainsi que l'ensemble des méthodes qui découlent de son exécution, sont exécutées en mode privilégié. Lorsqu'elle retourne, l'exécution se poursuit dans le mode initial (non privilégié dans la plupart des cas). Le passage en mode privilégié à proprement parler est effectué par un appel à la méthode `doPrivileged` de la classe `java.security.AccessController` en passant une instance de la classe implémentant l'action privilégiée en paramètre. La méthode `doPrivileged` est une méthode native qui fait appel à une interface de la JVM. Celle-ci prend en compte cette information et l'utilise lorsqu'elle doit générer un contexte d'exécution (lors des futures vérifications).

Les actions privilégiées n'augmentent pas l'ensemble des permissions du code exécuté en mode privilégié. Elles permettent cependant de ne pas prendre en compte les domaines de sécurité du code appelant. Lors de la vérification des permissions, l'algorithme de contrôle d'accès identifie la première occurrence d'un appel à `doPrivileged` (en partant du haut de la pile). Seuls les domaines de sécurité des méthodes du haut de la pile d'appels qui sont situées au dessus de cet appel sont pris en compte par le domaine de sécurité. Pour une pile d'appels de N méthodes, en supposant que la première occurence d'un appel à `doPrivileged` corresponde à la p$^{\text{ème}}$ méthode (la première méthode étant celle qui correspond à l'appel le « plus récent » et qui implémente le point de contrôle), l'algorithme de vérification de la permission p_v consiste donc à vérifier la propriété suivante :

$$p_v \in \left\{ \bigcap_{i=1}^{p} D_i \right\}$$

Il est important de noter que la dernière méthode prise en compte lors d'un contrôle d'accès dans une action privilégiée (la p$^{\text{ème}}$ méthode) correspond à la méthode qui a appelé `doPrivileged` et non celle appelée par `run`. Les permissions associées à la classe implémentant l'action privilégiée sont donc prises en compte (la méthode `run` étant incluse dans le contrôle). En théorie, les permissions associées à la classe `SecurityManager` sont également prises en compte du fait de l'appel à `doPrivileged`. Toutefois, d'après la spécification actuelle du mécanisme de contrôle d'accès, les classes de la bibliothèque standard, qui sont chargées par le chargeur de

classe initial, possède toutes les permissions. Plus exactement, aucun domaine de protection n'est associé à ces classes, ce qui est interprété par l'algorithme de contrôle d'accès comme la possession de la permission `AllPermission`.

6.2.2.3 Analyse de l'architecture

Plusieurs aspects caractérisent cette nouvelle architecture :

- les mécanismes de spécification et de mise en œuvre de la politique du contrôle d'accès sont séparés (dans les premières versions de Java, cette séparation était inexistante) ;
- les décisions de l'algorithme de contrôle d'accès sont indépendantes de la sémantique des permissions ce qui permet une certaine souplesse dans l'utilisation du mécanisme (le développeur peut définir ces propres autorisations et points de contrôle tout en réutilisant le mécanisme de contrôle d'accès standard de la plate-forme d'exécution) ;
- les permissions reposent sur les principes d'encapsulation et d'héritage pour masquer les aspects propres à chaque type d'accès ou d'objet accédé ;
- le mécanisme de contrôle d'accès est fortement dépendant du mécanisme de chargement de classe.

L'implémentation actuelle du mécanisme de contrôle d'accès sous Java se veut souple et modulaire. Elle amène cependant un certain nombre de critiques.

Il est difficile d'assurer des interdictions. La spécification de la politique repose uniquement sur la définition d'autorisations via les permissions. Il n'est pas possible de spécifier explicitement des interdictions avec le modèle standard de contrôle d'accès de Java. Afin de s'assurer qu'aucune méthode ne peut accéder à (ou modifier) une ressource, l'administrateur doit donc vérifier qu'aucune autorisation d'accès n'a été attribuée, et ce pour toutes les méthodes susceptibles de s'exécuter sur le système. Cette tâche peut se réveler d'autant plus fastidieuse que la politique peut être spécifiée en différents endroits (différents fichiers de configuration, permissions statiques codées « en dur » dans le code Java, etc.). Ce choix a été retenu afin de faciliter et d'optimiser l'implémentation de l'algorithme de contrôle d'accès (celui-ci s'arrête dès la première autorisation trouvée). Un tel choix est fréquent pour les mécanismes de contrôle d'accès utilisant des *ACL* ou des droits attachés aux objets. Pour les mécanismes qui, comme pour Java, utilisent un modèle de capacités associées aux sujets, l'absence d'interdiction peut en pratique être problématique pour limiter l'accès à certains objets sensibles. Cela nécessite une analyse de la politique de contrôle (pour tous les sujets, c'est-à-dire ici pour toutes les classes), qui est disséminée et qui repose sur des permissions dont la sémantique n'est pas toujours évidente à appréhender pour un humain (il peut être nécessaire de comprendre le code de la méthode `implies` de la permission). Il est toutefois possible d'implémenter de telles interdictions en modifiant le gestionnaire de sécurité, mais cette solution s'écarte du modèle standard.

Il est difficile pour un humain (un administrateur par exemple), d'apréhender la sémantique du contrôle effectué. La sémantique du contrôle d'accès s'appuie essentiellement sur l'implémentation des différentes permissions. Ce choix offre une relative souplesse quant à la spécification des différents types de contrôle à réaliser. Il tend cependant à multiplier les types de permissions, notamment pour les applications qui définissent leurs propres permissions (les permissions définies dans la bibliothèque standard étant limitées dès lors qu'il s'agit de contrôler des ressources propres aux applications). Cette profusion des types de permissions ne facilite pas la tâche de l'administrateur chargé de spécifier la politique. De plus, les outils disponibles de gestion de la politique sont relativement limités. Cela s'explique en partie du fait que chaque type de permission peut définir sa propre sémantique de contrôle d'accès. L'ensemble des permissions n'est donc pas homogène. Les seuls champs qu'elles partagent sont le type et le nom. Les actions, par exemple, ne sont pas implémentées par tous les types de permissions et leurs sémantiques peuvent varier d'un type à une autre. Or, ce type d'outil se révèle indispensable dès lors que l'ensemble des autorisations devient relativement important. Cette problématique existe pour d'autres systèmes de contrôle d'accès. C'est notamment le cas de SELinux pour les OS, dont l'essor (relatif) s'explique en partie en raison de l'amélioration des outils de gestion et de spécification de la politique. Toutefois, le problème peut s'avérer plus complexe en Java où la compréhension de la sémantique d'une permission peut nécessiter de lire le code de la classe implémentant cette permission.

Il n'existe pas de réelle hiérarchie entre les permissions. Le mécanisme n'utilise pas de représentation explicite et hiérarchisée des ressources (les objets, au sens du contrôle d'accès) sur lesquels s'appliquent les permissions. Cette représentation des ressources est laissée à la discrétion des différents types de permissions. Celles-ci définissent de manière *ad hoc* les ressources sur lesquelles elles doivent s'appliquer (pour certaines, la ressource est implicite, pour d'autres le type de ressource peut être spécifié, ou un identifiant d'une ressource particulière, etc.). Ceci implique notamment une absence de hiérarchie effective entre les permissions. En théorie, l'héritage entre permissions permet de définir une telle hiérarchie. En pratique, il est difficile et souvent impossible de définir cette hiérarchie de manière à ce qu'elle soit utilisée lors de la mise en œuvre du contrôle. Les classes `Permission` qui n'héritent pas directement de `java.security.Permission` héritent généralement de classes abstraites ou de classes « ombrelles » qui ne sont pas utilisables directement, mais qui permettent seulement de factoriser certaines méthodes.

Le mécanisme n'est pas activé par défaut pour l'exécution d'applications Java « locales ». Il doit être activé à l'aide d'une option passée en paramètre à la commande `java` (ou dans un fichier de configuration). Cela suppose donc de protéger (à l'aide d'un mécanisme de sécurité fourni par l'OS) l'accès au(x) fichier(s) de configuration et aux paramètres de la commande `java`.

De manière générale, le niveau d'implémentation privilégie la souplesse à la sécurité. En effet, le contrôle d'accès est implémenté principalement dans la bibliothèque standard et non

dans la JVM. Ce choix induit trois risques principaux :

- un développeur d'applications ou de bibliothèques Java peut, en modifiant le mécanisme de contrôle d'accès, introduire accidentellement des faiblesses dans le mécanisme. Ce risque dépend du niveau des modifications apportées. Ce risque est renforcé par la relative complexité de l'architecture et des interactions entre les différents éléments ;
- un attaquant ou une application malveillante peut modifier intentionnellement les mécanismes de contôle d'accès (par exemple, les désactiver). Il est donc nécessaire, pour se prémunir de ce risque, de protéger le mécanisme. Comme indiqué dans les différentes sections de cette étude, cela nécessite à la fois de recourir à des mécanismes fournis par l'OS (par exemple, pour protéger les fichiers de configuration) et au mécanisme de contrôle d'accès Java lui-même (par exemple, pour interdir la modification du gestionnaire de sécurité). La mise en place de ces protections peut s'avérer fastidieuse (notamment du fait de l'absence de notion d'interdictions) et nécessite que l'administrateur maîtrise le système de contrôle d'accès de Java ;
- la mise en œuvre de la politique repose sur l'insertion de points de contrôle dans les méthodes permettant d'accéder aux objets. L'efficacité du mécanisme repose donc sur l'hypothèse que toutes les méthodes permettant ces accès ont été identifiées et qu'elles implémentent les points de contrôle de manière correcte. Cette exhaustivité des points de contrôle peut en pratique être vérifiée pour les méthodes de la bibliothèque standard, bien qu'à notre connaissance aucun travail n'ait été réalisé dans le domaine Java. Une telle vérification sort du cadre de cette étude en raison de la complexité et des ressources à mettre en œuvre, il serait toutefois intéressant de s'inspirer des travaux existants dans le domaine du contrôle d'accès pour les OS (notamment ceux concernant la cohérence des *hooks* des LSM [5, 32]). En revanche, il existe un risque qu'une application ou une bibliothèque tierce implémente un accès aux objets protégés par le contrôle d'accès de Java. Ainsi, un développeur peut être amené à implémenter, par exemple pour des raisons d'optimisation, une bibliothèque utilisant des appels à des fonctions natives (grâce à l'interface JNI) pour accéder aux ressources de la plate-forme native (par exemple des fichiers, des IPC, des pages mémoires, etc.). Il n'est donc pas possible, pour un environnement ouvert où différentes applications sont susceptibles d'être installées, d'assurer la complétude des points de contrôle (ou tout du moins cela nécessiterait d'effectuer la vérification pour tout nouveau composant installé). Ce risque amène donc un certain nombre de recommandations à destination des développeurs d'applications Java qui sont détaillées dans le rapport [13]. De manière générale, l'accès aux objets de la plate-forme native doit être réalisé de préférence en utilisant les méthodes Java fournies par la bibliothèque standard. Pour les objets non couverts par la bibliothèque standard et qui nécessitent de développer une méthode native *ad hoc*, il est impératif d'implémenter correctement des points de contrôle. De plus, le risque qu'un attaquant introduise un composant logiciel permettant de cette manière de contourner le contrôle d'accès est certes limité mais non nul.

Le système est fortement dépendant d'une forme de contrôle d'accès (orienté code). L'implémentation par défaut du contrôle d'accès Java prend en compte toutes les méthodes de

la pile d'exécution du *thread* courant (sauf pour le code privilégié [33]), ce qui se révèle assez couteux. Historiquement, le contrôle d'accès de Java avait pour but de se prémunir des applications malicieuses en les confinant. Ce type de contrôle d'accès est donc essentiellement basé sur l'origine du code et nécessite effectivement de prendre en compte les différentes méthodes exécutées. En revanche, d'autres types de contrôle d'accès s'intéressent uniquement à l'utilisateur final et à la ressource ou à l'objet accédé. Ce type de contrôle d'accès est traité en partie par l'API JAAS qui a été ajoutée par la suite, mais qui utilise le mécanisme initial (il s'agit d'une sur-couche). Dans certains cas, l'algorithme de parcours de la pile d'appels n'est donc pas nécessaire, mais il est tout de même appliqué puisqu'il est « codé en dur » dans la classe `AccessControlContext`. Il est alors nécessaire pour implémenter efficacement certaines formes de contôle d'accès de ré-implémenter cette classe (ou de définir un gestionnaire de sécurité *ad hoc*), ce qui est loin d'être trivial.

Le système n'offre qu'une forme limitée de contrôle d'accès. Le système se veut obligatoire (système MAC) dans le sens où la politique appliquée découle uniquement de règles spécifiées par un administateur. Toutefois, ce mécanisme ne repose pas sur l'implémentation d'un moniteur de référence au sein d'un environnement protégé (par exemple, la JVM). Il est donc nécessaire qu'il « s'auto-protège ». De plus, le système ne permet pas de distinguer les différentes instances d'une même classe. Ces capacités de cloisonnement sont donc limitées. Il paraît intéressant d'envisager l'implémentation d'un mécanisme de contrôle d'accès et de cloisonnement intégré à la JVM et qui permettent notamment de distinguer les différentes instances de classes (en associant par exemple des éléments de sécurité à chaque objet).

6.2.3 Java Authentication and Authorization Service

Le mécanisme de contrôle d'accès initialement implémenté dans Java a été développé dans le but de restreindre les accès des applications Java en fonction de l'origine du code. Ce choix historique s'explique en partie par le marché initialement visé par la technologie Java. En effet, il s'agissait essentiellement de fournir un environnement d'exécution standard capable de supporter du code mobile, notamment via la notion d'*applet*. Dans ce contexte, il était primordial de contrôler les accès aux ressources natives du système (en particulier celles gérées par l'OS comme les fichiers) et aux méthodes sensibles (par exemple, les méthodes permettant la programmation réflexive). Comme cela est décrit dans la section 6.2.2, ce contrôle vise à définir différents niveaux de confinement en fonction de la confiance que l'on accorde au code exécuté.

Ces dernières années, l'usage de Java a été étendu à d'autes domaines. Cette technologie est notamment utilisée pour le développement d'applications serveurs [34] ou fonctionnant dans un

33. Le code privilégié correspond à celui explicitement désigné comme tel (utilisation des `PrivilegedAction`). Il existe également certains contrôles, notamment pour les méthodes d'introspection, qui prennent en compte uniquement la classe de la méthode appelante directe. Ces méthodes doivent donc être utilisées avec prudence (voir recommandations du rapport [13]).

34. Ce type d'application est principalement adressé par la version J2EE.

contexte multi-utilisateur (par exemple, une application de type kiosque internet). Les besoins de contrôle d'accès des applications Java ont donc évolué. En particulier, il est apparu nécessaire d'intégrer un mécanisme de contrôle d'accès orienté utilisateur qui permette de restreindre les accès en fonction de l'utilisateur final pour lequel les méthodes sont exécutées. Ce type de contrôle d'accès est traditionnellement implémenté au niveau de l'OS. L'API JAAS (*Java Authentication and Authorization Service*) fournit un *framework* permettant de spécifier et de mettre en œuvre ce type de contrôle d'accès. L'API a été introduite dans la version 1.3 de Java sous la forme d'une extension. Depuis la version 1.4 de Java, elle est intégrée à la bibliothèque standard. JAAS complète les mécanismes existants définis par l'architecture JPSA. Toutefois, les applications doivent être modifiées explicitement pour prendre en compte le contrôle d'accès fourni par JAAS. Il s'agit donc essentiellement d'un service fourni aux développeurs d'applications à la différence du contrôle d'accès « historique » de Java qui est supporté nativement par la plate-forme d'exécution.

JAAS fournit essentiellement deux types de services :

- l'authentification des utilisateurs ;
- la spécification et la mise en œuvre de règles de contrôle d'accès en fonction des utilisateurs authentifiés.

L'API d'authentification de JAAS se veut extensible et compatible avec le système de contrôle d'accès initialement implémenté dans Java. Elle repose sur un système de *plugin*. Le système s'inspire de l'architecture de PAM utilisé sur les systèmes de type UNIX et peut utiliser différents modules d'authentification, chaque module implémentant un moyen d'authentification particulier (login / mot de passe, carte à puce, Kerberos, etc.). Plusieurs API de la bibliothèque d'exécution permettent de gérer l'authentification à distance. SASL, présenté en section 6.4.3.3, apporte notamment un support générique d'authentification pour les protocoles en mode connecté. Cette API repose elle-même sur une architecture modulaire. L'API GSS-API, présentée en section 6.4.3.2, supporte différents moyens d'authentification à distance (comme Kerberos ou SPKM) qui peuvent être utilisés par les services fournis par SASL. JAAS peut utiliser les modules d'authentification fournis par ces différentes API. Elle implémente également ses propres moyens d'authentification, notamment locaux.

JAAS définit également des extensions du mécanisme natif de contrôle d'accès de Java (JPSA) afin de mettre en œuvre le contrôle d'accès orienté utilisateur. Ces extensions permettent notamment :

1. d'associer des permissions aux sujets ;
2. d'associer dynamiquement les sujets à un ensemble de méthodes, traduisant le fait que ces méthodes s'exécutent pour le compte des sujets.

6.2.3.1 Authentification

Le mécanisme d'authentification de JAAS repose sur la notion de sujet implémentée par la classe `javax.security.auth.Subject`. Cette classe déclarée `final` permet de modéliser la notion de sujet utilisée par les mécanismes de contrôle d'accès. Cette notion regroupe les utilisateurs, groupes d'utilisateurs ou services exécutés pour le compte d'utilisateurs. Traditionnellement, les sujets sont identifiés par leurs noms, ces derniers étant implémentés sous la forme de chaînes de caractères ou d'identifiants numériques. Cependant, certains types de mécanismes peuvent utiliser des clés cryptographiques, des certificats, des jetons, etc. Afin de dissocier la notion de sujet de celle d'identifiant de sujet et d'utiliser potentiellement plusieurs types d'identifiants de sujet au sein d'une même application, l'API JAAS introduit la notion de *principal*. Un *principal*, implémenté par une classe implémentant l'interface `java.security.Principal`, permet de lier un sujet à une identité, c'est-à-dire un identifiant de sujet. Un sujet est donc caractérisé par un ensemble de *principals* qui permettent d'identifier le sujet auprès de différents types de services.

Outre l'identifiant du sujet, certains services nécessitent également d'autres informations supplémentaires. JAAS introduit pour cela la notion de *credential* qui correspondent à des attributs de sécurité liés à un sujet. Ces attributs peuvent être de n'importe quel type de Java [35]. Ils permettent, par exemple, de lier un ticket d'authentification au sujet qui peut ensuite être utilisé par d'autres moyens d'authentification dans le cadre d'une architecture SSO. Des clés ou des certificats cryptographiques peuvent également être associés de la sorte à un sujet pour être par la suite utilisés lors des opérations de chiffrement ou de signature. Ces attributs sont accessibles une fois l'utilisateur authentifié. Plus exactement, la classe `Subject` distingue deux types de *credentials* :

- les *credentials* publics (par exemple, des clés publiques, des informations publiques sur le sujet, etc.) sont accessibles librement par n'importe quel code Java grâce à un appel aux méthodes `getPublicCredentials` de la classe `Subject` ;
- les *credentials* privés (par exemple, des clés privés, des *tokens* d'authentification, etc.) dont l'accès est restreint par le mécanisme de contrôle d'accès, sont accessibles uniquement aux classes possédant la permission `javax.security.auth.PrivateCredentialPermission`.

Comme indiqué précédemment, JAAS repose sur un système de module d'authentification inspiré de PAM illustré par la figure 9. Cette architecture garantit l'indépendance, au niveau des applications qui font appel au service d'authentification, de l'implémentation par rapport aux différents moyens d'authentification. Ceci permet également d'utiliser simplement plusieurs moyens d'authentification (par exemple, un login/mot de passe et un *token* physique d'authentification). Concrètement, une application souhaitant authentifier un utilisateur ou un service doit construire une instance de la classe `javax.security.auth.login.LoginContext`. Cette classe propose différentes méthodes génériques permettant d'initier une authentification quels

35. En toute rigueur, il doit s'agir d'une instance d'une classe, les types primitifs et les tableaux n'étant pas supportés.

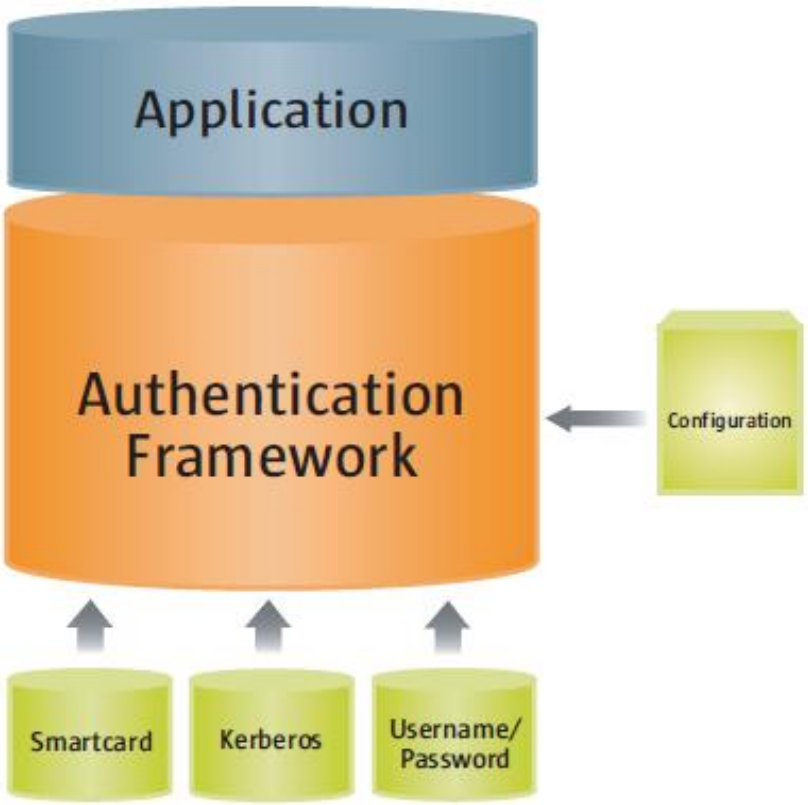


FIGURE 9 – Architecture modulaire d'authentification de JAAS

que soient les services d'authentification utilisés. Elle fait appel à une classe implémentant la classe abstraite `javax.security.auth.login.Configuration` qui permet de spécifier la politique d'authentification. Cette dernière définit quels modules d'authentification doivent être utilisés ainsi que différents paramètres. Chaque module d'authentification utilisé doit implémenter l'interface `javax.security.auth.spi.LoginModule`. Les modules d'authentification utilisés sont configurés par l'administrateur suivant les besoins d'authentification. Plusieurs modules peuvent être empilés et utilisés conjointement pour un même processus d'authentification. La configuration spécifie dans ce cas l'ordre de consultation des modules. Elle spécifie également pour chaque module s'il est optionnel ou obligatoire, l'authentification n'étant validée que si l'ensemble des modules « obligatoires » valide le processus. Par défaut, tous les modules sont consultés (y compris en cas d'échec des modules précédents) et l'authentification n'est validée que si tous les modules valident le processus.

Le processus d'authentification réalisé par l'instance de la classe `LoginContext` est effectué en deux temps :

- chaque module spécifié par la configuration est exécuté et le processus global est validé ou invalidé suivant la réponse de chaque module et la configuration spécifiée ;
- chaque module est de nouveau appelé une fois la première étape réalisée :
  - si le processus d'authentification a échoué, les appels aux modules provoquent une remize à zéro de chaque module,

– si le processus a réussi, les appels permettent à chaque module d'ajouter éventuellement des *credentials* et/ou des *principals* à l'instance de la classe `Subject` créée pour représenter l'utilisateur authentifié.

L'implémentation de Sun de la bibliothèque standard comprend un certain nombre de modules d'authentification permettant d'utiliser le système d'authentification d'UNIX, Kerberos, etc. Ces modules font partie du *package* `com.sun.security.auth.module`. En théorie, les modules ne doivent pas réitérer d'eux-mêmes le processus d'authentification qu'ils gèrent. C'est à l'application de prendre en compte les éventuels échecs d'authentification. Toutefois, c'est au développeur du module d'authentification de respecter ce comportement. Ce point doit donc faire l'objet de recommandations pour les modules d'authentification développés par un fournisseur externe. Les modules peuvent en revanche partager de l'information (par exemple, une même occurence d'un mot de passe entré par l'utilisateur peut être partagé entre différents modules).

Grâce à cette architecture, le code des applications est indépendant des modules d'authentification utilisés. Il est également nécessaire de garantir que le code des modules soit indépendant de l'application. Par exemple, un module d'authentification par *login*/mot de passe doit pouvoir intéragir avec l'utilisateur quel que soit le type d'application (application utilisant une interface graphique, application en ligne de commande, etc.). Pour garantir l'indépendance des modules, l'API permet à l'application de spécifier un mécanisme de *callback* utilisé par la suite par les modules pour s'interfacer avec l'utilisateur. Concrètement, l'application construit une instance d'une classe implémentant l'interface `javax.security.auth.callback.CallbackHandler` et la passe en paramètre à l'objet `LoginContext`. L'implémentation de cette classe spécifie les moyens utilisés pour interagir avec l'utilisateur. Les échanges d'informations entre les modules et l'objet `CallbackHandler` se fait sous la forme d'instance de classes implémentant l'interface `javax.security.auth.callback.Callback`. L'API JAAS fournit des implémentations pour des besoins « standards » (par exemple une demande de *login* et de mot de passe). Les classes concernées se situent dans le *package* `javax.security.auth.callback`. Le développeur de module d'authentification peut définir ses propres implémentations de *callback*. Toutefois, ces *callback* doivent être pris en compte dans le *callback handler* fourni par l'application, ce qui va à l'encontre de la volonté d'indépendance des applications vis-à-vis de l'implémentation des modules d'authentification. Ce point doit donc faire l'objet de recommandations à destination du développeur de modules d'authentification.

6.2.3.2 Contrôle d'accès

La mise en œuvre du contrôle d'accès orienté utilisateur fourni par JAAS suppose un certain nombre de prérequis :

1. l'application doit s'interfacer avec l'API JAAS pour authentifier correctement les sujets ;
2. la politique de contrôle d'accès doit spécifier les permissions associées à chaque *principal* ;

3. l'application doit être modifiée afin qu'elle spécifie explicitement les sujets pour le compte desquels s'exécutent les différentes méthodes.

Le premier point a été décrit dans le paragraphe précédent.

Le second point dépend de l'implémentation de l'objet `Policy`. Comme précisé en section 6.2.2, l'implémentation par défaut de cette classe s'appuie sur des fichiers de configuration permettant à un administrateur de spécifier la politique. Ce mécanisme a été modifié afin de prendre en compte la politique de JAAS. L'administrateur doit donc, en plus des champs relatifs au contrôle d'accès de JPSA, spécifier les règles relatives au contrôle d'accès orienté utilisateurs. Ces règles consistent à associer un ensemble de permissions à chaque *principal*. La spécification de la politique ne repose donc pas directement sur les sujets, mais sur les *principals*, les permissions associées à un sujet correspondant à l'union des permissions associées à chaque *principal* lié à un sujet. Ce choix permet par exemple d'associer plus ou moins de permissions à un sujet en fonction des moyens d'authentification utilisés (et de leur nombre). Par exemple, un utilisateur authentifié via un *token* d'authentification ou une carte à puce pourra se voir octroyer un plus grand nombre de permissions qu'un utilisateur authentifié par un mécanisme réputé plus faible, comme un *login*/mot de passe.

Le troisième point est réalisé grâce aux méthodes `doAs` de la classe `Subject`. Ces méthodes permettent à l'application de spécifier le sujet pour le compte duquel un certain nombre de méthodes vont être exécutées. Plus précisément, ce mécanisme reprend le concept d'actions privilégiées utilisé par le mécanisme de contrôle d'accès de JPSA. L'ensemble des méthodes exécutées par la méthode `run` de l'action privilégiée s'exécute alors pour le compte du sujet précisé en paramètre de la méthode `doAs`. En pratique, le contrôleur d'accès met à jour le contexte d'exécution et associe par la suite l'ensemble des *principals* liés au sujet au domaine de protection de chaque méthode exécutée. Lors de la vérification d'accès, le code effectuant l'accès appelle la méthode `checkPermission` du gestionnaire de sécurité installé sur le système.

Si l'implémentation par défaut de JPSA est utilisée, le gestionnaire de sécurité délègue la vérification des permissions au contrôleur d'accès en appelant la méthode `checkPermission` de la classe `AccessController`. Celle-ci met à jour le contexte d'exécution et détermine l'ensemble des permissions associées à ce contexte suivant l'algorithme présenté en section 6.2.2.3. Lors de l'utilisation de JAAS, le contrôleur d'accès prend également en compte les permissions associées à chaque *principal* lié au contexte d'exécution suite à l'appel de la méthode `doAs`. Le détail de l'algorithme est donné en section 6.2.2.3.

6.2.3.3 Risques

La mise en œuvre du contrôle d'accès suppose implicitement que le gestionnaire de sécurité soit installé. De plus, ce mécanisme dépend fortement du comportement par défaut du contrôle d'accès qui repose sur le contrôleur d'accès implémenté par la classe `java.security.AccessControler`. La modification du mécanisme utilisé par défaut pour la mise en œuvre du contrôle

d'accès peut donc conduire à la désactivation des mécanismes de JAAS, les implémentations des mécanismes par défaut de JAAS et de JPSA étant fortement imbriquées.

Il est important de noter que la configuration de la politique JAAS permet uniquement d'ajouter des permissions en fonction des sujets. Elle ne permet pas de restreindre les permissions déjà octroyées par la configuration de la politique la JPSA qui s'applique aux méthodes en fonction de l'origine du code et ce, quel que soit le sujet pour lequel s'exécute la méthode [36]. L'administrateur doit donc s'assurer que la configuration de la politique JPSA n'accorde pas trop de permissions aux classes dont les méthodes sont susceptibles d'être exécutées par les sujets. La configuration de la politique JAAS peut donc s'avérer relativement complexe et dépend fortement de l'architecture de l'application.

6.2.3.4 Recommandations

Il est recommandé d'utiliser des espaces de noms distincts via l'utilisation de *package* pour le code de l'application effectuant des opérations privilégiées (comme l'authentification et l'appel à la méthode `doAs`) et le code de l'application exécutée pour le compte d'un sujet authentifié. Ce point devra faire l'objet de recommandations à destination des développeurs.

6.2.4 Outils de gestion de la politique de sécurité

La spécification de la politique de contrôle d'accès dépend de l'implémentation de la classe `java.security.Policy` utilisée. L'implémentation par défaut repose sur des fichiers textes de configuration de la politique. Ceux-ci peuvent donc être édités à l'aide de n'importe quel éditeur de texte. Toutefois, certaines implémentations de la plate-forme Java fournissent des outils (graphiques ou en ligne de commande) permettant de spécifier la politique de contrôle d'accès de Java. Ainsi, l'implémentation de Sun fournit les outils suivants :

- `keytool`, un outil en ligne de commande de gestion des certificats des fournisseurs de classes ;
- `policytool`, un outil graphique permettant d'éditer les fichiers de définitions des permissions ;
- `jarsigner`, un outil en ligne de commande permettant de signer les archives (le mécanisme de signatures est présenté en section 6.2.5).

L'outil `keytool` permet de générer des certificats et de les importer dans un conteneur de certificats (*keystore*). L'outil `policytool` est le seul de ces outils à possèder une interface graphique qui reste malgré tout très rudimentaire. Cet outil offre au final peu de fonctionnalités par rapport à un éditeur de texte (la seule fonctionnalité réellement intéressante est la liste déroulante des permissions définies dans la bibliothèque standard).

36. Cela s'explique notamment par l'absence de notion d'interdiction.

6.2.5 Signature de Jar

Le mécanisme de contrôle d'accès de Java présenté en section 6.2.2 nécessite de définir l'origine du code exécuté. Les URL permettent d'identifier facilement l'origine des classes, qu'elle soit locale (la classe provient du système de fichier de l'OS sur lequel s'exécute la plate-forme Java) ou distante (la classe a été téléchargée depuis le réseau). Ce mécanisme n'offre en revanche aucune garantie sur cette origine, car elle ne s'appuie sur aucun élément prouvable. Dans le cadre d'un environnement où les exigences de sécurité sont fortes, il est nécessaire de recourir à un mécanisme d'authentification. Ce besoin est d'autant plus important que la plate-forme d'exécution est ouverte et supporte l'inclusion de code distant.

Le mécanisme de signatures d'archives JAR permet de répondre à ce besoin d'authentification du code. Ce mécanisme a été intégré dès la version 1.1 du JDK. Il a notamment permis d'assouplir la politique de bac-à-sable (*sandbox*) en permettant d'authentifier les *applets* provenant d'un tiers de confiance et de leur appliquer les mêmes règles de confinement que les applications « locales ». La signature s'appuie à la fois sur des algorithmes de hachage et de signatures asymétriques. Il suppose que chaque fournisseur d'application supportant ce mécanisme dispose d'un couple de clés publiques et privées et que la clé publique soit distribuée à l'utilisateur final de l'application sous la forme d'un certificat. Celui-ci peut faire confiance directement à ce certificat (par exemple, s'il lui a été remis directement par un canal réputé « sûr »). La confiance peut également être établie indirectement par une infrastructure de gestion de clés publiques (PKI). Le but du mécanisme de signature est :

- d'une part de prouver l'origine du code (en prouvant l'identité de celui qui a effectué la signature, c'est-à-dire le fournisseur de la classe) ;
- d'autre part de prouver l'intégrité du code (en prouvant que celui-ci n'a pas été modifié entre la signature par le fournisseur et la vérification par la plate-forme de l'utilisateur).

Le mécanisme de signature de Java s'appuie sur les archives JAR. Celui-ci permet de regrouper différentes classes au sein d'un même conteneur qui est compressé. Il s'agit en fait d'un format dérivé du format d'archive compressé ZIP (il est d'ailleurs tout à fait possible de gérer les archives JAR à l'aide d'un outil de compression/décompression compatible avec le format ZIP, pourvu que celui-ci respecte la casse des noms de fichiers et de répertoires). Cette archive peut également comprendre, en plus des différents fichiers de classe, des méta-données relatives aux classes ou à l'archive. Ce format permet de distribuer facilement des applications, car il peut être exécuté directement à l'aide de la commande `java`. Le mécanisme de signature ne s'applique pas directement à l'archive JAR, mais à certains des fichiers de méta-données inclus dans l'archive. Une archive JAR peut être signée par différents fournisseurs. Le mécanisme repose sur trois étapes détaillées par la suite.

La première étape consiste à garantir l'intégrité de chacun des fichiers `.class` de l'archive en calculant un haché (encodé en base 64) de chaque fichier `.class` concerné par la signature. Ce haché est ensuite inclus dans le fichier de méta-données `MANIFEST.MF` situé dans le répertoire `META-INF` de l'archive JAR. Ce fichier texte contient différentes sections séparées par des

lignes vides, chaque section permettant de préciser un certain nombre d'attributs pour chaque fichier. Il est utilisé par d'autres mécanismes que la signature (il permet par exemple d'identifier la classe principale de l'application, qui contient la méthode `main` qui permet de démarrer l'application). En ce qui concerne le mécanisme de signature, chaque fichier `.class` concerné par la signature doit faire l'objet d'une entrée dans le fichier `MANIFEST.MF`. Cette entrée doit comprendre au moins deux attributs :

- le nom de la classe sous la forme `Name: Class_Name` avec `Class_Name` le nom de la classe exprimée sous la forme d'une URL ou d'un chemin relatif, par exemple `org/entrepriseA/MaClasse.class` ;
- le haché du fichier classe précédé du nom de l'algorithme utilisé pour calculer ce haché, par exemple `SHA1-Digest: jhfctfu7n5bvFGfy/gd8=`

Chaque entrée peut comprendre plusieurs hachés correspondant à l'utilisation de différents algorithmes de hachage. Elle peut également comprendre un attribut optionnel appelé *Magic* qui permet de donner des renseignements sous forme de mot-clés au mécanisme de vérification de signatures. Ces mots-clés sont spécifiques à l'application et aucun format n'est imposé. Par exemple, le champ `Magic: Multilingual` indique que le fichier offre un support multilingue et qu'il existe différents hachés suivant la langue.

La seconde étape consiste à garantir l'intégrité de chacune des entrées du fichier `MANIFEST.MF`. Ce premier niveau d'indirection permet de garantir à la fois l'intégrité du code et des métadonnées qui lui sont associées. Il permet également de supporter différents signataires. Pour cela, un fichier de signatures (*signature instructions file*) est créé pour chaque signataire. Ce fichier est placé dans le répertoire `META-INF` de l'archive et possède un suffixe `.SF`. La structure de ce fichier texte est similaire à celle du fichier `MANIFEST.MF`. Il comprend un en-tête (section principale) et différentes sections correspondant aux sections du fichier `MANIFEST.MF`. Les informations contenues dans la section principale correspondent au signataire. Elle peut également comprendre un attribut *algo*`-Digest-Manifest-Main-Attributes` (avec *algo* le nom de l'algorithme utilisé) correspondant au haché des attributs globaux du fichier `MANIFEST.MF`. Chaque section suivante comprend le nom d'une section présente dans le fichier `MANIFEST.MF` ainsi qu'un attribut correspondant au calcul d'un haché de l'entrée correspondante du fichier `MANIFEST.MF`. Les entrées du fichier `MANIFEST.MF` qui ne sont pas présentes dans le fichier de signatures ne sont pas prises en compte par le mécanisme de vérification des signatures pour ce signataire. Un fournisseur peut donc signer un sous-ensemble des fichiers d'une archive JAR.

La troisième étape constitue l'étape de signature à proprement parler. Elle se matérialise par la création d'une signature du fichier de signatures, elle même contenue dans un fichier binaire (*block signature file*) dont le format et l'extension dépendent de l'algorithme de signature utilisé. Ce fichier comprend, outre la signature réalisée avec la clé privée du signataire, le certificat contenant la clé publique du signataire. Les algorithmes de signature les plus utilisés sont :

- signature PKCS7 utilisant RSA avec la primitive de hachage SHA1 (extension `.RSA`). L'outil de signature d'archives JAR choisit par défaut SHA1 comme algorithme de hachage pour une signature RSA. Il est également possible d'utiliser MD5, MD2, SHA512, SHA384, SHA256 ;
- signature PKCS7 utilisant DSA avec la primitive de hachage SHA1 (extension `.DSA`).

D'autres formats peuvent être utilisés. Les fichiers sont alors préfixés par `SIG-`. Plusieurs fichiers *block signature* peuvent être utilisés pour un même fichier de signatures (*signature instructions file*) correspondant à l'utilisation de différents algorithmes de signatures. Ils doivent en revanche correspondre au même signataire.

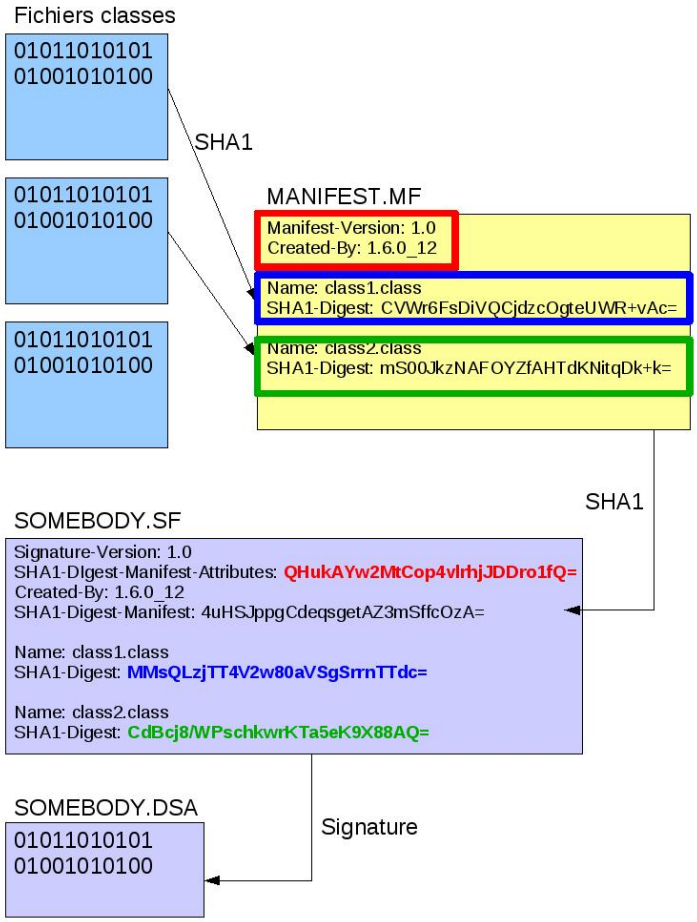


FIGURE 10 – Signature d'une archive JAR

Lors du chargement des classes contenues dans l'archive, la vérification de la signature a lieu en plusieurs étapes :

1. Lors de l'ouverture de l'archive et de la consultation du fichier `MANIFEST.MF`, les signatures des différents fichiers de signatures sont vérifiés à l'aide des clés publiques des signataires. Si ces clés correspondent à des certificats de tiers de confiance, le mécanisme peut valider l'identité des signataires.
2. Ensuite, l'intégrité des entrées concernées dans le fichier `MANIFEST.MF` sont vérifiées à l'aide des hachés présents dans chaque fichier de signatures.
3. Enfin, lors du chargement d'un fichier de classe, le mécanisme calcule son haché et le compare à la valeur stockée dans l'entrée du fichier `MANIFEST.MF`. À ce stade, les

différentes signatures du fichier sont validées. Le chargeur de classes peut notamment associer l'ensemble des certificats des signataires de la classe au domaine de sécurité associé à la classe chargée. L'identité des signataires de la classe peut alors être prise en compte par le contrôle d'accès.

Remarque 3 *La vérification des signatures d'une archive n'est faite qu'une seule fois au moment du chargement de la première classe de l'archive. La vérification de l'intégrité des fichiers* `.class` *est faite au moment de leur chargement en mémoire.*

Le mécanisme de vérification des signatures impose également une règle de cohérence sur les signatures des classes d'un même *package* Java. Cette règle impose que toutes les classes d'un même *package* soient signées de la même manière (signataires identiques). Cette règle vise à se prémunir des risques d'attaques de type *package insertion*. Cette attaque consiste à leurrer le système de chargement de classes et à insérer une classe dans un *package* donné. Ceci permet potentiellement de contourner le contrôle d'accès si ce dernier s'appuie exclusivement sur les URL. Ce type d'attaque montre la faiblesse de l'identification des classes par l'URL. D'un point de vue sécurité, le mécanisme de signature devrait être privilégié, mais ce dernier génère bien évidemment un surcoût à l'exécution lié à la vérification des signatures.

Remarque 4 *Au sein d'un même* package *(même répertoire), toutes les classes doivent avoir été signées par les mêmes signataires. En effet, au chargement de la première classe d'un* package*, une liste des signataires est enregistrée. Si au moment de charger une deuxième classe de ce même* package*, la liste des signataires diffère, alors une* `SecurityException` *est levée :* class "B"'s signer information does not match signer information of other classes in the same package.

Remarque 5 *Au sein d'un même* package *(même répertoire), des signataires différents peuvent utiliser des algorithmes de signature différents. Il en va de même pour les algorithmes de hachage utilisés pour les motifs d'intégrité.*

Remarque 6 *Dans le cas où une archive est signée par plusieurs signataires, le fait qu'une signature soit invalide (perte d'intégrité sur le fichier de signatures (*.SF) ou sur la signature même) conduit à l'invalidation totale de l'archive. Le contrôle des signatures se fait au moment où une première classe de l'archive est utilisée.*

Remarque 7 *Les fichiers de signatures contiennent un haché du fichier* `MANIFEST.MF`*. Ce haché n'est pas vérifié à l'exécution de l'application Java. En effet, il doit toujours être possible d'ajouter de nouvelles classes à l'archive et de réaliser une signature avec un signataire différent sur l'ensemble de l'archive. Cela implique une modification du fichier* `MANIFEST.MF`*, car de nouveaux hachés ont été ajoutés pour les nouvelles classes. L'utilitaire* jarsigner *détecte ce genre de modification.*

Remarque 8 *Dans le cas où une archive JAR fait l'objet de plusieurs signatures, il est nécessaire de refaire les signatures précédentes. En effet, les entrées des fichiers concernés par les entrées des fichiers situées dans le* `MANIFEST.MF` *peuvent avoir évoluées (cela est le cas si un nouvel algorithme de hachage est utilisé). Par conséquent, il est nécessaire de recalculer les hachés des entrées du fichier* `MANIFEST.MF` *du fichier signature et de refaire la signature du fichier de signatures.*

Remarque 9 *La signature d'une archive JAR se fait par le biais de l'utilitaire de Sun* jarsigner. *Cet outil permet de signer et de vérifier une archive JAR. La signature est réalisée sur l'ensemble de l'archive. L'outil ne permet pas de réaliser une signature sélective sur un nombre restreint de fichiers.*

6.3 Chargement de classes

Le mécanisme de chargement des classes Java est un mécanisme propre aux environnements d'exécution reposant sur une représentation intermédiaire de type *bytecode*. Il consiste à identifier les fichiers de classes nécessaires à l'application, à les lire et à les charger au sein de la mémoire de la JVM, puis à générer à partir de leurs contenus une représentation des classes en mémoire. Ce mécanisme fonctionne de concert avec d'autres mécanismes de la JVM comme le vérificateur de *bytecode* (qui doit vérifier chaque classe chargée en mémoire) et le mécanisme de gestion de la mémoire. Le format des fichiers de classes est défini par la spécification de la JVM [49]. En revanche, cette dernière offre beaucoup de libertés sur la manière de distribuer les fichiers classes. Plusieurs cas de figure sont ainsi possibles, dont entre autres :

- stockage et chargement des fichiers `.class` depuis la mémoire de masse de la plate-forme d'exécution ;
- téléchargement depuis le réseau (c'est notamment le cas des *applets*) ;
- génération « à la volée » (instrumentation de *bytecode*, programmation orientée aspects).

Pour autant, la majorité des opérations réalisées au cours du chargement sont identiques. Ce mécanisme est donc implémenté par deux éléments distincts de la plate-forme d'exécution Java :

- les chargeurs de classes (*class loaders*), généralement implémentés en Java, qui assurent les opérations « amont » du chargement. Ils permettent notamment d'adapter le mécanisme suivant les différents moyens de distribution mis en œuvre. Ils sont décrits plus en détail dans ce document par la suite ;
- l'implémentation de la JVM, qui assure les opérations communes aux différents types de classes. Elle est décrite plus en détail dans les documents [14] et [11].

Plus précisément, la spécification de la JVM distingue les étapes suivantes :

1. le chargement de classes à proprement parler, qui consiste, à partir du nom développé (*fully qualified name*) d'une classe, à identifier ou générer le fichier `.class` associé et à fournir un tableau d'octets représentant son contenu ;

2. la phase d'édition de liens qui regroupe différentes opérations qui permettent d'aboutir, à partir du tableau d'octets issu de l'étape précédente, à une représentation de chaque classe au sein de la mémoire de la JVM. Cette phase comprend elle-même différentes sous-phases, qui sont réalisées de manière plus ou moins « paresseuse » [37] :
   - la vérification, qui consiste à s'assurer que le contenu du fichier `.class` respecte le format défini dans la spécification et que le *bytecode* des méthodes des classes chargées possède certaines propriétés de sécurité (respect du typage, par exemple) évoquées en section 5. Cette phase est réalisée par le vérificateur de *bytecode* [38],
   - la préparation, qui consiste à allouer les structures nécessaires à la représentation, au sein de la mémoire de la JVM, de la classe et de ses instances. Cette étape réalise également une mise à zéro (valeur par défaut) des différents champs de la classe. Ces structures sont en partie accessibles depuis l'environnement Java par une instance de la classe `java.lang.Class` représentant la classe chargée,
   - la résolution, qui consiste à vérifier et traduire les références symboliques qui désignent, au sein de la classe chargée, les autres classes ou leurs éléments (champs et méthodes), par des références directes (dans la pratique, il s'agit généralement de pointeurs) ;
3. la phase d'initialisation, qui consiste à exécuter le code Java d'initialisation de la classe (initialisateur statique [39]).

Parmi ces différentes étapes, seule la première n'est pas réalisée exclusivement par l'implémentation de la JVM, mais par un composant indépendant présenté plus en détail par la suite : le chargeur de classes (ou *class loader*).

6.3.1 Chargeur de classes

Le chargeur de classes est généralement implémenté en Java sous la forme d'une classe héritant de la classe abstraite `java.lang.ClassLoader`. Ce composant permet d'assurer la propriété de chargement dynamique en Java : le type de certaines classes Java peut n'être déterminé qu'à l'exécution. En effet, le chargement peut être initié de deux manières :

- explicitement, par le code de l'application ;
- implicitement, par la JVM (par exemple, à l'issue de la phase de résolution).

Dans tous les cas de figure, le chargement est initié par un appel à la méthode `loadClass` du chargeur de classes. Le nom développé de la classe est passé en paramètre à cette méthode. À

37. Notamment pour la résolution qui, en pratique, est réalisée lors de la première utilisation effective de l'élément référencé.

38. En théorie, toutes les classes doivent être vérifiées, mais en pratique certaines implémentations de la JVM n'effectuent pas les vérifications pour les classes système chargées par le *bootstrap class loader*, comme indiqué dans le rapport [11].

39. L'initialisateur statique regroupe l'ensemble des blocs `static{...}` de la classe et l'ensemble des champs `static` qui sont initialisés dès leur déclaration.

partir de ce nom, le chargeur de classes doit identifier et lire le fichier de classe correspondant ou générer le code correspondant « à la volée ». Chaque implémentation du chargeur de classes peut mettre en œuvre une stratégie qui lui est propre en ce qui concerne cette phase du chargement (recherche d'un fichier local, requête sur le réseau, etc.). Certains chargeurs peuvent ainsi pré-charger les classes les plus couramment utilisées lors de certaines phases (par exemple, lors du démarrage de la JVM). Le code binaire correspondant à la classe chargée est ensuite fourni à l'implémentation de la JVM à l'aide d'un appel à la méthode finale `defineClass` de la classe `java.lang.ClassLoader`. Cette méthode s'appuie sur du code natif afin de dialoguer avec l'interface propre à chaque implémentation de la JVM. Elle fournit en retour une instance de la classe `java.lang.Class` représentant la classe chargée qui peut alors être utilisée [40].

La plupart des chargeurs de classes sont développés en Java, mais il est nécessaire d'initier le processus de chargement. Les classes de base de la bibliothèque standard (notamment celles du *package* `java.lang.*`, dont `java.lang.Class` et `java.lang.ClassLoader`) sont donc chargées par le chargeur de classes initial (*bootstrap class loader*), qui est développé en C ou C++ et qui est intégré à l'implémentation de la JVM. Toutefois, il s'agit d'un chargeur simplifié qui ne peut charger que les classes système (généralement contenues dans une archive, par exemple `rt.jar`) et qui n'implémente pas les mécanismes complexes de vérification de signature, de recherche suivant une URL, etc.

Le chargeur de classes assure également plusieurs propriétés de sécurité :

- il assure en partie l'intégrité et la disponibilité du code des applications (sous forme de *bytecode*), notamment lors du transfert des fichiers `.class` d'un support (mémoire de masse, réseau, etc.) vers la mémoire de la JVM ;
- il participe au cloisonnement de certaines classes au sein d'une même instance de la JVM ;
- il participe au contrôle d'accès, notamment en définissant les domaines de protection associés à chaque classe.

Le premier point repose essentiellement sur la confiance dans le code du chargeur de classes utilisé. Les autres points sont décrits plus en détail dans les sections suivantes.

6.3.2 Délégation

Les chargeurs de classes peuvent déléguer en partie le processus de chargement d'une classe. Cette opération de délégation définit une hiérarchie parmi les chargeurs. De fait, ceux-ci sont organisés suivant une double hiérarchie :

- une hiérarchie « statique » définie par les relations de typage. Tous les chargeurs de classes héritent de `java.lang.ClassLoader` (sauf le *bootstrap class loader*). Dans la

40. En toute rigueur, il est également nécessaire d'initier la phase d'édition de liens avant de pouvoir utiliser la classe. Cette étape est généralement réalisée automatiquement, mais doit être initiée explicitement par un appel à la méthode `resolveClass` lors d'un chargement explicite.

pratique, les chargeurs de classes définis par le développeur d'une application héritent souvent d'un sous-type de cette classe abstraite (par exemple, `URLClassLoader` ou `SecureClassLoader`). Cette hiérarchie porte sur la sémantique et les fonctionnalités des chargeurs de classes (un chargeur qui hérite de `URLClassLoader` fournit au moins tous les services de cette classe) ;

– une hiérarchie « dynamique » fondée sur la délégation. Cette hiérarchie est construite dynamiquement : lors de la construction d'un chargeur de classes, il est nécessaire de fournir en paramètre au constructeur une référence sur le chargeur de classes « père », auquel les requêtes de chargement seront déléguées. Lorsque le constructeur par défaut est utilisé, le chargeur de classes père est le chargeur de classes « système » (celui utilisé par défaut sur le système).

Typiquement, la hiérarchie dynamique comprend les chargeurs de classes suivants :

– le chargeur de classes initial (*bootstrap class loader*), écrit en C/C++ et intégré à la JVM, assure le chargement des classes de base de la bibliothèque standard ;

– le chargeur de classes des extensions (*extension class loader*), écrit en Java et fourni avec l'implémentation de la bibliothèque standard, définit, comme son nom l'indique, les classes contenues dans les archives intégrées à la bibliothèque standard via le mécanisme d'extension ;

– le chargeur de classes système (*system class loader*), écrit en Java et fourni avec l'implémentation de la bibliothèque standard, cherche les classes dans l'ensemble des répertoires et archives définis dans la variable d'environnement `CLASSPATH`. Il s'agit en fait plutôt d'un chargeur applicatif, puisque c'est généralement lui qui charge les classes des applications ;

– les éventuels chargeurs de classes, fournis avec l'application ou les bibliothèques qu'elles utilisent, assurent le chargement et la définition de certaines classes de l'application.

La figure 11 illustre l'organisation hiérarchique dynamique de ces différents chargeurs de classes.

Le mécanisme de délégation permet de dédier le chargement de certaines classes à des chargeurs de classes particuliers. Lors du chargement, la plate-forme Java distingue pour cela deux types de chargeurs de classes :

– le chargeur qui initie le chargement (*initiating class loader* ou *initial class loader*) ;

– le chargeur qui définit la classe (*defining class loader* ou *effective class loader*).

Concrètement, le premier est celui qui est appelé explicitement au sein de l'application Java ou implicitement par la JVM (via un appel à sa méthode `loadClass`). Lors d'un chargement implicite, le chargeur de classes qui initie le chargement est celui qui a défini la classe qui a provoqué le chargement. Cette classe possède une méthode ou un initialisateur statique dont le code comprend, lorsque la méthode est exécutée, une référence symbolique non encore résolue vers la classe qui va être chargée. Ce chargeur peut alors lui-même chercher le fichier `.class` correspondant et définir la classe chargée via un appel à la méthode finale `defineClass`. Il peut cependant choisir de déléguer le chargement à son père. Cet algorithme est itéré jusqu'au

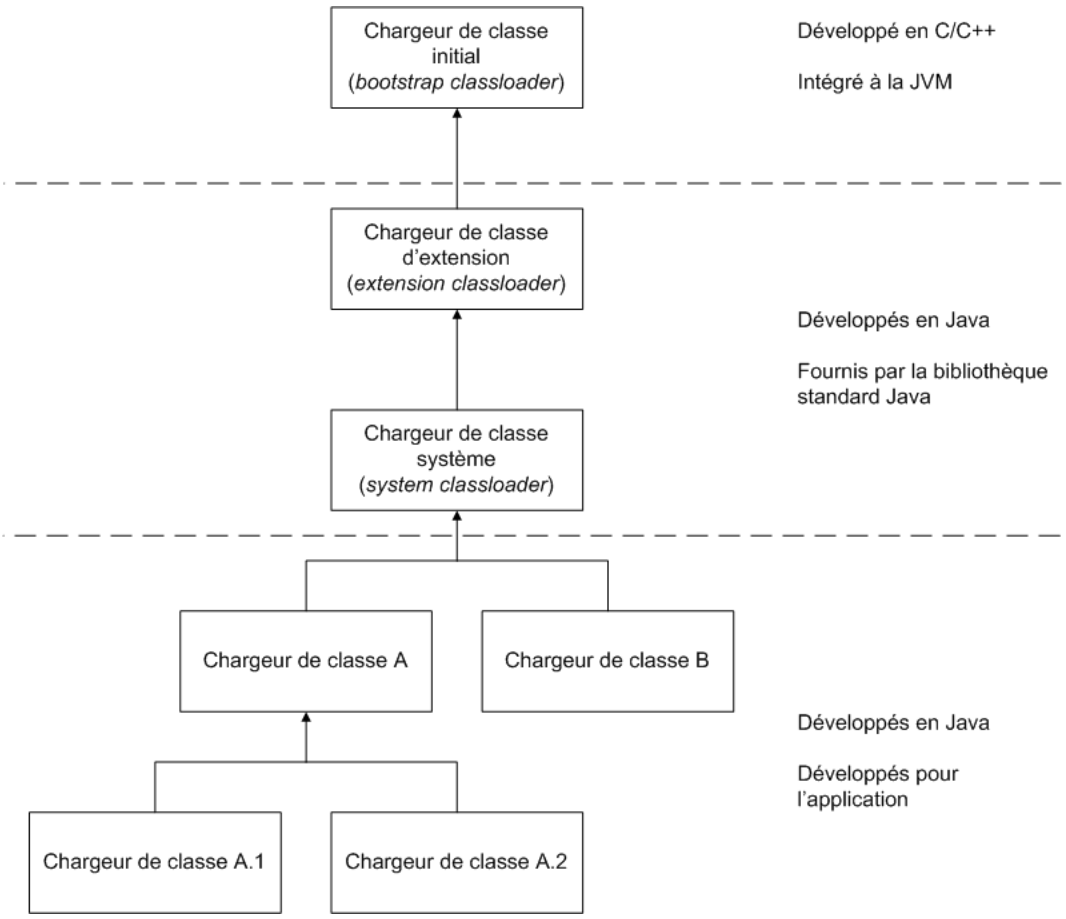


FIGURE 11 – Exemple de hiérarchie de délégation de chargeurs de classes.

chargeur racine (le *bootstrap class loader*). Le chargeur qui définit la classe peut donc être différent de celui qui initie le chargement.

Par défaut, les chargeurs de classes délèguent systématiquement le chargement à leur père avant de tenter eux-mêmes le chargement. Cela équivaut donc à déléguer systématiquement le chargement au *bootstrap class loader*. Chaque chargeur définit un ensemble de classes dont il a la responsabilité et auxquelles il peut accéder. Lorsqu'un chargeur ne peut accéder à une classe, il lève une exception de type `java.lang.ClassNotFoundException` qui est récupérée par le chargeur qui a délégué le processus de chargement (un de ses fils). Ce chargeur tente alors à son tour de trouver la classe. L'algorithme est itéré jusqu'à ce qu'un chargeur trouve et définisse la classe. Si aucun n'a été en mesure de le faire (c'est-à-dire si le chargeur qui a initié le chargement n'a pu trouver la classe), le programme se termine suite à la levée d'une exception `ClassNotFoundException`. Ce comportement est implémenté dans le code de la méthode `loadClass` de la classe `java.lang.ClassLoader`. Plus précisément, le comportement par défaut réalise les opérations suivantes :

1. lors d'un appel à la méthode `loadClass`, le chargeur vérifie qu'il n'a pas déjà chargé la classe. Il réalise cette vérification à l'aide d'un appel à la méthode finale `findLoadedClass`. Cette vérification permet de s'assurer qu'une instance d'une même classe

n'est chargée qu'une seule fois par un chargeur de classes donné. Ceci garantit que le code d'une classe n'évolue pas au cours de l'exécution d'une application [41] ;

2. le chargeur délègue ensuite le chargement à son père si celui-ci est défini (appel à `getParent().loadClass`);
3. si aucun de ses ancêtres n'a pu définir la classe, le chargeur initie la recherche du fichier `.class` via un appel à la méthode `findClass`. Cette dernière implémente l'algorithme de recherche propre au chargeur de classes et définit en quelque sorte les classes qui sont du ressort du chargeur. Deux cas de figure sont alors possibles :
   - si la classe est trouvée, elle est définie via un appel à la méthode finale `defineClass`,
   - si la recherche échoue, une exception de type `ClassNotFoundException` est levée.

Ce comportement par défaut est le comportement attendu d'un chargeur de classes (selon la documentation de la bibliothèque standard Java SE). Il n'est donc pas recommandé, lors de la création d'un nouveau type de chargeur de classes, de modifier le code de la méthode `loadClass`. Il suffit généralement de surcharger la méthode `findClass`. Toutefois, la méthode `loadClass` n'est pas déclarée `final` et il est possible de modifier ce comportement. Par exemple, certains chargeurs de classes de serveurs d'applications Java tentent tout d'abord de trouver la classe et de la définir avant de déléguer éventuellement le chargement. En effet, cela permet précisemment d'assurer une forme de confinement (ce comportement est d'ailleurs recommandé par la spécification des Servlet pour les environnements de type Java EE, qui contredit donc la spécification Java SE). Toutefois, il est nécessaire de déléguer systématiquement le chargement des classes système, qui ne peuvent être définies par un chargeur autre que le *bootstrap class loader* (la plupart des JVM effectue cette vérification). Cette restriction vise à limiter les risques de piégeage par un code tiers malicieux ou lui même piégé.

6.3.3 Cloisonnement

Outre la possibilité de paramétrer la manière de trouver et d'obtenir les fichiers de classe, le mécanisme de chargement de classes permet également d'assurer le cloisonnement entre certaines classes chargées au sein d'une même instance de la JVM. Concrètement, l'utilisation de différents chargeurs de classes au sein d'une même instance de la JVM permet de mettre en place différents espaces de nommage : les instances d'une même classe (identifiée par son nom développé, par exemple `monpackage.MaClasse`), chargées par des chargeurs de classes distincts, possèdent en fait des types différents. Ce mécanisme permet donc de cloisonner certaines parties d'une application Java [42].

41. Cette propriété peut cependant être mise en défaut par les mécanismes d'instrumentation et d'audit introduits dans les dernières versions de Java et qui sont présentés en section 9.

42. Il est notamment utilisé pour les applications de type Java SE où les différentes « applications » Web s'exécutant sur un serveur d'applications Java (donc sur une même instance d'une JVM) sont cloisonnées entre elles de la sorte.

Le cloisonnement mis en place par le système de chargeurs de classes repose sur la mise en place d'un espace de nommage. Le type complet d'une classe correspond en fait au nom développé de la classe augmenté du chargeur de classes qui la définit. Formellement, si N est le nom développé de la classe (par exemple, `java.lang.String`) et L l'instance du chargeur de classes qui a défini cette classe, alors le type complet de la classe est : $T =< N, L >$. Concrètement, différents chargeurs de classes peuvent définir des classes qui possèdent un nom identique, mais qui sont de types différents. Ces classes peuvent être issues de fichiers `.class` différents et être distinctes en termes de code et de structures. Elles peuvent également être identiques et provenir d'un même fichier[43]. Considérons par exemple deux chargeurs de classe L_A et L_B. Ces deux chargeurs peuvent tous les deux définir une classe nommée *MaClasse*. Lors de la phase de résolution, la référence symbolique `MaClasse` est traduite, selon les classes, par une référence de type $< MaClasse, L_A >$ ou $< MaClasse, L_B >$. Si l'on suppose par exemple que L_A et L_B ne sont pas ancêtres (dans la hiérarchie de délégation établie dynamiquement) l'un de l'autre, les références symboliques `MaClasse` contenues dans les classes définies par L_A (ou un de ses descendants) seront traduites par des références directes de type $< MaClasse, L_A >$. Ainsi, les classes définies par L_A (ou un de ses descendants) ne peuvent accéder aux champs et aux méthodes de $< MaClasse, L_B >$. Toute tentative d'assignation d'une référence de type $< MaClasse, L_B >$ dans une variable ou un champ d'une de ces classes donne lieu à la levée d'une exception (par exemple, de type `ClassCastException`).

Si les différents chargeurs de classes respectent le comportement par défaut, le confinement permet à une classe de voir et de manipuler uniquement les classes qui ont été définies par le même chargeur de classes ou par un de ses ancêtres. En revanche, la classe ne peut pas accéder aux classes définies par les chargeurs de classes descendants (ou appartenant à des branches différentes) de son propre chargeur de classes. Tous les chargeurs de classes ont pour ancêtre le chargeur de classes initial (*bootstrap class loader*) qui doit définir les classes de base de la bibliothèque standard (typiquement, toutes les classes du *package* `java.*`). D'après le comportement par défaut, ces classes sont obligatoirement chargées et définies par le *bootstrap class loader*. De plus, en pratique, l'implémentation de la méthode finale `defineClass` s'assure que seul le *bootstrap class loader* puisse définir ces classes de base de la bibliothèque standard. Une tentative de chargement et de définition d'une de ces classes par un autre chargeur de classes se solde par la levée d'une exception. Le confinement est donc partiel :

- les classes appartenant à différents « domaines » de confinement peuvent communiquer à travers les classes chargées et définies par les chargeurs de classe qui sont des ancêtres communs aux chargeurs des différents domaines ;
- les types définis dans les classes de bases étant visibles et manipulables par toutes les classes, celles-ci peuvent être amenées à stocker et modifier des références sur des instances de classes qui appartiennent à des domaines distincts. En effet, il est toujours possible pour ces classes de manipuler ces références via des conteneurs (variables locales, champs d'objets ou de classes) correspondant à un sous-type, notamment `java.lang.Object` dont hérite chaque classe. En revanche, l'utilisation d'une classe (c'est-à-dire l'accès à ses champs et à ses méthodes) suppose une opération de surcharge ascendante (*up-cast*), qui précise le type et qui lève une exception si la classe

43. Toutefois, ceci n'est pas conseillé, car les chargeurs doivent en principe administrer des groupes de classes (par exemple, des répertoires) distincts.

utilisatrice et la classe utilisée appartiennent à des domaines distincts.

6.3.4 Liens avec le contrôle d'accès

Le mécanisme de chargement de classes est également lié au mécanisme de contrôle d'accès, décrit en section 6.2, à différents niveaux :

- certaines opérations implémentées par les chargeurs de classes font l'objet de contrôle ;
- le chargeur de classes participe au processus de contrôle d'accès en associant notamment les domaines de protection à chaque classe.

Le premier niveau permet de limiter l'usage du mécanisme de chargement de classes depuis certaines parties du code de l'application Java. En effet, les méthodes permettant de charger et de définir les classes ou celles permettant de modifier la hiérarchie de délégation implémentent des fonctions critiques. Par exemple, ce type de méthode peut être utilisé par du code malicieux afin de contourner le cloisonnement ou pour piéger l'application en définissant une classe à son insu (et dont le code ne correspond pas à celui attendu par l'application). Les constructeurs de la classe `ClassLoader` ainsi que les méthodes `getSystemClassLoader` et `getParents` implémentent notamment des points de contrôle d'accès. Ainsi, seul le code disposant des permissions adéquates peut exécuter ces méthodes.

Le second niveau témoigne de l'importance du chargeur de classes dans le processus de détection. En effet, il participe à l'« authentification » des classes chargées :

- si le chargeur est amené à définir des classes contenues dans une archive JAR, il doit au préalable s'assurer de l'intégrité de la classe chargée en vérifiant sa ou ses signatures (si la classe est signée) et en mémorisant les certificats de ses fournisseurs. Il peut s'appuyer pour cela sur des classes *ad hoc* de la bibliothèque standard (par exemple `java.util.jar.JarFile` et `java.util.JarEntries` qui utilisent elles-mêmes `java.util.JarVerifier`). Cette opération n'est pas spécifiée dans les méthodes de la classe `java.lang.ClassLoader`. Le développeur du chargeur de classes doit donc les spécifier explicitement ou faire hériter le chargeur qu'il développe d'une classe qui les implémente (par exemple, `java.net.URLClassLoader`) ;
- le chargeur qui définit une classe doit associer un domaine de protection à l'instance de la classe `java.lang.Class` correspondant à la classe chargée. Comme décrit en section 6.2, le domaine de protection regroupe notamment la description de l'origine du code en termes d'URL et de certificats des fournisseurs de la classe. Plusieurs classes peuvent partager le même domaine de protection, le chargeur de classes est donc amené à gérer un cache de domaines de protection. La classe `java.security.SecurityClassLoader` implémente des méthodes qui facilitent cette association. Lors du développement d'un chargeur de classes, il est donc utile de faire hériter la classe implémentant le chargeur de la classe `SecurityClassLoader`. C'est notamment le cas de la classe `URLClassLoader` (dont héritent la plupart des chargeurs), qui définit par défaut un domaine pour chaque classe chargée en fonction de son URL précise et des signataires de la classe ;

– lors de la définition du domaine de sécurité de la classe, le chargeur peut également y associer un ensemble de permissions statiques. Comme indiqué en section 6.2, chaque domaine de protection est associé à deux ensembles de permissions, statique et dynamique, le dernier étant déterminé lors de la vérification à l'aide de la classe implémentant la politique de sécurité (`java.security.Policy`). Le contrôle d'accès considère l'union de ces ensembles. Les permissions statiques sont surtout présentes pour des raisons historiques, les implémentations récentes s'appuyant principalement sur les permissions dynamiques. Toutefois, certains chargeurs de classes définissent des permissions statiques qui complètent celles définies par la politique de sécurité et qui sont nécessaires au bon fonctionnement de certains mécanismes.

6.3.5 Risques

Le chargement de classes Java est un mécanisme offrant une grande souplesse de paramétrage et qui assure des fonctions critiques pour la sécurité des applications Java. Plusieurs motifs peuvent justifier le développement et l'utilisation d'un chargeur de classes spécifique et qui supplante en partie le chargeur système :

– l'adaptation du chargement de classes à un mode de distribution particulier (utilisant par exemple une base de données) ;
– la mise en œuvre d'opérations spécifiques lors du chargement (par exemple, des contrôles additionnels ou de l'instrumentation de *bytecode*) ;
– le cloisonnement de différents blocs fonctionnels de l'application Java s'exécutant sur l'instance de la JVM.

Toutefois, il s'agit d'un mécanisme relativement complexe. Le risque principal consiste à définir différents chargeurs possédant des domaines de définition non-disjoints et qui n'implémentent pas de manière correcte le mécanisme de délégation. Ce type d'erreur conduit à des dysfonctionnements qui provoquent la levée d'exceptions ou l'utilisation de versions inappropriées de certaines classes. Ce risque est accru par le fait que le comportement par défaut (délégation au père avant de tenter de charger la classe) est fortement recommandé, mais qu'il peut être modifié. Il l'est notamment par certaines classes de la bibliothèque standard qui s'appuient sur le *context class loader*[44].

Le deuxième risque majeur consiste à mettre en œuvre un cloisonnement partiel, qui peut être contourné. La complexité du mécanisme renforce la difficulté de mise en place d'un cloisonnement strict des classes. En effet, il est possible d'échanger de l'information avec les types visibles depuis les différents domaines (notamment, les classes définies par le chargeur initial). Les mécanismes de sérialisation ou d'appel de méthodes à distance (RMI) peuvent également constituer des mécanismes d'échappement. À l'inverse, il n'est pas non plus aisé de définir un mécanisme d'échange souple et paramétrable.

44. Chaque *thread* possède un champ *context class loader* qui est défini à la création du *thread* et qui peut être modifié à l'aide de la méthode `setContextClassLoader` de la classe `java.lang.Thread`.

Le dernier risque concerne les interactions entre les chargeurs de classes et le contrôle d'accès. Le développeur peut, en se concentrant sur l'implémentation des fonctions principales du chargeur de classes, omettre l'implémentation des points de contrôle ou du mécanisme d'association des domaines de protection. Ces mécanismes sont critiques :

- l'absence de point de contrôle peut conduire à une utilisation abusive du chargeur par un code malicieux ;
- l'absence d'association des domaines de protection constitue potentiellement une faille dans le contrôle d'accès puisque celui-ci s'appuie sur les domaines de protection associés à chaque classe pour mettre en œuvre la politique de sécurité. Une classe qui ne possède pas de domaine de protection n'est pas prise en compte par le mécanisme de contrôle (c'est le cas des classes système) : elle possède donc virtuellement toutes les permissions.

En outre, l'utilisation des permissions statiques peut s'avérer déroutante pour un administrateur puisqu'elles n'apparaissent pas dans la définition de la politique de sécurité.

6.3.6 Recommandations

En raison de la complexité évoquée dans la section précédente, le développement d'un chargeur de classes doit donc être envisagé uniquement dans les cas qui le nécessitent explicitement. Il doit être assuré par un développeur Java qui maîtrise ce mécanisme et qui suit les recommandations évoquées dans ce document et dans le guide de recommandations [13].

En particulier, le développeur doit respecter autant que faire se peut le comportement par défaut. Il doit également ne pas omettre l'implémentation des points de contrôle et du mécanisme d'association des domaines de sécurité.

Les limites du cloisonnement qu'il est possible de mettre en place à l'aide de chargeur de classes justifient le recours au système d'exploitation afin de confiner les différentes applications Java. Celles-ci s'exécutent en général au sein de différentes instances de la JVM (correspondant à différents processus du système d'exploitation). Seules certaines implémentations, notamment dans le domaine de l'embarqué, permettent d'utiliser une même instance de la JVM pour différentes applications Java. Les « applications » Web constituent un cas à part puisqu'elles s'appuient sur une application Java SE (le serveur d'applications ou le conteneur de *Servlet/JSP*) qui gère les différentes interactions entre ces « applications ». Le mécanisme de chargement de classes n'est pas suffisant pour implémenter un environnement Java « multi-processus » (au sein d'une instance commune de la JVM). Se pose notamment le problème de la gestion des exceptions qui peuvent avoir un impact sur l'ensemble des « processus » Java de la JVM.

Des projets ont été initiés pour inclure dans la spécification de la plate-forme Java un mécanisme dédié au confinement de différentes applications Java s'exécutant sur une unique instance de la JVM. C'est notamment le cas de l'API d'isolation Java (*Java isolate API*) qui a fait l'objet

d'une démarche de spécification (JSR 121 [45]). Il s'agit de pouvoir exécuter différentes applications sur une même plate-forme Java (appelée *aggregate*), au sein d'environnements confinés (appelés *isolates*). Par défaut, les environnements confinés partagent peu de données Java (ils disposent de leur propre tas), mais ils s'appuient en revanche sur les mêmes services offerts par la JVM. La plate-forme repose également sur une extension de l'API de la bibliothèque standard Java permettant d'implémenter des moyens de communication entre les domaines confinés. Ces échanges peuvent être contrôlés par un mécanisme adéquat reposant sur la définition d'une politique de confinement. Toutefois, la spécification ne précise pas les contraintes qui doivent s'appliquer sur le partage des ressources. De plus, elle ne précise pas la stratégie d'implémentation (même si la plupart des implémentations s'appuient sur une instance unique de la JVM s'exécutant sur un ou plusieurs processus OS). À l'heure actuelle, cette spécification n'a pas été intégrée à la plate-forme standard. Initialement, il était prévu de l'intégrer à la version 1.5 du JRE, mais cette intégration a été successivement reportée et elle ne semble pas à l'ordre du jour pour la future version standard (1.7). De fait, il n'existe que des implémentations expérimentales, notamment au sein du projet de recherche Joe [46] (ex Barcelona) de Sun.

6.4 Cryptographie

6.4.1 Présentation des API cryptographiques

Nativement, Java J2SE6 propose un panel d'API cryptographiques pour sécuriser des applications Java. Ces API et ces *frameworks* sont les suivants :

- les API cryptographiques à proprement parler :
  - JCA, JCE : *Java Cryptography Architecture* et *Java Cryptography Extension* permettent l'utilisation de fonctions cryptographiques telles que le chiffrement, la génération de clés, la signature, etc.
- les APIs de communications sécurisées :
  - JSSE et JGSS : *Java Secure Socket Extension* et *Java Generic Security Service* permettent de mettre en place des communications sécurisées (support des protocoles cryptographiques) ;
  - JSASL : *Simple Authentication and Security Layer* apporte un support de mécanismes d'authentification et d'autorisation normalisé.
- les APIs de gestion de PKI :
  - JCP : *Java CertPath* permet la gestion des PKI et trousseaux de clés ;
  - le *package* `Java.securityCert` fournit des classes permettant de gérer les certificats ;

45. `http://jcp.org/jsr/detail/121.jsp`
46. `http://research.sun.com/projects/joe/`

- le support de l'OCSP *On-Line Certificate Status Protocol* permet de gérer l'état (notamment la révocation) des certificats X.509 ;
- Java PKCS#11 : Ce *framework* permet d'accéder aux services du standard PKCS#11 permettant la gestion des *tokens* cryptographiques.

6.4.2 Caractéristiques des API cryptographiques

6.4.2.1 Présentation

JCA (*Java Cryptography Architecture*) est apparue la première fois dans le JDK 1.1. À ses débuts, les services offerts se limitaient à des fonctions de hachage et à des algorithmes de signatures numériques. Par la suite, Java 2 SDK a étendu JCA afin de fournir plus de mécanismes cryptographiques. À partir de ce moment, JCA possède déjà une architecture basée sur le système de *provider* (qui est explicité par la suite). Cette architecture a pour objectif d'assurer l'interopérabilité entre plusieurs implémentations de services cryptographiques en s'appuyant notamment sur des conventions et des spécifications bien définies.

Lorsque Java 2 SDK est disponible, il couvre l'ensemble des APIs cryptographiques définies dans le SDK et celles définies dans JCE 1.2 (Java Cryptography Extension). JCE est un *package* optionnel fournissant des APIs de chiffrement, d'échange de clés, de MAC, etc. À partir de Java 2 SDK 1.4, JCE est intégré dans le SDK.

JCA suit deux principes :

- indépendance et extensibilité des algorithmes ;
- indépendance et interopérabilité des implémentations.

Le premier principe est réalisé en définissant un ensemble de classes de haut niveau (généralement appelées *engine class*) représentant chacune un type de service. Ces classes constituent une interface unique et indépendante des algorithmes cryptographiques sous-jacents et de leurs implémentations. Lorsqu'une application souhaite utiliser un service cryptographique, cela se fait toujours par le biais de ces classes de haut niveau quel que soit l'implémentation ou l'algorithme (d'un type de service donné) utilisé. JCA définit les services cryptographiques (classes de haut niveau) suivants :

- `SecureRandom` : utilisée pour générer des nombres aléatoires ou pseudo-aléatoires ;
- `MessageDigest` : utilisée pour calculer le condensé d'une donnée ;
- `Signature` : utilisée pour signer des données ou vérifier des signatures ;
- `Cipher` : utilisée pour chiffrer / déchiffrer des données ;
- `Mac` : utilisée pour calculer un motif d'intégrité de message ;
- `KeyFactory` : utilisée pour convertir des clés de chiffrement en type `Key` et vice versa ;

- `SecretKeyFactory` : utilisée pour convertir des clés de chiffrement en type `SecretKey` et vice versa ;
- `KeyPairGenerator` : utilisée pour la génération de bi-clés ;
- `KeyGenerator` : utilisée pour la génération de clés ;
- `KeyAgreement` : utilisée par une ou plusieurs entités pour se mettre en accord d'une clé ;
- `AlgorithmParameters` : utilisée pour stocker les paramètres d'un algorithme particulier ;
- `AlgorithmParameterGenerator` : utilisée pour générer un ensemble de paramètres d'algorithme ;
- `KeyStore` : utilisée pour créer et gérer des trousseaux de clés ;
- `Certificatefactory` : utilisée pour créer un certificat d'une clé publique ou des listes de revocation (CRL) ;
- `CertPathBuilder` : utilisée pour construire une chaîne de certification ;
- `CertStore` : utilisée pour récupérer des certificats et CRL.

Le second principe est réalisé par le biais d'une architecture basée sur des *providers*. Un *provider* dans le contexte de JCA est un fournisseur d'implémentation de services cryptographiques (CSP, *Cryptographic Service Provider*). Il se présente sous la forme d'une ou plusieurs archives JAR intégrant des implémentations d'algorithmes cryptographiques. L'architecture basée sur des *providers* permet de faire cohabiter des implémentations différentes d'un même algorithme au sein d'une même plate-forme d'exécution Java. L'utilisation des *providers* permet de faciliter l'insertion de nouveaux algorithmes ou de nouvelles implémentations. Le chapitre 10.2 en annexe illustre l'architecture des *providers*.

En pratique, l'utilisation d'un service cryptographique est réalisée via l'une des classes de haut niveau qui se charge d'instancier l'implémentation de l'algorithme souhaité selon le *provider* désiré. L'utilisation des classes de haut niveau dans une application présente un certain nombre d'intérêts :

- elle garantit une certaine forme d'homogénéité du code ;
- les appels aux services cryptographiques sont plus simples pour le développeur (des exemples d'appels sont présentés en annexe en section 10.2) ;
- les mises à jour des services sont simplifiées et ont un faible impact sur le code de l'application.

6.4.2.2 Risques

JCA permet de gérer le cycle de vie des clés de chiffrement à l'exception de l'effacement. JCA possède les services adéquats pour générer, convertir et stocker les clés. Toutefois, l'effacement sécurisé en mémoire des clés n'existe pas. Les clés sont gérées comment des objets Java

classiques. La suppression d'une clé est à la charge du *garbage collector* lorsque cette dernière n'est plus référencée. La suppression de l'objet clé ne signifie pas que la mémoire est mise à zéro.

L'indépendance vis-à-vis des implémentations d'algorithmes n'est pas complète concernant les clés de chiffrement. En effet, d'une implémentation à l'autre, la représentation (encodage en mémoire) des clés peut varier d'un *provider* à l'autre : une clé générée par un *provider* A n'est pas forcément utilisable par un algorithme implémenté par un *provider* B. Il est donc nécessaire d'avoir recours aux classes de haut niveau `KeyFactory` et `SecretKeyFactory` pour réaliser des conversions de clés et de bi-clés. En général, chaque *provider* implémente ses services.

Remarque 10 *L'utilisation d'une clé générée pour un algorithme de chiffrement d'un* provider *A, pour un autre algorithme d'un* provider *B peut engendrer un comportement erroné qui ne sera pas forcément détecté. Nous avons fait le test le suivant :*

1. *génération d'une bi-clé RSA avec le* provider *de Sun.*
2. *chiffrement d'un message avec cette bi-clé en utilisant une implémentation d'un algorithme fournie par le* provider *Sun.*
3. *déchiffrement du message chiffré avec cette bi-clé en utilisant une seconde implémentation du même algorithme fournie par le* provider *BouncyCastle.*

À l'issue du test, nous avons remarqué que le message déchiffré n'est pas le bon et qu'aucune exception n'a été levée.

6.4.2.3 Recommandations

Il est nécessaire, dans la mesure du possible, d'effacer manuellement les informations confidentielles. Ceci peut en partie être réalisé par le développeur à l'aide d'une méthode d'effacement qui doit être appelée explicitement dès lors que l'information n'est plus utilisée et à défaut dans la méthode `finalize` de la classe. Ce point fera l'objet de recommandations à l'intention du développeur [13]. Des modifications de la JVM permettant de gérer au mieux les données confidentielles peuvent également être envisagées. Ces modifications seront évoquées dans le rapport « Stratégie d'évolution et d'implémentation d'une JVM pour l'exécution d'applications de sécurité » [15].

6.4.3 Caractéristiques des API permettant de sécuriser un canal de communication

6.4.3.1 Java Secure Socket Extension

L'API JSSE supporte les versions 2.0 et 3.0 de SSL et la version 1.0 de TLS. Ces protocoles de sécurité sont utilisés pour encapsuler des données entre deux entités de manière bidirectionnelle. De plus, l'API apporte un support en authentification, en chiffrement et en protection de l'intégrité. JSSE ajoute à la bibliothèque standard ses propres implémentations d'algorithmes cryptographiques qui sont définies dans les packages `java.security` et `java.net`. JSSE est construit de la même manière que le *framework JCA*. Il est indépendant vis-à-vis de l'implémentation des algorithmes. Il utilise de la même manière la notion de CSP ou *provider*.

Les caractéristiques globales de JSSE sont les suivantes :

- il est intégré à la bibliothèque standard depuis la version 1.4 du JRE ;
- il utilise la notion de CSP, il est dit « ouvert » à de nouveaux algorithmes ;
- il est implémenté entièrement en Java ;
- il est compatible avec SSL v2, v3 et TLS v1 ;
- il utilise des classes dédiées à la création de tunnels sécurisés (`SSLSocket`, `SSLServerSocket` et `SSLEngine`) ;
- il supporte les négociations de clés entre les entités ;
- il supporte le protocole HTTP encapsulé, (HTTPS) ;
- il permet la gestion de la mémoire des sessions SSL ;
- il implémente des algorithmes de chiffrement (voir le tableau ci-dessous).

Algorithmes implémentés		
Algorithme	**Fonction**	**Longueur des clés**
RSA	Authentification et échange de clés	$>$ 512 bits
RC4	Chiffrement	40 à 128 bits
DES	Chiffrement	56 bits
Triple DES	Chiffrement	168 bits
AES	Chiffrement	256, 128 bits
Diffie-Hellman	Négociation de clés	1024, 512 bits
DSA	Authentification	1024 bits

L'API JSSE est disponible dans les packages `javax.net` et `javax.net.ssl`. Elle couvre :

- les *sockets* sécurisés (`SSL`) et les *sockets* côté serveur ;

- *SSLEngine*, un mécanisme de non-blocage des flux produits et consommés par les données SSL et TLS ;
- la création de *socket*, de *socket* SSL, etc. ;
- la gestion des clés et la gestion des interfaces de confiance ;
- la sécurisation de HTTP (connexion `HTTPS`).

L'implémentation de Sun de la bibliothèque standard inclut un CSP nommé SunJSSE qui est pré-installé et pré-configuré avec JCA. Ce CSP fournit les services suivants :

- une implémentation des protocoles SSL 3.0 et TLS 1.0 ;
- une implémentation des algorithmes les plus connus et les plus utilisés par SSL et TLS pour l'authentification, la négociation des clés, le chiffrement et la protection de l'intégrité ;
- une implémentation de la gestion des clés X.509 (clés authentification) ;
- une implémentation de la gestion de confiance X.509 (sous la forme de règles sur la chaîne de confiance) ;
- une implémentation de PKCS#12.

6.4.3.2 Java GSS-API

Java GSS-API (JGSS) est utilisée pour sécuriser les échanges de messages entre deux applications. JGSS est défini dans la RFC *Java Bindings* (RFC2853). Il fournit aux développeurs des mécanismes de sécurité s'appuyant sur Kerberos.

Des services additionnels sont disponibles pour la sécurisation des communications, à savoir :

- la délégation ;
- l'authentification mutuelle ;
- les mécanismes anti-rejeu ;
- les mécanismes de non répudiation ;
- l'authentification anonyme.

6.4.3.3 Java Simple Authentication and Security Layer (SASL)

SASL est un standard internet défini dans la RFC2222. SASL spécifie un protocole d'authentification et d'établissement de tunnel sécurisé entre un client et un serveur d'applications. L'API Java SASL définit des classes et des interfaces pour les applications qui utilisent les mécanismes de SASL. L'API supporte les applications clientes et serveurs. Elle permet à l'application de sélectionner un mécanisme de sécurité à sa convenance. L'API Java SASL permet

également aux développeurs d'utiliser leurs propres mécanismes SASL. Ces mécanismes de SASL sont installés avec l'API JCA.

SASL est un protocole de défi-réponse. Le protocole est prévu pour supporter les méthodes d'authentification utilisant plusieurs défis-réponses.

L'API Java SASL possède deux interfaces `SaslClient` et `SaslServer`. Comme leurs noms l'indiquent, l'une est côté client, et l'autre côté serveur.

6.4.4 Caractéristiques des API de gestion des PKI

6.4.4.1 Java.security.cert

Le *package* `java.security.cert` fournit une API permettant la gestion des certificats X.509. Les services (classes) inclus dans le *package* `java.security.cert` sont les suivants :

- la classe `CertificateFactory` est utilisée pour la génération de certificats et de la liste de révocation de certificats (CRL) ;
- la classe `Certificate` (abstraite) est utilisée pour la gestion des différents types de certificats ;
- la classe `CRL` (abstraite) est utilisée pour la gestion des différents types de listes de révocation de certificats (CRL) ;
- la classe `X509Certificate` (abstraite) permet d'accéder à tous les attributs d'un certificat X.509 ;
- la classe `X509Extension` (interface) définit les extensions des certificats X.509 v3 et CRL v2 ;
- la classe `X509CRL` (abstraite) définit une liste de révocation de certificats X.509. Elle est signée par l'autorité de certification ;
- la classe `X509CRLEntry` est une classe abstraite de `CRL Entry`.

6.4.4.2 API Java Certification Path

L'API *Java Certification Path* est constituée de classes et d'interfaces permettant de gérer les chaînes de certification [47]. Cette API permet de créer et de valider les chaînes de certificats. À l'instar de JCA, le développeur doit utiliser un CSP ou *provider*. L'API inclut également les algorithmes les plus utilisés pour générer et valider les chaînes de certificats X.509 selon les standards PKIX (standard développé par le groupe IETF PKIX).

47. Une chaîne de certification est une liste ordonnée de certificats.

L'API est définie sur une extension du *package* `java.security.cert`. Les classes peuvent être réparties en quatre catégories : `basic`, `validation`, `création` et `stockage` :

Basic : `CertPath`, `CertificationFactory`, `CertPathParameters`.

Validation : `CertPathValidator`, `CertPathValidatorResult`.

Création : `CertPathBuilder`, `CertPathBuilderResult`.

Stockage : `CertStore`, `CertStoreParameters`, `CertSelector`, `CRLSelector`.

6.4.4.3 Support de l'OCSP Java

Le support côté client pour l'OCSP (*On-Line Certificate Status Protocol*) défini dans la RFC2560 est supporté par Java (depuis la version 1.5.0 de Java). L'OCSP fait partie de l'API *Java certPath*. L'OCSP est contrôlé par cinq propriétés de sécurité :

ocsp.enable : active ou non l'OCSP ;

ocps.responderURL : identifie où se situe le *responder* de l'OCSP ;

ocsp.responderCertSubjectName : nom du certificat du *responder* de l'OCSP ;

ocsp.responderCertIssuerName : nom de l'émetteur du certificat du *responder* de l'OCSP ;

ocsp.responderCertSerialNumber : numéro de série du certificat du *responder* de l'OCSP.

6.4.4.4 Java PKCS#11

À l'instar de JCA, Java Sun fournit un CSP dédié à PKCS#11. PKCS#11 est produit par la société RSA Security. *A contrario* des autres CSP, le CSP PKCS#11 de Sun n'implémente pas d'algorithmes cryptographiques lui-même. Il fait office de pont entre l'API Java JCA/JCE et l'API native PKCS#11. Le CSP traduit les communications entre ces deux parties.

Par l'intermédiaire de ce CSP, l'API JCA/JCE peut utiliser les services implémentés par PKCS#11 comme par exemple :

- les mécanismes cryptographiques pour les SmartCard ;
- l'accélérateur cryptographique matériel ;
- l'augmentation des performances logicielles due à l'implémentation.

Le CSP permet de simplifier les accès à l'implémentation native de PKCS#11.

6.4.5 Présentation de la démarche d'analyse

6.4.5.1 Description fonctionnelle de JCA

L'acronyme « JCA » signifie *Java Cryptography Architecture*. Le document d'entrée servant de référence à cette analyse est le guide de référence de JCA pour Java SE 6. Ce guide expose les fonctionnalités et les spécifications des fonctions de l'API. Il n'est identifié par aucune version d'édition.

L'objectif de l'API JCA est de permettre l'utilisation aisée et facilitée de fonctions de cryptographie, telles que le chiffrement, la génération de clés et la signature, aux implémentations Java SE 6. L'API elle-même est implémentée en langage Java.

6.4.5.2 Critères d'évaluation de la pile

Introduction Ce paragraphe instancie la métrique d'évaluation d'une API cryptographique propre au laboratoire d'évaluation d'Amossys à l'API JCA.

Niveau de l'API L'API JCA est dite de **haut niveau**, c'est-à-dire qu'elle n'a pas connaissance des algorithmes utilisés ou des services de sécurité associés (clés, certificats, ...).

L'application demande donc un service, par exemple :

- confidentialité ;
- intégrité ;
- authentification.

Indépendance de l'API vis-à-vis des algorithmes L'API JCA est indépendante vis-à-vis des algorithmes utilisés. JCA assure l'indépendance des algorithmes en définissant des classes de haut niveaux pour différentes familles de services cryptographiques (chiffrement, signature, fonction de hachage, génération de clés, ...). Chacune de ces classes possède une interface que doivent respecter les implémentations sous-jacentes des algorithmes. Ces classes sont appelées *engine classes*.

Les algorithmes sont gérés par des CSP (*Cryptographic Service Provider* ou *provider*) qu'il est possible de choisir via un ordre particulier. Pour mémoire, le CSP est constitué par un ensemble de classes Java, généralement regroupées dans une même archive JAR.

L'utilisation des CSP permet de simplifier l'usage des algorithmes cryptographiques. Par

exemple, un programme souhaitant réaliser une signature DSA n'a qu'à instancier la classe haut niveau *Signature* en précisant le nom de l'algorithme souhaité : `Signature engine = Signature.getInstance("DSA", "SUN")`.

Pour illustrer cette indépendance, le code présenté en annexe 10.3.1 expose la création d'un objet `java.security.SecureRandom`.

Indépendance de l'API vis-à-vis des applications L'API JCA est indépendante vis-à-vis des applications grâce à la notion de *provider* dont le fonctionnement est présenté en annexe 10.2. Que ce soient les algorithmes d'un *provider* ou d'un autre qui sont utilisés, les appels à ces services seront toujours réalisés par le biais des classes de haut niveaux. L'architecture de *provider* permet de faire le lien de manière transparente entre un algorithme souhaité et une implémentation de cet algorithme.

Par défaut, Sun distribue plusieurs implémentations de services et d'algorithmes cryptographiques via les *providers* suivants :

- `Sun` ;
- `SunJSSE` ;
- `SunJCE` ;
- `SunRsaSign`.

Enfin, l'architecture de JCA permet facilement de mettre à jour les *providers*. En effet, les interfaces consituées par les classes de haut niveau sont fixes, seules les implémentations sollicitées derrière ces interfaces sont modifiées.

Sûreté de programmation La sûreté de programmation est basée sur trois critères :

- les conventions de nommage cohérentes ;
- le niveau d'opacité des informations ;
- la facilité d'utilisation.

L'API JCA utilise des conventions de nommage cohérentes pour les méthodes, constantes et types de données. Les méthodes ont toutes des commentaires très détaillés (Javadoc). Un exemple d'utilisation de ces conventions de nommages et commentaires est fourni en annexe 10.3.2. Les commentaires informent du service rendu par la méthode en indiquant ses données d'entrées et ce qu'elle retourne. Les paramètres et variables utilisés sont compréhensibles et facilement utilisables. L'utilisation de l'API JCA est intuitive pour un développeur Java. Le guide d'utilisation de l'API est très clair et propose des exemples concrets et simples d'approche.

Sécurité de l'API À ce jour, aucune faille de sécurité n'a été relevée ou identifiée **publiquement**. En général, les vulnérabilités proviennent surtout des éléments gravitant autour

de l'API : JVM, JRE, fichiers de configurations. Toutefois, la configuration des *providers* doit faire l'objet de beaucoup d'attention, car il s'avère relativement aisé de corrompre l'ensemble d'une application en insérant un *provider* piégé en tête de la liste de préférences. En effet, certaines méthodes prennent par défaut le premier *provider* de la liste des *providers* lorsqu'aucun *provider* n'est spécifié. Ce cas de figure se retrouve par exemple dans la classe de l'API JCA implémentant le `java.security.SecureRandom`, présentée en annexe 10.3.3.

Cette remarque est valable pour toutes les applications ou les *providers* faisant appel aux services cryptographiques qui n'explicitent pas la provenance des implémentations des algorithmes utilisés.

Services auxiliaires offerts par l'API Les services de base de l'API JCA sont les services cryptographiques natifs, tels que :

- la fonction de génération de nombres aléatoires ;
- les fonctions de hachage ;
- les fonctions de chiffrement/déchiffrement ;
- les fonctions de calcul et de vérification de signature.

Ces services peuvent être complétés par des services dits « auxiliaires » liés à la gestion de la sécurité :

- gestion du cycle de vie des clés ;
- gestion des certificats ;
- authentification des utilisateurs ;
- commnunication sécurisée.

La gestion du cycle de vie des clés est assurée par les services de haut niveau suivants :

- `KeyFactory` : ce service est utilisé pour convertir des clés de chiffrement en type `Key` et vice versa. Il est notamment utilisé pour assurer la compatibilité des clés d'un *provider* à l'autre ;
- `SecretKeyFactory` : ce service est utilisé pour convertir des clés de chiffrement en type `SecretKey` et vice versa. Il est notamment utilisé pour assurer la compatibilité des clés d'un *provider* à l'autre ;
- `KeyPairGenerator` : ce service est utilisé pour la génération de bi-clés ;
- `KeyGenerator` : ce service est utilisé pour la génération de clés ;
- `KeyStore` : ce service est utilisé pour créer et gérer des trousseaux de clés.

La gestion des certificats est assurée par les services de haut niveau suivants :

- `Certificatefactory` : ce service est utilisé pour créer un certificat à clé publique ou des listes de révocation (CRL) ;

- `CertPathBuilder` : ce service est utilisé pour construire une chaîne de certification ;
- `CertStore` : ce service est utilisé pour récupérer des certificats et des listes de révocation.

JCA ne dispose pas de service dédié à l'authentification des utilisateurs (même si elle regroupe tous les algorithmes nécessaires). Toutefois un *framework* adjacent à JCA peut être utilisé, il s'agit de JAAS (*Java Authentication and Authorization Service*). JAAS fait partie intégrante du JRE depuis Java 1.4.

JCA ne dispose pas de service pour sécuriser des communications, mais des *frameworks* externes peuvent être utilisés, il s'agit de :

- JSSE *Java Secure Socket Extension* : ce *framework* implémente TLS et est fourni avec la JRE. Les fonctions de JSSE sont accessibles par le biais du *provider* SunJSSE ;
- JGSS *Java Generic Security Service* : il s'agit d'un *framework* basé sur Kerberos v5. Les fonctions de JGSS sont accessibles par le biais du *provider* SunJGSS ;
- JSASL *Java Simple Authentication and Security Layer* : ce *framework* implémente SASL. Les fonctions de JSASL sont accessibles par le biais du *provider* SunSASL .

Support des requêtes asynchrones JCA ne supporte pas le fonctionnement en mode asynchrone. Elle est en mode synchrone. Les services, une fois exécutés, sont bloquants. Par exemple une méthode de chiffrement ne retourne qu'une fois l'opération effectuée.

Partage de charge par l'API JCA n'offre pas le partage de charge. Cette tâche est à la charge de la JVM.

Priorités des requêtes JCA n'offre pas la possibilité d'indiquer et/ou d'offrir la priorité de traitement des requêtes.

Mécanisme de contrôle de flux L'API JCA ne dispose pas de mécanismes prévus à cet effet, puisque son mode de fonctionnement est synchrone. Concernant la gestion du *multithreading*, cette tâche est réalisée par la JVM.

Tableau récapitulatif des critères Pour résumer les critères d'évaluation de JCA, on constate que le *framework* est une API cryptographique de haut niveau garantissant l'indépendance vis-à-vis des algorithmes, de leurs implémentations et des applications. Cela est rendu possible en définissant une architecture basée sur les *providers* qui permet de regrouper les implémentations des mécanismes cryptographiques. L'utilisation de ces mécanismes se fait par le

biais de classes de haut niveau (*engine class*) qui abstrait les problématiques liées à l'implémentation des algorithmes ou à la manière d'accéder aux services. Les mécanismes cryptographiques sont regroupés par famille de service à qui on attribue une unique *engine class*.

Enfin, JCA ne gère pas les mécanismes liés à l'asynchronisme, au partage de charge ou au contrôle de flux. Ces tâches sont de la responsabilité de la JVM.

Critère	**Valeur du Critère**	**Métrique**
Niveau de l'API	Haut	Bas ou Haut
Indépendance vis-à-vis des algorithmes	Oui	Oui ou Non
Indépendance vis-à-vis des applications	Oui	Oui ou Non
Indépendance vis-à-vis du module cryptographique	Oui	Oui ou Non
Degré d'expertise cryptographique	3	1 Nulle - 5 Excellente
Services auxiliaires - Gestion du cycle de vie des clés - Gestion des certificats - Authentification des utilisateurs - Commnunication sécurisée	 Oui Oui avec JCP Oui avec JAAS Oui avec JSSE et JGSS	
Sûreté de programmation	2	1 Nulle - 5 Excellente
Gestion de l'asynchronisme	Non	Oui ou Non
Partage de charge	Non (JVM)	Oui ou Non
Priorité des requêtes	Non	Oui ou Non
Mécanisme de contrôle de flux	Non	Oui ou Non

TABLE 8 – Récapitulatif des critères d'évaluation de JCA

6.4.6 Analyse de l'implémentation

6.4.6.1 Sélection des composants logiciels pertinents

L'objet de ce chapitre est de déterminer les briques logicielles qui vont être étudiées par la suite. Ces briques sont réparties selon leur appartenance à un type de service cryptogra-

phique, chaque type de service étant représenté par sa classe de haut niveau : `SecureRandom`, `MessageDigest`, `Signature`, `Cipher`, `Mac`, `KeyFactory`, `SecretKeyFactory`, ... (Cf. 6.4.2)

Cette étude a pour objectif d'évaluer par échantillonnage les services cryptographiques. Nous nous concentrerons sur les services qui nous semblent intéressant d'étudier de part :

- la manière dont sont implémentés les services cryptographiques : structuration du code, effacement des données sensibles, etc. ;
- l'usage des autres services cryptographiques ;
- les choix d'implémentation.

Les services concernés par l'étude sont les suivants :

- `SecureRandom` : la génération d'aléa est un élément crucial de la cryptographie. Cet élément est souvent implémenté de manière incorrecte dans les produits. L'étude de ce service permet donc de juger du niveau de compétence et de maturité des concepteurs ;
- `KeyPairGenerator` et `Key` : les clés de chiffrement sont des éléments sensibles qui doivent être effacés de manière sécurisée lorsqu'ils ne sont plus nécessaires. De plus, la création de bi-clés (par exemple RSA) nécessite l'utilisation de données temporaires qui doivent être effacées à la fin de la génération. L'étude du service de création et des structures des clés permettra de juger de la qualité de l'implémentation des services cryptographiques.

Ces services sont susceptibles de faire appel à d'autres services cryptographiques. L'étude permettra donc de vérifier la manière dont sont réalisés ces appels.

6.4.6.2 Présentation des briques logicielles

SecureRandom Par défaut le service de génération d'aléa est utilisé via la classe de haut niveau `java.security.SecureRandom`. Cette classe possède un attribut nommé `secureRandomSpi` de type `java.security.SecureRandomSpi`. Cet attribut va constituer le générateur en tant que tel. L'objet référencé par cet attribut provient de la classe du générateur d'aléa qui doit hériter de la classe `java.security.SecureRandomSpi`. Les requêtes passées à la classe de haut niveau sont ensuite relayées à cet objet.

La classe `java.security.SecureRandomSpi` est une interface définissant les méthodes que doit implémenter la classe du générateur d'aléa sous-jacent. Cette interface définit les méthodes suivantes :

Méthodes interfacées	Descriptif
`protected abstract byte[] engineGenerateSeed(int numBytes)`	Retourne un tableau d'octets de longueur donnée pouvant servir de graine pour un autre générateur d'aléa.
`protected abstract void engineNextBytes(byte[] bytes)`	Génère des octets aléatoirement. Le nombre d'octets générés est spécifié implicitement par l'utilisateur via la taille du tableau passé en paramètre.
`protected abstract void engineSetSeed(byte[] seed)`	Réinitialise la clé du générateur.

En se basant sur le code source d'OpenJDK, l'implémentation *open source* du JDK de Sun, les générateurs d'aléa déjà implémentés sont les suivants :

CSP	Nom de l'algorithme	Classe
SUN	SHA1PRNG	`sun.security.provider.SecureRandom`
SUN	NativePRNG	`sun.security.provider.NativePRNG`
SunMSCAPI	Windows-PRNG	`sun.security.mscapi.PRNG`

Remarque 11 – *Étant basé sur `/dev/random` et `/dev/urandom`, `NativePRNG` n'est disponible que sur les systèmes UNIX ;*
– *`Windows-PRNG` n'est disponible que sur les systèmes Windows.*

Nous étudierons les implémentations SHA1PRNG de Sun et Windows-PRNG de SunMSCAPI.

KeyPairGenerator Par défaut, le service de génération de bi-clés est accessible par l'intermédiaire de la classe de haut niveau `java.security.KeyPairGenerator`. Toutefois, l'application utilisant ce service envoie ses requêtes directement à la classe implémentant le générateur de bi-clés. Cette classe doit hériter de la classe `java.security.KeyPairGenerator` qui hérite de `java.security.KeyPairGeneratorSpi`. Lorsqu'une application demande à utiliser ce service pour un algorithme donné, un objet est alors instancié suivant la classe implémentant le générateur approprié.

Les classes implémentant les algorithmes doivent posséder les méthodes suivantes définies par la classe abstraite `java.security.KeyPairGeneratorSpi` :

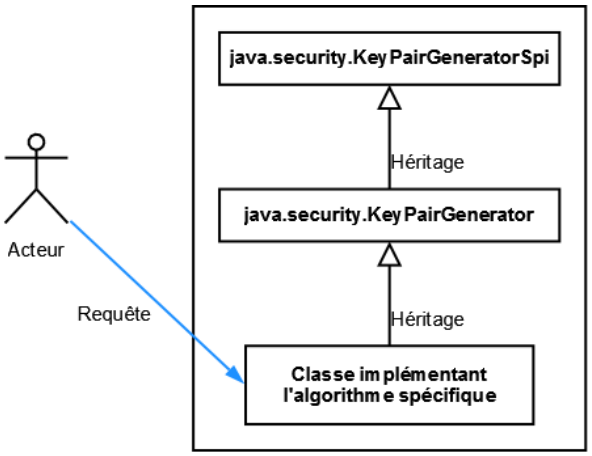


FIGURE 12 – Architecture du service de génération de bi-clés

Méthodes interfacées	Descriptif
`abstract KeyPair generateKeyPair()`	Permet de générer une paire de clés
`void initialize(AlgorithmParameterSpec params, SecureRandom random)`	Initialise le générateur de paires de clés en utilisant l'ensemble des paramètres spécifiés et une source d'aléa fournie par l'utilisateur.
`abstract void initialize(int keysize, SecureRandom random)`	Initialise le générateur de paires de clés pour une longueur de clé donnée en utilisant l'ensemble des paramètres spécifiés.

En se basant sur le code source d'OpenJDK, les implémentations de générateur de clés sont les suivantes :

CSP	Nom de l'algorithme	Classe
SUN	DSA	`sun.security.provider.DSAKeyPairGenerator`
SunRsaSign	RSA	`sun.security.rsa.RSAKeyPairGenerator`
SunJSSE	RSA	`sun.security.rsa.RSAKeyPairGenerator`
SunJCE	DiffieHellman	`com.sun.crypto.provider.DHKeyPairGenerator`
SunMSCAPI	RSA	`sun.security.mscapi.RSAKeyPairGenerator`

Dans ce document, nous étudions le générateur de bi-clés DSA de Sun.

Key La classe `Key` est une interface de haut niveau pour toutes les classes se réferant aux clés. Cette classe définit trois caractéristiques communes à toutes les clés :

- le nom de l'algorithme qui concerne la clé ;
- le nom de la méthode d'encodage de la clé ;
- l'encodage de la clé.

De manière équivalente, l'interface `java.security.spec.KeySpec` permet de regrouper les classes spécifiant les clés.

Les classes héritant de `KeySpec` définissent un moyen commun à tous les CSP de représenter un type de clé ou de bi-clés (clé RSA, clé DSA, clé symétrique, ...). Ces classes sont définies par l'architecture JCA. Elles comprennent notamment les classes suivantes :

Type de clés	**Classe**
DSA (privée)	`sun.security.spec.DSASecretKeySpec`
DSA (publique)	`sun.security.spec.DSAPublicKeySpec`
RSA (privé)	`sun.security.spec.RSASecretKeySpec`
RSA (publique)	`sun.security.spec.RSAPublicKeySpec`
Courbe elliptique (privée)	`sun.security.spec.ECSecretKeySpec`
Courbe elliptique (publique)	`sun.security.spec.ECPublicKeySpec`
Diffie-Hellman (privée)	`javax.crypto.spec.DHSecretKeySpec`
Diffie-Hellman (publique)	`javax.crypto.spec.DHPublicKeySpec`
Clé symétrique	`javax.crypto.spec.SecretKeySpec`
Clé DES	`javax.crypto.spec.DESKeySpec`

Quant aux classes héritant de `Key`, elles définissent des moyens de représentation (via l'encodage) qui ne sont pas nécessairement communs à tous les CSP. Seules les méthodes imposées par l'héritage de la classe `Key` sont communes. Ces méthodes permettent à une application de connaître la manière dont la clé est encodée.

Le recours aux instances de la classe `Key` et aux spécifications de clés de type `KeySpec` permet à un CSP *A* d'utiliser des clés générées par un CSP *B*.

Etant donné le très grand nombre de classes impliquées dans les mécanismes cryptographiques, toutes les classes relatives à la gestion des clés ne sont pas abordées dans cette étude. Celle-ci se concentre sur les mécanismes d'effacement des clés après usage.

Notre analyse du code source nous a montré qu'aucun mécanisme de ce type n'est imposé par les classes de haut niveau. Par conséquent, l'implémentation d'un tel mécanisme est à la charge des développeurs de CSP ou des applications. En parcourant le code source, nous avons

pu constater que ce mécanisme est dans la majorité des cas absent. L'effacement des clés dans la mémoire est à la charge du *garbage collector*. L'analyse du *garbage collector* fera l'objet d'une étude complémentaire. A priori, le *garbage collector* des JVM existantes ne fait qu'assurer la libération de la mémoire allouée : aucune procédure d'effacement de la mémoire n'est réalisée.

Dans les cas où un tel mécanisme est présent, il est implémenté via la redéfinition de la méthode `finalize`. Cette méthode est appelée par le *garbage collector* lors de la libération de la mémoire. Ci-dessous, le code Java utilisé :

```
protected void finalize() throws Throwable {
  try {
    if (this.key != null) {
      java.util.Arrays.fill(this.key, (byte)0x00);
      this.key = null;
    }
  }
  finally {
    super.finalize();
  }
}
```

Les classes utilisant ce procédé sont :

- `com.sun.crypto.provider.DESedeKey;`
- `com.sun.crypto.provider.DESKey;`
- `com.sun.crypto.provider.PBEKey;`
- `com.sun.crypto.provider.PBKDF2KeyImpl.`

L'utilisation de la méthode `finalize` pour l'effacement sécurisé est discutable, car elle suppose que l'application puisse forcer l'action du *garbage collector*. En toute rigueur, ceci n'est pas possible pour un environnement d'exécution Java standard. Toutefois, cet avis est à tempérer car, à défaut de pouvoir utiliser des mécanismes de bas niveau pour effacer des zones mémoires, il s'agit d'un début de solution.

6.4.6.3 Etude de l'implémentation des composants sélectionnés

Etude de sun.security.provider.SecureRandom L'accès aux spécifications du générateur d'aléa est réalisé via une revue du code source. Le commentaire fourni en annexe 10.5 issu de la classe `sun.security.provider.SecureRandom`[48] du CSP SUN montre que le générateur d'aléa spécifié par Sun ne fait référence à aucun standard connu. De plus, le même commentaire indique que cette implémentation est prévue pour un service de génération d'aléa. Or, les extraits du code source fournis en annexe font référence à un générateur de pseudo-aléa.

48. `openjdk_source/jdk/src/share/classes/sun/security/provider/SecureRandom.java`

Le générateur d'aléa de Sun est un automate déterministe composé d'un état interne S_t et d'une sortie R_t ayant chacun une taille de 20 octets liés par les relations suivantes :

- l'état interne S_t évolue selon une transition non-linéaire du type $S_{t+1} = F(S_t, R_t)$;
- la sortie R_t est liée à l'état interne par une relation non linéaire du type $R_t = G(S_t)$.

La description pseudo-formelle de l'algorithme est présentée en annexe 10.4.

L'état interne du générateur est initialisé par une graine `seed` dont on ne maîtrise ni l'entropie ni l'origine. Dans l'esprit, l'architecture du générateur d'aléa ne respecte donc pas les recommandations de l'ANSSI, à savoir :

- absence de spécification d'une source d'aléa physique ;
- pas de retraitement algorithmique de niveau standard.

L'évaluation des propriétés cryptographiques de la fonction de transition de l'état interne ainsi que la qualification de l'aléa généré par la classe `SecureRandom` sortirait du cadre de ce projet. Une analyse en boîte noire sur un fichier de 500 Mo via un test d'entropie a cependant montré l'existence de biais statistiques sur les suites générées par l'automate [49] à partir de l'ordre 15.

Concernant l'implémentation du service, nous avons constaté que la fonction de hachage SHA-1 est utilisée via JCA. Pour cela, une classe de haut niveau `MessageDigest` est utilisée :

```
/**
* This call, used by the constructors, instantiates the SHA digest
* and sets the seed, if given.
*/
private void init(byte[] seed) {
  try {
    digest = MessageDigest.getInstance ("SHA");
  }
  catch (NoSuchAlgorithmException e) {
    throw new InternalError("internal error: SHA-1 not available.");
  }
  if (seed != null) {
    engineSetSeed(seed);
  }
}
```

Nous pouvons constater que le service n'impose pas le CSP à utiliser. L'algorithme SHA-1 effectivement utilisé correspond donc à la première implémentation trouvée dans la liste des services des CSP. Cette manière de procéder est très discutable surtout lorsque le CSP auquel appartient ce générateur dispose de son propre service SHA1. En effet, le risque de piégeage est très élevé. Si un attaquant modifie les fichiers de configuration de Java afin d'insérer un CSP piégé en tête de liste, la qualité de l'aléa ne sera pas garantie.

49. En première approche, les propriétés statistiques de la fonction de hachage devraient assurer une décorrélation statistique des sorties du générateur d'aléa.

L'étude de cette brique permet de soulever deux points essentiels :

- les fonctions existantes ne sont pas forcément sûres. Il est fort possible que certains services soient à réimplémenter ;
- les points (*a minima*) à améliorer de ce générateur sont les suivants :
  - un algorithme de retraitement pseudo-aléatoire de niveau standard doit être utilisé,
  - actuellement le générateur utilise une source physique d'aléa uniquement au moment de l'initialisation. Ensuite, le générateur fonctionne en circuit fermé. De l'aléa provenant d'une source physique devrait être injecté continuellement,
  - il est recommandé d'employer de la mémoire non volatile comme accumulateur,
  - l'état interne du générateur d'aléa doit être verrouillé en mémoire, afin d'éviter des fuites d'information via le fichier d'échange du système d'exploitation ;
- lorsqu'un service en appelle un autre, il convient de s'assurer que le service appelé est conforme à ce qu'il est censé réaliser. Pour cela, certaines mesures élémentaires peuvent être prises :
  - protéger la liste des CSP dans la configuration de Java (cas où l'appel à un service se fait sans préciser le CSP),
  - utiliser les services existants dans le CSP utilisé,
  - embarquer des vecteurs de tests afin de vérifier la conformité des services appelés.

Etude de sun.security.mscapi.PRNG Ce générateur d'aléa n'est disponible que sous Windows. Il s'appuie largement sur la CryptoAPI de Windows. Le générateur est développé en code natif (C++). La partie codée en Java ne réalise aucune opération cryptographique. Les détails de l'implémentation du générateur sont présentés en annexe 10.6. La sortie du générateur de la CryptoAPI de Windows est directement retrournée à l'application sans retraitement algorithmique.

Considérant l'absence d'informations sur l'architecture et l'implémentation du générateur d'aléa de Windows, nous nous sommes focalisés sur l'évaluation en boite noire du générateur. Les tests ont montré que la sortie du générateur présente un biais statistique équivalent au générateur précédent [50].

L'étude de cette brique souligne à nouveau le fait que les services déjà présents ne sont pas nécessairement de confiance et qu'ils doivent être réimplémentés. Enfin, cette brique est intéressante, car elle fait intervenir du code natif. Cela implique qu'une bibliothèque partagée (`sunmsapi.dll`) est chargée et que la fonction suivante est exportée : `Java_sun_security_mscapi_PRNG_generateSeed`.

Le mécanisme de DLL peut faciliter l'utilisation de techniques d'API Hooking par un attaquant.

50. L'aléa a été généré sur Windows Vista 32 bits.

Il est donc possible d'intercepter des informations sensibles provenant de services de sécurité implémentés en code natif.

Etude de sun.security.provider.DSAKeyPairGenerator Cette classe constitue l'implémentation du générateur de bi-clés DSA du CSP SUN.

Dans la suite du paragraphe, les lettres p, q, g, y, x désignent respectivement les différents paramètres des bi-clés DSA :

- le nombre premier ;
- le nombre premier divisant $p-1$;
- le générateur du groupe ;
- la clé publique ;
- la clé privée.

L'étude a été réalisée via des tests fonctionnels qui ont permis de soulever les points suivants :

- par défaut, la taille des clés est de 1024 bits ;
- si la méthode `initialize(int keysize[, SecureRandom random])` (`keysize` étant égale à 512, 768 ou 1024) est utilisée pour initaliser le générateur, alors les paramètres p, q, g sont des valeurs précalculées ;
- si la méthode `initialize(int keysize[, SecureRandom random])` avec (`keysize` étant différente de 512, 768 ou 1024) est utilisée pour initaliser le générateur, alors les paramètres p, q et g sont générés par l'intermédiaire de la classe `DSAParameterGenerator` ;
- il est possible d'initialiser le générateur en lui fournissant manuellement les paramètres p, q et g. Toutefois, aucune vérification sur la robustesse des paramètres n'est réalisée ;
- par défaut, si aucun générateur d'aléa n'est précisé lors de l'initialisation du générateur de bi-clés, alors le générateur d'aléa utilisé est le premier trouvé dans la liste du premier CSP.

Cette implémentation du générateur de bi-clés DSA montre qu'une mauvaise connaissance du service en question peut entrainer facilement une mauvaise utilisation. En effet, la génération d'une clé DSA 1024 réalisée de la manière suivante :

```
KeyPairGenerator gen;
KeyPair keys;

gen = KeyPairGenerator.getInstance("DSA", "SUN");

gen.initialize(1024, new SecureRandom());
//Facultatif, il s'agit du choix par defaut
```

```
keys = gen.generateKeyPair();
```

peut potentiellement entraîner la génération d'une clé faible, du fait que les paramètres p, g et q sont des constantes précalculées (`sun.security.provider.ParameterCache`) comme le montre le code source fournit en annexe 10.7 qui force la génération des paramètres p, g et q pour une taille de 1024 bits. Une analyse plus approfondie du processus de génération serait nécessaire pour valider la génération des bi-clés.

De plus, le fait que l'algorithme SHA1PRNG du CSP SUN soit l'algorithme de génération d'aléa par défaut pour un environnement Java classique nous semble préjudiciable en vertu de l'analyse effectuée précédemment. En effet, la lecture du code source de cette classe a montré que la taille des clés générées est comprise entre 512 et 1024 bits, ce qui est contraire au référentiel de l'ANSSI relatif à l'algorithme DSA.

6.4.6.4 Conclusion

Cette étude a permis d'analyser plusieurs implémentations des services `SecureRandom`, `KeyPairGenerator` et des structures de données `Keys`. Les résultats de cette étude concernent les cinq points suivants :

- effacement des données ;
- utilisation des mécanismes existants ;
- confiance dans les mécanismes existants ;
- utilisation du code natif ;
- utilisation de services dans d'autres services.

L'étude a montré que l'effacement des clés n'est quasiment pas utilisé. En effet, seules quelques classes l'implémentent par le biais de la méthode `finalize`. Dans tous les cas, l'effacement des données dépend fortement du *garbage collector*. Cela était prévisible, étant donné que Java ne met pas à disposition de mécanismes de gestion avancée de la mémoire. Plusieurs solutions peuvent être envisagées afin de résoudre ce problème :

- une modification de la JVM (fonctions d'effacement bas niveau de la mémoire, modification du comportement du *garbage collector*, etc.) ;
- utilisation de code natif pour réaliser ces opérations.

Concernant l'utilisation et la confiance des fonctions cryptographiques, nous avons pu observer qu'une bonne connaissance des services est nécessaire. Par exemple, l'utilisation du générateur de clés 1024 bits DSA avec le CSP SUN peut conduire à la création de clés où les paramètres p, q et g sont constants ce qui réduit fortement la diversification des clés. L'exemple de code Java montrant cette mauvaise utilisation, présenté dans ce document, est élémentaire et intuitif dans le sens où il y a de fortes chances que cette erreur soit reproduite par d'autres

développeurs. Il est donc évident qu'une bonne maîtrise des services existants est nécessaire pour pouvoir les utiliser correctement.

L'étude a montré qu'il fallait être vigilant quant à la qualité des services déjà implémentés. En effet, l'étude sur les générateurs d'aléa a mis en évidence la mauvaise qualité de l'aléa généré. Ces résultats appellent à se questionner sur la confiance que l'on peut avoir sur des services aussi sensibles.

L'utilisation du code natif est intéressante, car celui-ci permet de réaliser des opérations que ne permet pas la JVM : verrouillage des pages mémoires, effacement de zones mémoires, appels à des bibliothèques dynamiques externes, etc. Toutefois, son principal inconvénient réside dans la perte des propriétés de sécurité assurées par la plate-forme Java. En effet, le code natif est généralement développé dans un langage tel que le C ou le C++. Il est donc potentiellement sujets aux vulnérabilités caractéristiques de l'utilisation de ces langages (notamment les dépassements de tampon). De plus, l'utilisation de fonctions natives est facilement décelable et peut devenir la cible privilégiée d'un attaquant pour une opération de piégeage. En effet, les méthodes et les attributs de classe traduits par du code natif sont généralement exportés avec des noms extrêmement intelligible quant à leurs rôles. Ainsi un attaquant peut piéger une application à l'aide des techniques d'API hooking. Toutefois, la mise en oeuvre d'une telle attaque demande un contexte d'utilisation et des conditions particulières qui doivent être étudiés.

Enfin, l'étude nous a permis d'observer la manière dont est réalisée l'utilisation d'un service dans un autre (par exemple, l'utilisation du générateur d'aléa dans un générateur de clés). Nous avons constaté qu'un service donné dans un CSP donné ne fait jamais explicitement appel à un autre service déjà implémenté dans son propre CSP. À chaque fois, le service par défaut est utilisé. Le service par défaut est le premier service trouvé (correspondant au besoin) dans le premier CSP. L'ordre de préférence des CSP est déterminé dans les fichiers de configuration de la sécurité de Java. Cette manière de réaliser les appels entre services conduit à un possible vecteur d'attaque :

- insertion d'une archive JAR contenant un CSP piégé ;
- modification de l'ordre de préférence des CSP dans le fichier de configuration afin de définir le CSP piégé en tête de liste.

Afin de se prémunir contre cette menace, il convient de mettre en œuvre un certain nombre de bonnes pratiques :

- protéger les fichiers de configuration de Java ;
- toujours expliciter le nom du CSP que l'on souhaite utiliser.

7 BIBLIOTHÈQUES D'EXTENSIONS RELATIVES À LA SÉCURITÉ

La bibliothèque standard de Java définie par Sun propose différentes API et *frameworks* pour sécuriser des implémentations. Cependant, d'autres organismes proposent leur propres implémentations d'API de sécurité qui peuvent être utilisées en sus ou à la place des classes de la bibliothèque standard.

7.1 Présentation de la démarche et de la métrique

Dans un premier temps, les principales API de sécurité sont identifiées. Elles sont ensuite comparées suivant une métrique qui permet de sélectionner les extensions pertinentes d'un point de vue sécurité et qui semblent les plus abouties d'un point de vue fonctionnel.

Les extensions de sécurité identifiées sont les suivantes :

- les API cryptographiques :
  - Bouncy Castle Crypto API ;
  - Oracle Security Developer Tools ;
  - FlexiProvider ;
  - BSAFE - RSA ;
  - Cryptix ;
  - SIC - IAIK-JCE ;
  - Forge ;
  - Jasypt ;
- les API liées au TPM :
  - jTPM ;
  - TPMj.

La comparaison de ces API repose sur la spécification de critères. Ceux-ci mettent en évidence si l'API est tenue à jour, la nationalité de son éditeur, son coût, la disponibilité des sources, sa facilité d'intégration et d'utilisation, sa popularité (si elle est reconnue dans son domaine), son évolutivité et ses performances. Ces critères sont présentés dans le tableau 9.

Critère	Echelle, valeur	Remarques éventuelles
Produit	Nom du produit	
Editeur	Nom de l'éditeur	
Origine	Nationalité de l'éditeur	
Confiance en l'éditeur	Basse, modérée, haute	Haute si la société ou le groupe est transparent vis-à-vis de ses contacts et de ses activités. Modérée si la société ou le groupe n'est pas totalement transparent vis-à-vis de ses contacts et de ses activités. Basse s'il y a peu d'informations sur le groupe ou la société, ou bien si la société ou le groupe n'est pas du tout transparent vis-à-vis de ses contacts et de ses activités.
Sources	Licence *open source*, licence propriétaire	Disponibilité des sources
Niveau de fonctionnalités	Bas, Modéré, Haut	
Popularité	Faible, Moyenne, Grande	Popularité de l'API dans le monde et dans son domaine.
Mise à jour	Faible, Peu fréquente, Fréquente	Indice de fréquence de mises à jour de l'API (faible < 2 ans, peu fréquente < 1 an, fréquente < 6 mois)
Date	Date de la dernière version	
Intégration	Facile, Modérée, Difficile	
Coût	Coût de l'API	
Performances	Faibles, Moyennes, Grandes	Performances globales de l'API suivant les retours d'expérience.

TABLE 9: Critères retenus pour la comparaison des API

7.2 Analyse des extensions cryptographiques

7.2.1 Bouncy Castle Crypto API

L'API Bouncy Castle constitue une implémentation libre en Java d'algorithmes cryptographiques. Cette API est conforme aux différents standards cryptographiques en vigueur (FIPS notamment). Elle inclut les composants suivants :

- une API de cryptographie en Java (chiffrement / déchiffrement, signature, génération de nombre pseudo-aléatoire) ;
- un *provider* pour JCA ;
- une bibliothèque pour la lecture et l'écriture des objets encodés en ASN.1 ;
- une API TLS côté client ;
- des générateurs pour les versions 1 et 3 des certificats X.509 ;
- des générateurs de liste de révocation de certificat (CRL) pour la version 2 des certificats X.509 ;
- des générateurs de fichiers PKCS#12 ;
- des générateurs pour la version 2 des attributs de certificats X.509 ;
- des générateurs/processeurs pour S/MIME et CMS (PKCS#7) ;
- des générateurs/processeurs pour OCSP (RFC 2560) ;
- des générateurs/processeurs pour TSP (RFC 3161) ;
- des générateurs/processeurs pour OpenPGP (RFC 2440).

L'API est compatible avec l'implémentation de Sun de la bibliothèque Java SE (de la version 1.1 à 1.6). Elle est séparée en sept packages distincts :

- *JCE Utility and Extension Packages* (boite à outils pour JCE) ;
- *OCSP and OpenSSL PEM Support Packages* (support de l'OCSP et de OpenSSL) ;
- *ASN.1 Support Packages* (support du langage ASN.1) ;
- *Lightweight Crypto Packages* (algorithmes cryptographiques) ;
- *Utility Packages* (boite à outils) ;
- *JCE Provider and Test Classes* (Provider JCE) ;
- *Other Packages* (géneration de certificat X.509).

Le détail des classes des packages est fourni en annexe 10.8.

Critère	Echelle, valeur	Remarques
Produit	Bouncy Castle Crypto APIs	
Editeur	The Legion of the Bouncy Castle	Organisation à but non lucratif.
Origine	Internationale	Différents contributeurs de différentes nationalités.
Confiance en l'éditeur	Haute	Le projet est open source et multi-développeurs.
Sources	open source (MIT X11 adaptée).	
Niveau de fonctionalités	Haut	
Popularité	Grande	
Mise à jour	Fréquente	
Date	18/04/2009	
Intégration	Facile	Une documentation (Javadoc) détaillée est disponible.
Coût	Gratuit	
Performances	Grande	L'implémentation se veut « légère ».

7.2.2 Oracle Security Developer Tools

Oracle Security Developer Tools (OSDT) sont des outils de sécurité pour le langage Java développés par la société Oracle. Le synoptique ci-dessous présente l'architecture et les dépendances des API les unes par rapport aux autres.

7.2.2.1 Liste et présentations des outils de sécurité disponibles de Oracle Security Developer Tools

Oracle Crypto : API implémentant des algorithmes dédiés à la cryptographie ;

Cryptoki : (n'est plus utilisée car présent dans l'actuel JCE de Sun) ;

Provider JCE Oracle : provider pour le JCE de Sun des mécanimes cryptographiques (Oracle Crypto) ;

Cryptographic Message Syntax (CMS) : API définissant une protection de don-

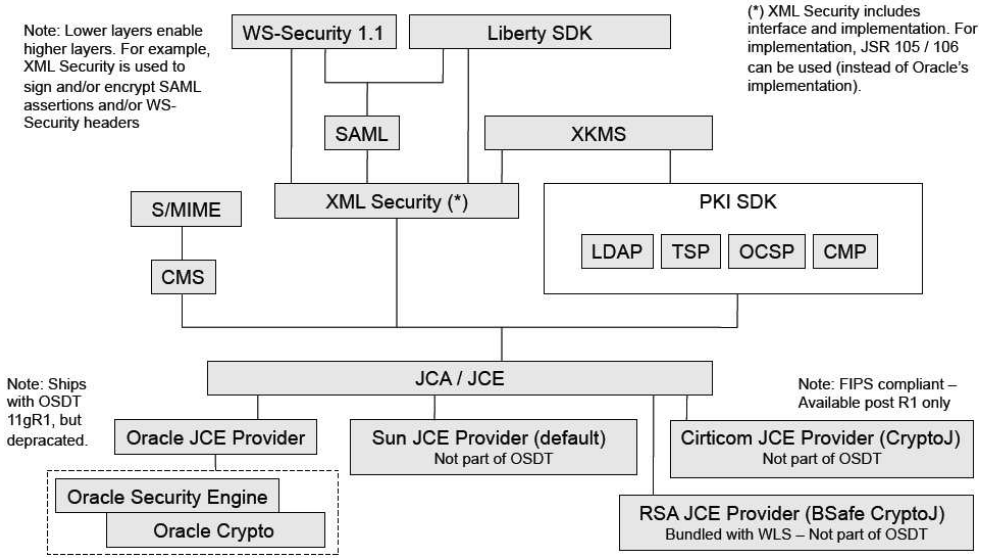


FIGURE 13 – Architecture OSDT version 11g - 2009 (source Oracle).

nées ;

S/MIME (Secure / Multipurpose Internet Mail Extensions : Support de S/MIME v2, v3 et v3.1 ;

Public Key Infrastructure (PKI) : support de PKI (PKCS#11) ;

XML Security (XML Signature et XML Encryption) : API java pour la signature et le chiffrement de fichiers XML ;

SAML (Security Assertion Markup Language) : API java apportant le support de SAML ;

XKMS (XML Management Specification) : permet la gestion de clés (distribution / enregistrement) ;

WS-Security 1.0 : API pour les documents XML (support certificat X.509, tickets SAML et Kerberos) ;

Liberty Alliance : API Java permettant le developpement de Single Sign-on (SSO) et de FIM (Federated Identity Management).

7.2.2.2 Précision sur l'API Oracle Crypto

Oracle Crypto est une API Java de sécurité. Elle propose des algorithmes pour la génération de clés, de chiffrement/déchiffrement, de signature, de génération de nombre pseudo-aléatoire, etc. L'API est certifiée FIPS 140-2 Level 1. Elle est developpée entièrement en Java et ne contient aucune méthode native. L'API est distribuée sous la forme d'une archive `jar`. Un *provider* de cette API est disponible. Elle est compatible avec JCE depuis la version 1.2.1 de cette extension de la bibliothèque standard.

Les algorithmes supportés et implémentés sont les suivants :

- RSA, DSA et EC ;
- RC4 ;
- AES, DES et Triple-DES ;
- MD5 et SHA-1 ;
- Diffie-Hellman ;
- Générateur de nombre pseudo-aléatoire (PRNG).

Critère	Echelle, valeur	Remarques
Produit	API Oracle Crypto	
Editeur	Oracle	
Origine	Americaine	
Confiance en l'éditeur	Modérée	Les sources ne sont pas disponibles, seule la notoriété de l'éditeur peut être gage de confiance.
Sources	Sous licence propriétaire	
Niveau de Fonctionalités	Haut	
Popularité	Grande	
Mise à jour	Peu fréquente	
Date	Premier trimestre 2009	
Intégration	Modérée	Des techniciens peuvent accompagner les développeurs dans l'intégration.
Coût	Prix non communiqué	
Performances	Aucun retour.	

7.2.3 FlexiProvider

FlexiProvider est une API de sécurité Java et un *provider* pour le JCA/JCE de Sun Java developpée par le Theoretical Computer Science Research Group dirigé par le professeur Johannes Buchmann de l'université des sciences de Daemstadt (Allemagne).

Elle implémente les algorithmes cryptographiques les plus répandus et dispose en fait de quatre *providers* différents open source (CoreProvider, ECProvider, PQCProvider, NFProvider). Le CoreProvider est sous licence LGPL, le ECProvider, PQCProvider et NFProvider sont eux

sous licence GPL.

Description des *providers* :

CoreProvider le CoreProvider contient les algorithmes cryptographiques publiques les plus connus, tels que RSA, AES, 3-DES, chiffrement de mot-de-passe selon PKCS#5, MD5, SHA1, RIPEMD, CBC-MAC, CMAC, HMAC et un générateur de nombre pseudo aléatoire (BBS) ;

ECProvider le ECProvider contient les algorithmes cryptographiques dédiés aux courbes elliptiques, tels que ECDSA (Elliptic Curve Digital Signature Algorithm), ECNR, ECDH (Elliptic Curve Diffie-Hellman), ECIES (Elliptic Curve Integrated Encryption Scheme) ;

PQCProvider le PQCProvider contient des algorithmes cryptographiques quantiques expérimentaux, non standardisés, tels que CMSS et CFS (signature), *McEliece cryptosystem* et quatre de ses variantes, *Niederreiter cryptosystem* ;

NFProvider le NFProvider est fourni à titre expérimental. Il a été developpé pour prouver la viabilité des nombres finis dans un contexte quadratique imaginaire (IQ).

FlexiProvider est compatible avec Sun Java version 1.3.1_18, 1.4.2_12, 1.5.0_10 et 1.6.0_03. Cette API est disponible à l'adresse suivante : `http://www.cdc.informatik.tu-darmstadt.de/flexiprovider`.

Critère	Echelle, valeur	Remarques
Produit	FlexiProvider	
Editeur	Université de Darmstadt	
Origine	Allemand	
Confiance en l'éditeur	Haute	Les sources sont disponibles et les auteurs sont des chercheurs du domaine.
Sources	open source	LGPL et GPL.
Niveau de Fonctionalités	Modéré	
Popularité	Faible	
Mise à jour	Peu fréquente	
Date	Non communiquée	La date de la dernière version (1.6p5) n'est pas indiquée, mais la dernière version majeure (1.6) date de mai 2008.
Intégration	Inconnue	
Coût	Gratuit	
Performances	Aucun retour	

7.2.4 BSAFE - RSA

La société RSA propose une API de sécurité pour le langage Java. Cette API, BSAFE, est implémentée entièrement en Java. Elle est disponible à l'adresse `http://www.rsa.com/node.aspx?id=1319`.

Cette API propose les algorithmes cryptographiques les plus connus, des services pour sécuriser les communications et des mécanismes de sécurité pour les applications Web avec un support de WS-Security. Les algorithmes supportés et implémentés sont les suivants :

- RSA, DSA et Diffie-Hellman ;
- AES, RC5, RC4, RC2, DES, 3DES and DESX ;
- MD2, MD5, HMAC, SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512 ;
- RIPEMD-160 ;
- Elliptic Curve Digital Signature Algorithm (ECDSA), Elliptic Curve Diffie-Hellman (ECDH), ECDH with co-factor (ECDHC), Elliptic Curve Authenticated Encryption Scheme (ECAES), Elliptic Curve Integrated Encryption Scheme (ECIES).

Les principaux standards supportés sont les suivants :

- FIPS 140 ;
- Protocoles SSL v2, SSL v3, et TLS v1 - pour les communications sécurisées ;
- American National Standards Institute (ANSI) - X9.31 ;
- RIPEMD-160 ;
- PKCS #1, 5, 8, 11, and 12 - pour la Cryptographie ;
- PKCS #1, 3, 5, 7, 8, 10, 11 and 12 - pour la gestion de PKI ;
- PKCS #1, 7, 8, 10, 11, and 12 - pour les communications sécurisées ;
- Certificate format - X.509 v3 - pour les PKI et les communications sécurisées ;
- LDAP directory - v2 - pour les PKIs.

Critère	Echelle, valeur	Remarques
Produit	BSAFE	
Editeur	RSA	
Origine	Américain	
Confiance en l'éditeur	Modérée	Les sources ne sont pas disponibles. Même si RSA est très populaire, il peut subsister un doute concernant la qualité des produits à l'export.
Sources	Sous licence propriétaire	

Niveau de fonctionnalité	Haut	
Popularité	Grande	Due à la notoriété de RSA.
Mise à jour	Peu fréquente	
Date	Année 2008	
Intégration	Inconnue	
Coût	Non communiqué	
Performances	Aucun retour	

7.2.5 Cryptix

Le projet Cryptix est une implémentation d'un *provider* de fonctions cryptographiques pour le JCA/JCE de Sun disponible à l'adresse `http://www.cryptix.org/`. La dernière version de l'API date de mars 2005.

Critère	Echelle, valeur	Remarques
Produit	Cryptix	
Editeur	Cryptix	
Origine	Américain	
Confiance en l'éditeur	Modérée	Les sources sont disponibles
Sources	Sous licence (Cryptix General License), mais disponible	
Niveau de fonctionnalité	Bas	
Popularité	Faible	
Mise à jour	Faible	
Date	Dernière version de Mars 2005	
Intégration	Inconnue	
Coût	Gratuit	
Performances	Aucun retour	

7.2.6 SIC - IAIK-JCE

IAIK-JCE est un *provider* JCA/JCE de Sun. Cette bibliothèque a été évaluée EAL+3 par le laboratoire autrichien TÜV IT. L'API est compatible avec les versions Java 1.1 ou ultérieure. Elle implémente toutes les fonctions cryptographiques fournies par le JCE/JCA de Sun Java (les algorithmes sont présentés ci-dessous). Une API dédiée à la gestion de PKI est également présente.

Hachage

- SHA-1, SHA-256, SHA-384, SHA-512 et RIPEMD-160.

Générérateur de nombres aléatoires

- basé sur SHA-1, SHA-256, SHA-384, SHA-512 et RIPEMD-160 ;
- FIPS 186-2.

Signature

- RSA selon la version 1.5 de PKCS#1 (basé sur SHA-1, SHA-256, SHA-384, SHA-512 et RIPEMD-160).
- RSA PSS selon la version 2.1 de PKCS#1 (basé sur SHA-1, SHA-256, SHA-384, SHA-512 et RIPEMD-160).

Chiffrement

- AES, Triple-DES, RC2, ARCFOUR, RSA et RSA OAEP .

MAC

- basé sur SHA-1, SHA-256, SHA-384, SHA-512 et RIPEMD-160.

Liens internet :
`http://jce.iaik.tugraz.at/sic/media/product_pdfs/iaik_jce`.
`http://jce.iaik.tugraz.at/sic/products/core_crypto_toolkits/jce_cc_core`.

Critère	Echelle, valeur	Remarques
Produit	IAIK-JCE	
Editeur	TÜV IT	
Origine	Autrichien	
Confiance en l'éditeur	Haute	Le produit a été évalué EAL 3+.

Sources	Sous licence propriétaire	
Niveau de fonctionnalité	Haut	
Popularité	Moyenne	
Mise à jour	Fréquente	
Date	décembre 2008	
Intégration	Inconnue	
Coût	Non communiqué	
Performances	Aucun retour	

7.2.7 Proteckt - Forge

Proteckt, de la société Forge, est une API de sécurité entièrement développée en Java qui implémente entièrement TLS version 1.0 et assure une compatibilité avec SSL v3.0. L'API peut être utilisée avec le JCA 1.2 de Sun. La dernière version est la v3.0 datant de Mai 2000. Aucune nouvelle annonce n'a été faite récemment. Une version d'évaluation de 30 jours est téléchargeable depuis le site après un enregistrement. Aucun prix n'est néanmoins mentionné, il faut contacter le service commercial.

Les algorithmes supportés sont les suivants :

- RSA, DSA et Diffie-Hellman avec des clés de 512, 1024, 2048 bits et plus ;
- RC4 et IDEA 128 bits ;
- Triple-DES ;
- DES ;
- RC ;
- un générateur de nombres aléatoires.

Cette API est disponible à l'adresse : `http://www.forge.com.au/Research/products/Protekt/protekt.html`.

Critère	Echelle, valeur	Remarques
Produit	Proteckt	
Editeur	Forge	
Origine	Australien	

Confiance en l'éditeur	Basse	Les sources ne sont pas disponibles et la société n'est pas connue.
Sources	Sous licence propriétaire	
Niveau de fonctionnalité	Modéré	
Popularité	Faible	
Mise à jour	Faible	
Date	mai 2000	
Intégration	Inconnue	
Coût	Non communiqué	
Performances	Aucun retour	

7.2.8 Jasypt

Jasypt est un projet open source qui permet de simplifier l'utilisation des méthodes des fonctions cryptographiques de Java en ajoutant une couche d'abstraction supplémentaire à JCA/JCE. L'API est facilement utilisable avec les *providers* JCE/JCA de Sun ou d'autres éditeurs, tels que Bouncy Castle. En revanche, elle n'implémente aucun algorithme cryptographique.

Cette API est disponible à l'adresse : `http://www.jasypt.org/features.html`.

Critère	Echelle, valeur	Remarques
Produit	Jasypt	
Editeur	Jasypt	Communauté de développeurs.
Origine	Américain	
Confiance en l'éditeur	Haute	Jasypt a une bonne notoriété et ses sources sont disponibles.
Sources	open source	
Niveau de fonctionnalité	Bas	
Popularité	Moyenne	
Mise à jour	Peu fréquente	
Date	Dernière version de juin 2008	

Intégration	Inconnue	
Coût	Gratuit	
Performances	Aucun retour	

7.3 Analyse des extensions de sécurité dédiées à la gestion d'un TPM

7.3.1 TPM/j

TPM/j est une API Java offrant des fonctionnalités pour utiliser un TPM. Elle est développée par une équipe du MIT. Toutefois, elle s'addresse plus à des usages expérimentaux et n'implémente pas de façon complète les spécifications de la *TPM Software Stack* (TSS). Elle est compatible depuis la version 1.5 du JDK de Sun Java et fonctionne avec des puces TPM implémentant au moins la version 1.1b des spécifications. Pour pouvoir utiliser cette API, le TPM doit être obligatoirement activé depuis le BIOS. Cette API est disponible à l'adresse : `http://projects.csail.mit.edu/tc/tpmj/`.

Critère	Echelle, valeur	Remarques
Produit	TPM/J	
Editeur	Groupe MIT	Intégrant le Trusted Computing Group.
Origine	Américain	
Confiance en l'éditeur	Haute	Les sources sont disponibles et les développeurs du MIT sont reconnus.
Sources	open source	
Niveau de fonctionnalité	Modéré	
Popularité	Moyenne	
Mise à jour	Peu fréquente	
Date	3 avril 2007	
Intégration	Difficile	Paramétrage parfois difficile et compatibilité incomplète avec les spécifications du TCG.
Coût	Gratuit	
Performances	Aucun retour	

7.3.2 jTPM

jTPM ou encore *Trusted Computing for the Java Platform* est une API Java offrant des fonctionnalités pour utiliser un TPM. Elle offre des fonctionnalités similaires à TPM/j, mais semble plus aboutie. Elle peut être considérée comme une alternative à Trousers, l'implémentation de la TSS en C. Cette API est compatible depuis la version 1.5 du JDK. Elle est toutefois considérée comme expérimentale pour les versions actuelles. Le TPM doit être activé depuis le BIOS.

Les puces TPM compatibles sont les suivantes :

- Infineon 1.2 TPM ;
- TPM Emulator from ETH Zurich (logiciel) ;
- Atmel 1.2 TPM ;
- Infineon 1.1b TPM ;
- Broadcom 1.2 TPM ;
- ST Microelectronics 1.2 TPM ;
- Atmel 1.1 TPM (limité).

Deux API principales sont définies dans jTPM : jTSS et jTpmTools.

7.3.2.1 Spécification de l'API jTpmTools

L'API jTpmTools permet notamment de :

- gérer le propriétaire du TPM ;
- gérer les registres PCRs ;
- utiliser des fonctions sur les clés EK (Endorsement Key) ;
- utiliser des fonctions pour la gestion de la PKI ;
- gérer les clés ;
- utiliser des fonctions de chiffrement/déchiffrement, scellement/déscellement de données.

7.3.2.2 Spécification de l'API jTSS

L'API jTSS est découpée également en deux sous parties, la TCS (*TSS Core Services*) qui permet de centraliser toutes les requêtes vers le TPM et la TSP (*TSS Service Provider*) qui permet de fournir les fonctions aux applications qui veulent se servir du TPM.

Lien internet : `http://trustedjava.sourceforge.net/`

Critère	Echelle, valeur	Remarques
Produit	jTPM	
Editeur	Open Trusted Computing	
Origine	Américain	
Confiance en l'éditeur	Haute	
Sources	open source	
Niveau de fonctionnalité	Modéré	
Popularité	Moyenne	
Mise à jour	Fréquente	
Date	2 mars 2009	Les différents outils ne sont pas mis à jour forcement en même temps.
Intégration	Difficile	Paramétrage et compatibilité non aisés.
Coût	Gratuit	
Performances	Aucun retour	

7.4 Sélection des extensions pertinentes

Au vu des critères définis précédemment, les API qui paraissent les plus pertinentes en termes d'apport pour la sécurité sont les suivantes :

- API Bouncy Castle Crypto ;
- API FlexiProvider.

API cryptographiques					
Nom	**Niveau de fonctionalité**	**Editeur**	**Confiance dans l'éditeur**	**Prix**	**Version**
Bouncy Castle Crypto	Haut	Groupe	Haute	open source	v1.41 (02/10/2008)
Oracle Security Developer Tool	Haut	Oracle	Modérée	Gratuit, source non disponible	version 11g (2009)
FlexiProvider	Moyen	Université	Haute	open source	v1.6p3 (2008)
BSAFE	Haut	RSA	Modérée	Payant, prix non communiqué	non communiqué
Cryptix	Bas	Groupe	Modérée	open source	Version de 03/2005
IAIK-JCE	Haut	SIC	Modérée	Payant, prix non communiqué	IAIK-JCE 3.17 (23/12/2008)
Protekt	Modéré	Forge	Modérée	Payant, prix non communiqué	v3.0 (05/2000)
Jasypt	Bas	Groupe	Modérée	open source	v1.5 (06/2008)
API dédiées à la gestion du TPM					
TPMj	Modéré	MIT	Modérée	open source	v0.3 (04/2007)
jTPM	Modéré	Institut IAIK (Autriche)	Modérée	open source	v0.4 (18/12/2008)

TABLE 20 – Récapitulatif des API Java de sécurité

8 APPORT DES MÉTHODES FORMELLES

Le terme « méthode formelle » désigne un ensemble de techniques mathématiques rigoureuses et d'outils pour la spécification, la construction et la vérification de systèmes logiciels et matériels. Le caractère rigoureux signifie ici que les spécifications utilisées dans ces approches, sont réalisées dans des langages logiques bien précis (contrairement au langage naturel), et que les vérifications réalisées sont des déductions rigoureuses dans cette logique (chaque étape du raisonnement suit un procédé mécanique bien identifié). L'intérêt de ces approches est de permettre une analyse symbolique prenant en compte l'ensemble des espaces d'état d'un système (par exemple, l'ensemble des exécutions possibles d'un programme) et d'établir ainsi une preuve de correction de propriétés valable pour tous les comportements possibles du système.

Une telle approche peut paraître séduisante en première approche, mais cache cependant deux grandes difficultés intrinsèques qu'il ne faut pas ignorer. La première concerne la complexité des systèmes riches qui freine énormément le déploiement des méthodes formelles. Les outils de méthodes formelles sont des objets complexes, difficiles à concevoir et souvent aussi à manipuler. Cette difficulté s'intensifie lorsque l'on aborde des systèmes riches comme le langage Java accompagné de tout son environnement standard. La deuxième, certainement la plus fondamentale, provient des impossibilités théoriques fortes qui pèsent sur le domaine. La théorie de la calculabilité nous apprend en effet que la preuve automatique de programmes par un autre programme informatique est un objectif impossible à atteindre, en toute généralité : aucun programme n'est capable de vérifier une propriété non triviale pour tous les programmes d'un langage de programmation aussi expressif que Java. En clair, l'outil automatique qui dira pour n'importe quel programme s'il est correct ou non, n'existe pas. Il existe donc des impossibilités théoriques fortes concernant la vérification automatique de programmes. Pour autant, cela n'a pas empêché l'émergence d'outils de vérification mécanisée extrêmement puissants capables de détecter les erreurs ou prouver leur absence sur certains logiciels complexes.

Différentes familles de techniques de vérification formelle ont été proposées. On peut les distinguer vis-à-vis de la stratégie qu'elles emploient pour surmonter les points durs évoqués précédemment.

8.1 La vérification déductive

Puisque la vérification complètement automatique est impossible, l'idée proposée par la vérification déductive est d'employer une interaction humaine durant le calcul de l'outil de vérification. C'est par exemple le cas de *la preuve assistée par ordinateur* où une propriété sur le comportement d'un programme est alors prouvée de façon interactive entre l'ordinateur et l'utilisateur. L'ordinateur se charge de vérifier la correction des étapes de raisonnement, et éventuellement d'effectuer certaines parties de la preuve automatiquement. Dans le contexte de la vérification de programme de type Java, cela se traduit par un environnement de vérifications

déductives généralement constitué des éléments suivants :

- Un **langage de spécification** pour écrire sous forme de formule logique le comportement attendu d'un programme. L'exemple le plus connu dans ce domaine est le langage de spécification JML. L'utilisateur doit annoter le programme avec des pré-conditions, post-conditions et des invariants de boucle.
- Un moteur symbolique de génération d'**obligations de preuve**, qui génère à partir des annotations précédentes un ensemble d'énoncés mathématiques dont la validité (au sens *être démontrable*) assure la correction globale du programme vis-à-vis des annotations insérées par l'utilisateur.
- Un **moteur de preuve** pour prouver les obligations de preuves générées précédemment de manière automatique ou semi-assistée.

8.2 Analyse statique

L'analyse statique est une technique de vérification de programme complètement automatique qui contourne les difficultés énoncés précédemment en approximant le comportement des programmes. Les outils d'analyses statiques calculent ainsi des informations sur le comportement des programmes, sans les exécuter, et prennent en compte pour cela plus de comportement que ce qu'il serait vraiment possible d'obtenir réellement avec le programme en question. Cette sur-approximation permet de simplifier suffisamment les propriétés manipulées pour toujours pouvoir répondre un verdict, tout en restant conservatif : aucune exécution du programme n'est oublié. Elle peut néanmoins donner lieu sur certains programmes à un verdict peu informatif du type « je ne sais pas ! », en raison des limites théoriques évoquées précédemment.

La plupart des analyses statiques ne prennent pas à proprement parler de spécifications en entrée. Chacune d'entre elles est donc dédiée à une propriété spécifique du langage : l'absence d'accès hors des bornes des tableaux, l'absence de déréférencement de pointeurs nuls, etc. Parfois cette propriété doit néanmoins être guidée par le programmeur qui annote son programme avec des types. L'analyse, alors appelée vérificateur de type, assure ensuite que la politique de sécurité désignée par cette annotation est bien respectée par le programme.

8.3 Software Model checking

Pour simplifier la tâche de vérification des programmes, il est possible de vérifier une abstraction de celui-ci, une représentation mathématique (« un modèle ») modélisant son comportement à un haut niveau de description. Le *model checking* est une technique de vérification dédiée aux modèles. Les vérifications sont généralement effectuées vis-à-vis d'une spécification écrite en « logique temporelle » permettant d'énoncer des propriétés du type : telle action a toujours lieu après telle autre action, etc. Le modèle abstrait d'un programme peut être obtenu

automatiquement grâce à des techniques d'analyse statique, puis vérifié ensuite par un *model checker*. On parle alors de *Software Model checking*.

Lorsqu'un programme viole une certaine propriété, la vérification échoue, mais le *model checker* a parfois la capacité de proposer une trace d'exécution exhibant une exécution fautive. Cette technique dîte de « génération de contre-exemple » donne une information intéressante pour comprendre et corriger une erreur.

Toutefois, il existe une contrepartie majeure à cette approche qui est son coût à l'exécution. En effet, un *software model checker* cherche à explorer tous les chemins d'exécution possibles afin de vérifier que tous satisfont une propriété donnée. Cette recherche d'exhaustivité peut mener à une explosion de l'espace d'états, le nombre d'états devant être explorés pouvant être au-delà des capacités des machines.

8.4 Code porteur de preuve

Les différentes techniques de méthode formelle décrites précédemment sont pour la plupart pensées comme une aide au développement. Néanmoins, dans un contexte de distribution du logiciel, il est nécessaire qu'un producteur de logiciel puisse convaincre un consommateur de la qualité de son produit. Le « Code porteur de preuve » (ou *Proof Carrying Code*), introduit par George Necula et Peter Lee [27, 26], propose d'accompagner un composant logiciel d'un certificat, une « preuve », attestant de sa validité vis-vis d'une certaine politique de sécurité. L'idée essentielle est de ne pas appliquer les mêmes techniques de vérification formelle du côté producteur et du côté consommateur. Le producteur a la charge de rechercher une preuve, alors que le consommateur peut se contenter de la vérifier (voir figure 14).

L'intérêt de cette approche réside dans les points suivants :

- **Efficacité** : La vérification de preuve est généralement plus facile que son inférence. On peut donc proposer des coûts de vérification plus faibles pour le consommateur que pour le producteur.
- **Base de confiance** : La vérification de preuve représente généralement une petite partie de l'activité d'un système de preuves de programme. Le découpage proposé par le code porteur de preuve permet d'isoler cette partie. Pour le consommateur, c'est la seule partie qui fasse vraiment partie de la base de confiance car même en cas d'erreur de construction (voir de transport) des preuves, un vérificateur correct rejettera la preuve.
- **Responsabilité du producteur** : il est très difficile de vérifier un programme que l'on n'a pas écrit. Avec le code porteur de preuve, c'est le producteur qui a la responsabilité de vérifier son programme, avec toute la connaissance qu'il possède à son sujet. Le consommateur n'a pas besoin de comprendre le programme pour vérifier sa preuve.

Cette approche n'est *a priori* pas spécifique à une technique de vérification particulière. Elle nécessite néanmoins d'avoir à sa disposition des outils de vérifications capables de générer

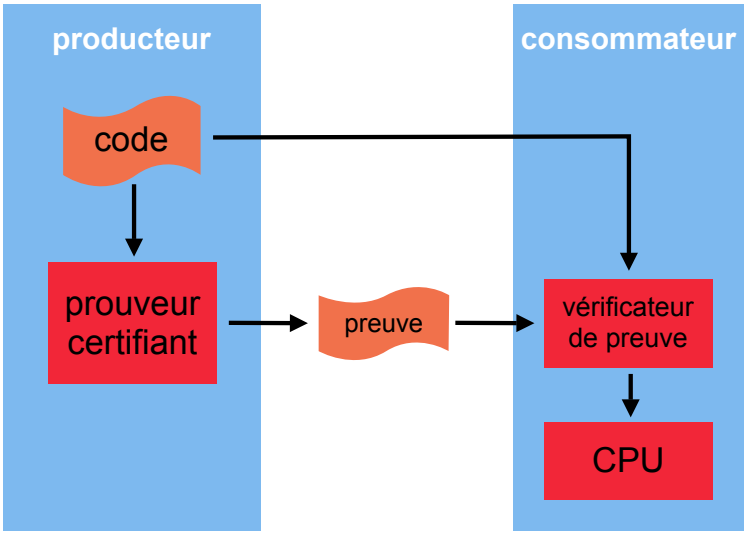


FIGURE 14 – Architecture code porteur de preuve

des *preuves* (prouveur certifiants). Une preuve peut être de diverses natures. Si les premiers travaux de Necula [26] prenaient pour format une notion très générique de $\lambda-$termes, capables d'exprimer des propriétés très riches, les travaux autour des langages d'assembleur typés [28] ou des certificats pour analyse statique [19, 20, 21] ont adopté des formats plus spécialisés. Dans le contexte de cette étude, le meilleur exemple est celui du vérificateur de *bytecode* Java, dit « léger » (voir section 5.3), qui depuis la version 6, s'appuie sur un certificat formé par des annotations de types (*stackmaps*) pour vérifier les programmes avant chargement.

Dans le cadre de Java, les autres exemples de mise en œuvre de code porteur de preuve sont les suivants :

- Le compilateur Special J, proposé par Necula et ses collaborateurs [7], permet de compiler un programme Java vers une représentation native x86 en l'accompagnant d'une preuve attestant des mêmes propriétés de sûreté mémoire que celles traditionnellement assurées par un vérificateur de *bytecode* sur un format de programme `.class`. Grâce à cette approche, l'utilisateur gagne l'efficacité du format natif sans sacrifier les propriétés de sûreté mémoire assurées habituellement sur le format *bytecode*.
- Pour étendre les propriétés assurées par le vérificateur de *bytecode*, plusieurs travaux ont proposé des approches à base de code porteur de preuve. Besson *et al.* assurent une vérification statique des interfaces pour éviter la vérification dynamique présentée dans la section 5.5.7. Besson *et al.* proposent des certificats à base d'intervalles ou de polyhèdres pour assurer une vérification statique des bornes des tableaux [21, 22] (l'utilisation d'une telle technique requiert une modification de la machine virtuelle qui par défaut effectue dynamiquement ces vérifications). Barthe *et al.* [25] proposent une vérification statique des flux d'informations avec un système de type similaire à Jif (voir section 8.5.2.1). Le vérificateur de certificat a la particularité d'avoir été formalisé et prouvé correct avec l'assistant de preuve Coq. Ghindici *et al.* [1] proposent un outil de vérification des flux d'information à base d'analyse d'alias de type *points-to* pour Java Card. Le vérificateur de certificat est suffisamment léger pour être

embarqué sur carte à puce.

8.5 Etat de l'art fonctionnel des outils existants

Dans cette section, nous présentons pour chaque classe d'outils un des représentants principaux de cette classe.

8.5.1 Méthodes déductives

En premier lieu, nous présentons le langage de spécification JML sur lequel plusieurs outils s'appuient.

8.5.1.1 JML

Le langage JML (Java Modelling Language) est un langage formel de spécification pour Java. Il permet d'ajouter une spécification du comportement du programme au code source, selon le paradigme de la programmation par contrat. Malgré une base formelle, dont l'interprétation est basée sur la logique de Hoare, le langage JML se veut un langage simple, accessible au plus grand nombre et proche de Java. Les spécifications JML sont ajoutées au code source au travers des commentaires, ce qui permet de rester compatible avec les outils ignorant ce langage. L'introduction de spécifications JML se fait au moyen de la notation standard des annotations à laquelle est ajouté le symbole `@` comme ci-dessous.

```
//@ Specification JML

/*@ Specification
  @ JML */
```

Dans la conception par contrat, les *préconditions* et les *postconditions* définissent un contrat entre une classe et ses clients. Un client doit garantir la validité des préconditions et peut supposer que les postconditions sont vérifiées. Une méthode peut supposer que les préconditions sont vérifiées et doit garantir la validité des postconditions. Les *invariants* sont des propriétés qui doivent être garanties par toutes les méthodes d'une classe. Considérons l'exemple de la figure 15. Cette classe implémente une gestion naïve d'un compte bancaire permettant de déposer et retirer une certaine somme et d'effectuer un virement depuis un autre compte. L'invariant que l'on cherche à assurer est que le solde du compte reste positif (`//@ invariant balance >=0`). La méthode `deposit` effectue un dépôt d'un montant `amount` (de type `int`) sur le compte. Pour que l'invariant soit satisfait par cette méthode, il faut que, en supposant qu'il le soit au moment

```
public class Account {

  public int balance;

  public Account(){
    balance=0;
  }
  //@ invariant balance >= 0;

  //@ requires amount >= 0;
  public void deposit(int amount){
    balance += amount;
  }

  /*@ requires balance >= amount; */
  public void withdraw(int amount){
    balance -= amount;
  }

  /*@ requires account != null &&
    @ account.balance >= amount &&
    @ amount >=0;
   */
  public void transfer(Account account, int amount){
    deposit(amount);
    account.withdraw(amount);
  }
}
```

FIGURE 15 – JML spécification

de l'appel, il le soit encore au moment du retour. Ceci n'est vrai que si l'entier `amount` est positif, ce que l'on exprime ici par la précondition `requires amount >=0`. La même approche est utilisée pour la méthode `withdraw` qui effectue un retrait. La méthode `transfer` qui effectue un virement impose trois préconditions. Les deux dernières permettent de satisfaire les préconditions de `deposit` et `withdraw`. Un autre moyen de satisfaire les préconditions de ces méthodes aurait été de réaliser les tests correspondant à l'aide de conditionnelles. On voit ici un avantage de l'utilisation de la programmation par contrat, elle permet d'alléger le code, pourvu que les assertions soient effectivement vérifiées. Ici, cette responsabilité est laissée au client qui doit respecter la spécification. La première précondition est simplement ajoutée pour garantir que l'on n'aura pas une exception de type `NullPointerException`.

En soit, le langage JML n'est qu'un langage de spécification, il n'implique aucune vérification ni statique ni dynamique et donc aucune garantie, il s'agit simplement d'annotations ajoutées au programme. Il ne faut toutefois pas sous-estimer l'intérêt intrinsèque d'un tel langage. Il offre un langage commun de description des programmes permettant de lever les ambiguités

du langage naturel. Dans l'exemple, ci dessous la spécification n'est pas respectée et l'invariant est violé.

```
Account account1 = new Account();
Account account2 = new Account();

account1.transfer(account2, 100);
```

Un compilateur standard considérant la spécification JML comme des commentaires, la compilation s'effectuera sans aucun avertissement. Il existe cependant des outils permettant d'obtenir des garanties basées sur ce langage de spécification. L'approche la plus simple consiste à instrumenter le code afin de vérifier dynamiquement les assertions devant être vérifiées pour respecter la spécification. C'est ce que font les outils OpenJML 1.6 associés à l'OpenJDK 1.6. En particulier, le compilateur (`jmlc`) introduit dans le code des vérifications dynamiques qui, si elles échouent, entraineront une erreur d'exécution. Une autre approche consiste à utiliser des outils permettant de vérifier à la compilation que ces assertions seront satisfaites à l'exécution. C'est à de tels outils que nous nous intéresserons par la suite. Le langage JML permet d'exprimer des spécifications nettement plus complexes que celles données ci-dessus. On se réferrera à la documentation du langage pour plus de renseignements.

8.5.1.2 ESC/Java(2)

Cet outil repose sur l'utilisation par le programmeur du langage d'annotations JML. Les annotations données constituent une spécification du programme qu'ESC/Java va chercher à valider à partir du code source. Parmi les propriétés pouvant être spécifiées, on peut noter l'absence de pointeurs nuls, le respect des bornes des tableaux ou des propriétés numériques simples. Si on reprend l'exemple vu précédemment, l'outil ESC/Java(2) rejettera ce programme en avertissant l'utilisateur que l'invariant peut être violé. En transformant le code de l'une des manières suivantes, le programme sera accepté.

```
Account a1 = new Account();
Account a2 = new Account();

if (a2.balance >= 100){
  a1.transfer(a2, 100);
}
```

```
Account a1 = new Account();
Account a2 = new Account();
a2.balance = 100;

a1.transfer(a2, 100);
```

On peut noter l'utilisation de `a2.balance=100` plutôt que `a.deposit(100)`. Dans le dernier cas, l'outil ne peut pas conclure que le compte `a2` dispose de suffisamment de ressources. En effet, la seule postcondition assurée par un appel à `deposit` est `balance >=0` qui provient de l'invariant de la classe. Seul ce qui est explicitement spécifié par l'utilisateur est utilisable. En ajoutant explicitement la postcondition `ensures balance == old(balance+amount)` à la méthode `deposit`, ce problème est réglé (le mot clé `ensures` désigne explicitement une postcondition). La notation `old` désigne la valeur avant appel.

ESC/Java(2) ne nécessite pas d'interaction de la part du programmeur. Les annotations et le code source sont utilisés pour générer des obligations de preuve pour le moteur de preuves Simplify (spécifiquement conçu pour ESC/Java, mais utilisé dans d'autres projets). En revanche, il n'est ni correct ni complet. Il n'est pas complet, car il peut produire des avertissements alors qu'une spécification est correcte. Il n'est pas correct, car il est possible qu'il ne rejette pas une spécification incorrecte. Considérons, l'exemple suivant, qui n'est pas rejeté par l'outil.

```
class A{
  public int val = 1;
  //Invariant val=1;
}

class B{
  A a;
  void m(){
    a = new A();
    a.val=2;
  }
}
```

L'invariant de la classe `A` est clairement violé par un appel à la méthode `m`. On a affaire ici à un problème de modularité, de manière générale, dans le cas d'une modification directe des champs d'un objet par un autre objet, l'outil ne perçoit pas les violations de propriétés.

Remarque 12 *Dans un contexte où l'utilisation de la réflexivité est contrôlée, la restriction de l'analyse à des champs privés pourrait ici contourner le problème de modularité. La correction de l'outil dans le cadre d'une telle restriction resterait toutefois un point à vérifier.*

L'absence de complétude est une propriété acceptée pour les outils d'analyse statique, en particulier si ceux-ci sont totalement automatisés. En revanche, l'absence de correction est beaucoup plus génante. Un tel outil ne peut pas être utilisé comme base de raisonnement formel ou comme outil de certification. Cela ne retire cependant rien à son intérêt en tant qu'assistant de programmation. De nombreuses erreurs de programmation peuvent être évitées *a priori*, ce qui réduit le temps de développement.

À l'origine, ESC/Java est un outil développé par Rustan Leino et al. (Compaq). Cette première version ne gérait que partiellement le langage JML. La version 2 de l'outil développée par Joe Kiniry (National University of Ireland, Dublin) et David Cok (Kodak) tend à corriger ce problème. La dernière distribution de cet outil date de novembre 2008. Un *plugin* Eclipse est disponible. Après les améliorations apportées par rapport à la première version, les développements de ces dernières années ont surtout consisté en la correction de bugs. Actuellement, il est assez difficile de prédire si l'outil est appelé à évoluer.

8.5.1.3 Krakatoa

Krakatoa est un outil pour la certification de programmes Java basé sur le langage JML (plus précisement sur KML basé sur JML). Par rapport à JML, le langage KML permet d'exprimer des propriétés plus complexes, comme des prédicats inductifs. Par exemple, il est possible d'exprimer la propriété « être triée » sur le contenu d'un tableau. Krakatoa est une interface gui génère une entrée pour Why en traduisant les programmes Java dans un langage impératif très simple.

À partir de programmes annotés, Why génère des obligations de preuves pour différents prouveurs (Coq, PVS, Simplify,...). Par rapport à un outil comme ESC/Java, Krakatoa permet de prouver des spécifications nettement plus complexes. En revanche, cela a un coût. Ces propriétés sont exprimées dans un langage plus complexe que JML et doivent généralement être prouvées par l'utilisateur. De manière générale, Why se contente de générer des conditions de vérifications pour un prouveur donné. Il est alors nécessaire de maitriser l'utilisation de ce prouveur pour pouvoir certifier un programme. L'outil est encore limité en termes d'interaction, si la preuve ne peut être réalisée de manière automatique, un fichier (coq, par exemple) est généré et l'utilisateur doit compléter ce dernier en prouvant les résultats manquants. Il est difficile de donner ici un exemple, car la lecture de ceux-ci suppose une connaissance d'au moins un des assistants de preuve utilisés par Krakatoa.

L'exemple ci-dessous est repris d'un tutoriel de Krakatoa dont l'objectif est de prouver que le résultat obtenu est bien un tableau trié par ordre croissant. Cette propriété est énoncée dans le langage d'annotations de la manière suivante :

```
/*@ predicate Sorted{L}(int a[], integer l, integer h) =
  @   \forall integer i; l <= i < h ==>
  @                          \at(a[i],L) <= \at(a[i+1],L) ;
  @*/
```

Dans cette définition, `L` dénote un état mémoire, `a` est le tableau et `l` et `h` dénotent la portion du tableau qui est triée. La notation `at(a[i],L)` dénote le contenu de la $i^{\text{ème}}$ case du tableau. Afin de pouvoir prouver que le résultat du programme est bien un tableau trié selon cette définition, deux prédicats auxiliaires sont nécessaires : le prédicat `SwapL1,L2(int a[], integer l, integer h)` et le prédicat (inductif) `PermutL1,L2(int a[], integer l, integer h)`. Le premier stipule que le tableau `a` dans l'état mémoire `L2` et obtenu en permutant les cases `l` et `h` de ce même tableau dans l'état mémoire `L1`. Le second définit de manière unique la propriété « être une permutation » d'un tableau. Ces différents prédicats sont utilisés dans les annotations des méthodes pour introduire les préconditions, postconditions et invariants de la classe. La possibilité de représenter les états mémoires dans les formules permet par exemple de relier la valeur d'une référence avant et après l'appel d'une méthode. Par exemple, dans le cas de la méthode `Swap`, on a la spécification `ensures SwapOld,Here(t,i,j)` qui demande que le tableau après appel (dans l'état mémoire `Here`) soit bien le résultat de la permutation des cases `i` et `j` du tableau avant l'appel (dans l'état mémoire `Old`).

```
/*@ predicate Sorted{L}(int a[], integer l, integer h) =
  @   \forall integer i; l <= i < h ==>
  @                          \at(a[i],L) <= \at(a[i+1],L) ;
  @*/

/*@ predicate Swap{L1,L2}(int a[], integer i, integer j) =
  @   \at(a[i],L1) == \at(a[j],L2) &&
  @   \at(a[j],L1) == \at(a[i],L2) &&
  @    \forall integer k; k != i && k != j ==>
  @                            \at(a[k],L1) == \at(a[k],L2);
  @*/

/*@ inductive Permut{L1,L2}(int a[], integer l, integer h) {
  @  case Permut_refl{L}:
  @   \forall int a[], integer l h; Permut{L,L}(a, l, h) ;
  @  case Permut_sym{L1,L2}:
  @    \forall int a[], integer l h;
  @      Permut{L1,L2}(a, l, h) ==> Permut{L2,L1}(a, l, h) ;
  @  case Permut_trans{L1,L2,L3}:
  @    \forall int a[], integer l h;
  @      Permut{L1,L2}(a, l, h) && Permut{L2,L3}(a, l, h) ==>
  @        Permut{L1,L3}(a, l, h) ;
  @  case Permut_swap{L1,L2}:
  @    \forall int a[], integer l h i j;
  @       l <= i <= h && l <= j <= h && Swap{L1,L2}(a, i, j) ==>
  @     Permut{L1,L2}(a, l, h) ;
  @ }
  @*/

class Sort {

  /*@ requires t != null &&
    @    0 <= i < t.length && 0 <= j < t.length;
    @ assigns t[i],t[j];
    @ ensures Swap{Old,Here}(t,i,j);
    @*/
  void swap(int t[], int i, int j) {
    int tmp = t[i];
    t[i] = t[j];
    t[j] = tmp;
  }

  /*@ requires t != null;
    @ behavior sorts:
    @   ensures Sorted(t,0,t.length-1);
    @ behavior permuts:
    @   ensures Permut{Old,Here}(t,0,t.length-1);
    @*/
```

```
    void min_sort(int t[]) {
      int i,j;
      int mi,mv;
      /*@ loop_invariant 0 <= i;
        @ for sorts:
        @  loop_invariant Sorted(t,0,i) &&
        @   (\forall integer k1 k2 ;
        @      0 <= k1 < i <= k2 < t.length ==> t[k1] <= t[k2]);
        @ for permuts:
        @  loop_invariant
        @   Permut{Pre,Here}(t,0,t.length-1);
        @*/
      for (i=0; i<t.length-1; i++) {
        // look for minimum value among t[i..n-1]
        mv = t[i]; mi = i;
        /*@ loop_invariant
          @   i < j &&
          @   i <= mi < t.length &&
          @   mv == t[mi];
          @ for sorts:
          @  loop_invariant
          @   (\forall integer k; i <= k < j ==> t[k] >= mv);
          @ for permuts:
          @  loop_invariant
          @   Permut{Pre,Here}(t,0,t.length-1);
          @*/
        for (j=i+1; j < t.length; j++) {
          if (t[j] < mv) {
            mi = j ; mv = t[j];
          }
        }
        swap(t,i,mi);
      }
    }
}
```

La spécification de la méthode `min_sort` demande que le nouveau tableau soit bien trié et soit bien une permutation de l'ancien. Le corps de la méthode contient plusieurs invariants qui sont nécessaires à la preuve. À partir de la spécification de ce programme, l'outil Krakatoa va générer les conditions de vérifications pour l'assistant de preuve choisi. Il reste à la charge du programmeur de prouver ces conditions. Le nombre de conditions de vérifications peut être important. Dans le cas de cet exemple, il y en a 18 que nous n'allons évidemment pas énumérer ici. Certaines demandent de prouver des résultats numériques triviaux, alors que d'autres demandent des raisonnements plus subtils sur l'algorithme utilisé. Ci-dessous, à titre d'exemple, on donne la condition de vérification associée à l'annotation `ensures Swap(Old,Here)(t,i,j)` de la méthode `swap`. On retrouve sous la barre le résultat à prouver, et au dessus les hypothèses disponibles. Les différents noms apparaissant dans les hypothèses proviennent de l'axiomati-

sation en Coq de la sémantique. Par exemple, `select intM_intP(shift t i_2)` dénote la valeur à l'adresse correspondant à un décalage `i_2` par rapport à l'addresse `t` (la case `i_2` du tableau `t`) dans l'état mémoire `intM_intP`. Pour plus d'information sur les autres éléments, on pourra se référer à la documentation de Krakatoa.

```
this_0 : pointer Object
t : pointer Object
i_2 : Z
j_1 : Z
Object_alloc_table : alloc_table Object
intM_intP : memory Object Z
HW_1 : left_valid_struct_intM t 0 Object_alloc_table /\
       valid_struct_Sort this_0 0 0 Object_alloc_table /\
       Non_null_intM t Object_alloc_table /\
       0 <= i_2 /\
       i_2 < offset_max Object_alloc_table t + 1 /\
       0 <= j_1 < offset_max Object_alloc_table t + 1
result : Z
HW_2 : result = select intM_intP (shift t i_2)
result0 : Z
HW_3 : result0 = select intM_intP (shift t j_1)
intM_intP0 : memory Object Z
HW_4 : intM_intP0 = store intM_intP (shift t i_2) result0
intM_intP1 : memory Object Z
HW_5 : intM_intP1 = store intM_intP0 (shift t j_1) result
============================
 Swap t i_2 j_1 intM_intP1 intM_intP /\
 not_assigns Object_alloc_table intM_intP intM_intP1
   (pset_union (pset_range (pset_singleton t) j_1 j_1)
      (pset_range (pset_singleton t) i_2 i_2))
```

8.5.2 Analyse statique

L'analyse statique fait d'ores et déjà partie intégrante de l'architecture Java standard à travers la présence du vérificateur de *bytecode* (voir la section 5.3) qui assure, avant l'exécution des programmes, une partie des propriétés intrinsèques du langage. Nous présentons dans cette section trois exemples d'analyses statique qui permettent d'étendre le spectre des propriétés assurées par le vérificateur de *bytecode*.

8.5.2.1 Vérification des flux d'information avec Jif

Jif [4] est une extension du langage Java rajoutant des propriétés de confidentialité aux propriétés intrinsèques du langage. Cette extension s'appuie sur une vérification statique et dyna-

mique afin de protéger la confidentialité et l'intégrité des informations manipulées. L'outil agit sur les programmes sources et permet de générer des programmes Java standards. Il s'oriente donc vers les développeurs voulant vérifier les flux d'information de leur programmes. Il est pour cela nécessaire d'annoter les champs, paramètres, variables des programmes avec des labels indiquant la politique de sécurité souhaitée en termes de flux d'information.

Jif s'appuie sur un langage d'annotation évolué [3] permettant d'exprimer une politique de sécurité multi-acteurs : plusieurs acteurs (*principals* en anglais) peuvent intervenir dans la mise en place de la politique de sécurité. Une annotation de donnée consiste en une conjonction de contraintes où chaque acteur peut imposer, pour sa politique, une liste d'acteurs (en plus de lui-même) autorisés à lire (ou écrire) la donnée. Par exemple, l'annotation suivante :

```
int {Alice:Robert,Carole} x;
```

déclare non seulement la variable `x` avec un type `int`, mais impose sur cette variable une politique de sécurité qui indique que l'acteur Alice autorise seulement que cette information soit lue (directement ou indirectement) par les acteurs Robert et Carole. Implicitement, Alice a aussi ce droit. Une variable entièrement publique a l'annotation `{}` qui n'impose aucune contrainte. Le système de type de Jif assure ensuite qu'une valeur est toujours affectée à une variable possédant au moins autant de contraintes. Ainsi le code suivant est accepté, car `b` possède au moins autant de restrictions que `a` (Alice autorise Robert et Carole à lire `a`, mais elle n'autorise que Robert pour lire `b`).

```
int{Alice:Robert,Carole} a = In.read_int();
int{Alice:Robert} b = a;
```

Le code suivant est par contre rejeté, car Carole n'a ici pas de droit de lecture sur `a`, alors qu'elle en a sur `b`.

```
int{Alice:Robert} a = In.read_int();
int{Alice:Robert,Carole} b = a;
```

Le langage d'annotation est capable de gérer plusieurs acteurs. Cela est utile, car une même classe peut être annotée avec un acteur craintif qui donnera très peu de droits aux autres et un acteur plus laxiste qui mettra en place une politique moins restrictive. Dans les exemples précédents, Alice est « propriétaire » de l'information au sens DAC, mais le système de type assure plus de chose que pour le contrôle d'accès standard, puisqu'il permet de contrôler la propagation des informations : une fois l'information lue par Robert, ce dernier ne peut pas le donner en lecture à un acteur à qui Alice n'a pas donné de droit de lecture. Ce type de propriété sort du cadre du contrôle d'accès standard.

Les deux exemples précédents sont des *flux directs* : une information est directement placée dans une donnée. Jif tient aussi compte des *flux indirects* : la valeur dépend indirectement d'une décision prise pour suivre le flot de contrôle. L'exemple suivant est ainsi rejeté

```
int{Alice:Robert} a = In.read_int();
int{Alice:Robert,Carole} b;
if (a == 1) { b = 0; } else { b = 1; };
```

Celui ci est accepté.

```
int{Alice:Robert,Carole} a = In.read_int();
int{Alice:Robert} b;
if (a == 1) { b = 0; } else { b = 1; };
```

Dans les deux cas, la valeur finale de `b` dépend de la valeur de `a`. Il y a donc une affectation implicite qui est sujette aux mêmes restrictions que l'affectation directe.

Pour alléger le travail d'annotation par le programmeur, Jif s'appuie sur un mécanisme d'inférence puissant pour « deviner » une partie des annotations. Il utilise aussi plusieurs analyses statiques pour prédire l'absence de certaines exceptions dans le programme. Cela est nécessaire, car toute levée potentielle d'exception induit une branche dans le graphe de flot de contrôle du programme, et, par conséquent, un risque de flux implicite. Dans l'exemple suivant, le programme est rejeté, car il y un flux indirect entre la valeur de `x` (nulle ou non) et l'affectation `b = 0`.

```
A{Alice:Bob} x = A.read();
int{Alice:Bob,Carole} b;
b = 1;
x.m();
// une exception NullPointerException pourrait etre lancee ici
b = 0;
```

Dans le programme suivant, Jif utilise son analyse de pointeurs nuls pour comprendre que `x` n'est jamais nul et que, par conséquent, aucun branchement n'a lieu avant l'affectation `b = 0`.

```
A{Alice:Bob} x = new A();
int{Alice:Bob,Carole} b;
b = 1;
x.m();
b = 0;
```

Pour conclure, Jif est un langage puissant et est certainement le meilleur représentant actuel de l'état de l'art en termes de vérification statique des flux d'information. Son utilisation reste difficile en raison de la complexité de sa politique de sécurité et des contraintes qu'il impose au programmeur pour réussir à écrire un programme vérifiable. Son emploi pour des projets de programmation conséquents (mais encore académiques) a néanmoins été rapporté par Askarov et Sabelfeld [2].

8.5.2.2 FindBugs

Les analyses statiques raisonnent sur les programmes en considérant tous les chemins possibles d'exécutions. Les limites théoriques évoquées dans l'introduction de cette section obligent cependant ces outils à considérer plus de chemins qu'il n'en existera vraiment. L'outil peut ainsi

prédire une erreur qui ne correspond à aucun chemin réaliste et donc ne jamais se produire. Ce phénomène, dit de « fausse alarme », pénalise grandement les outils d'analyses statiques qui ont, en pratique, tendance à inonder l'utilisateur de fausses alarmes, sans pouvoir l'aider à discerner les vraies alarmes (celles pour lesquelles il existe un chemin d'exécution capable de produire l'erreur), des fausses, autrement que par une revue de code manuelle. Si l'analyse est correcte, cet ensemble d'alarmes reste cependant une sur-approximation correcte des véritables points d'erreurs, et il est inutile de chercher des erreurs en dehors de ces points.

L'outil FindBugs [23] est un outil d'analyse statique qui a pour objectif de restreindre les alarmes annoncées à l'utilisateur à celles qui ont de grandes chances d'être de vraies alarmes. Cette approche n'est pas correcte, puisqu'elle cache parfois certaines vraies alarmes à l'utilisateur, ce qui rend son usage problématique dans un contexte sécuritaire. Cependant, elle apporte une solution intéressante aux problèmes des fausses alarmes et a démontré, en pratique, sa capacité à montrer aux programmeurs des erreurs de codage subtiles. L'outil a ainsi été testé [23] sur les développements Java de la société Google avec un étonnant ratio de vraies alarmes : sur 4000 alarmes désignées par l'outil comme des erreurs hautement probables, 75% ont été marquées par les développeurs de la société comme des erreurs effectives à corriger après une revue de code manuelle.

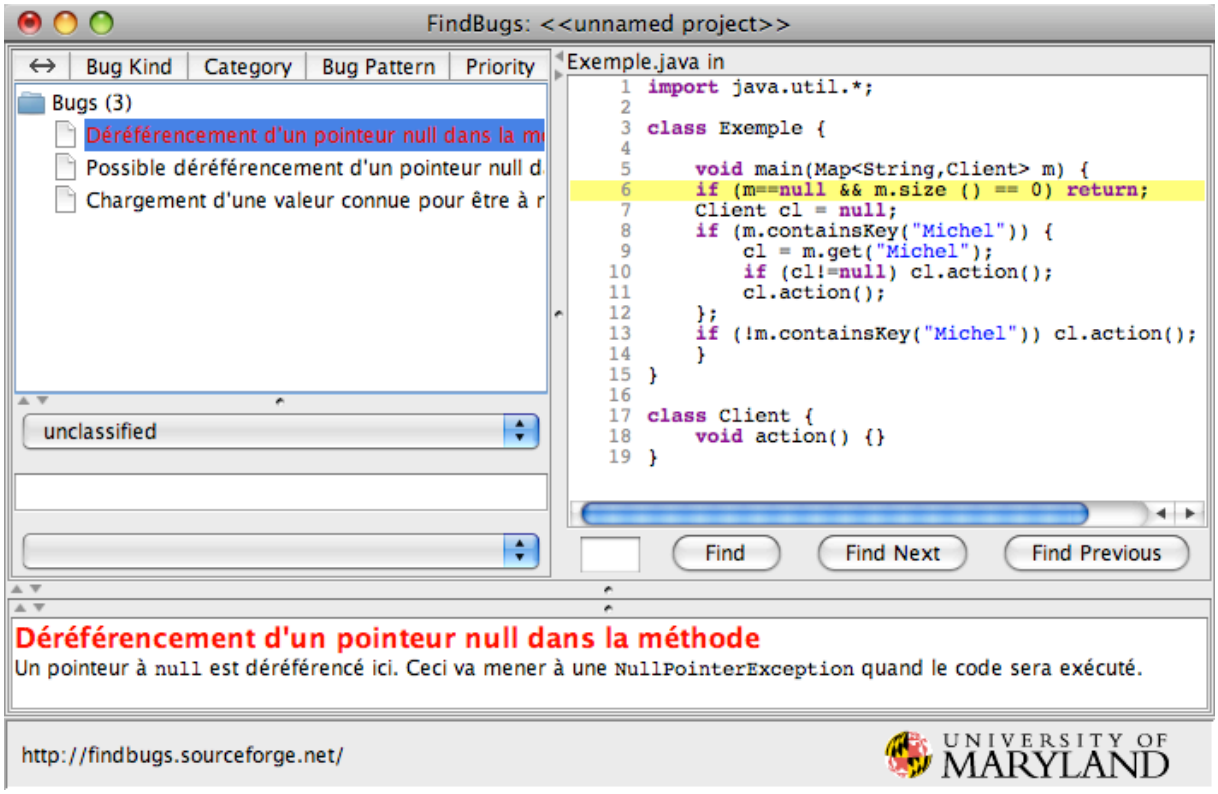


FIGURE 16 – Exemple d'utilisation de FindBugs

La figure 16 présente un exemple d'utilisation de l'outil. Sur le programme en exemple, l'outil présente trois alarmes. La première et la dernière désignent de vraies alarmes, car, à la ligne 6, `m.size()` provoquera une exception `NullPointerException`. Le programmeur a dans cet exemple de code utilisé un `&&` à la place d'un `||`. La deuxième alarme désigne la

ligne 11 comme susceptible de déréférencer un pointeur nul. Il s'agit d'une fausse alarme, car le test `m.containsKey("Michel")` à la ligne 8 assure que `m.get("Michel")` à la ligne 9, ne sera pas égal à `null`. Sur cet exemple, deux autres alarmes ont été masquées par l'outil. La première concerne la ligne 10 où le déréférencement `cl.action()` est correct grâce au test qui le précède. L'analyseur a compris cela et a ainsi automatiquement considéré cette alarme comme fausse. La deuxième concerne la ligne 13. Cette fois, l'outil a masqué une vraie alarme. Ce genre d'erreurs, qui dépend d'une condition de chemin complexe, est généralement masquée à l'utilisateur, car elle provoque beaucoup de fausses alarmes en pratique. Sur cet exemple, il démontre l'absence de correction de l'outil.

FindBugs cherchait initialement [17] a détecter principalement des erreurs de déréférencement de pointeurs nuls. L'outil a depuis élargi son spectre de vérifications à d'autres règles de codage. Il cherche à assurer une cinquantaine de propriétés liées à l'initialisation des champs, à la synchronisation des *threads*, etc. . . , mais sans liens directs avec la sécurité. L'outil travaille au niveau *bytecode*, mais affiche ses résultats au niveau source. C'est un projet très actif qui semble bien accueilli par des grandes sociétés comme Google. Dans un contexte sécuritaire, l'outil n'apportera aucune certitude, mais on ne peut que conseiller de le lancer sur un développement Java afin de déceler rapidement des erreurs de programmation pouvant mener à des dysfonctionnements.

8.5.2.3 Recherche de vulnérabilités par analyse statique d'alias

Les analyses statiques que nous avons présentées jusqu'à présent (le vérificateur de *bytecode*, Jif, Findbugs) ont un traitement très peu précis des « alias ». Une analyse d'alias cherche à prédire quels noms symboliques vont pointer vers la même référence durant l'exécution d'un programme. Un nom symbolique peut être le nom d'une variable locale dans une méthode, mais aussi un nom de variable locale, suivi d'un déréférencement à un champ, suivi d'un accès à un élément de tableaux (par exemple `x.f.a[i+1]`). Comme nous l'avons déjà expliqué, une analyse doit inévitablement réaliser des approximations. Dans le contexte des alias, cela se caractérise par deux types de calculs d'alias. Une analyse de *must-alias* calcule des chemins symboliques qui seront systématiquement en alias à l'exécution, mais elle ne calcule pas forcément tous ces chemins. Une analyse de *may-alias* calcule un sur-ensemble des chemins symboliques qui pourraient être en alias à l'exécution, mais elle calcule parfois plus de chemins que ceux qui seront vraiment en alias.

Une analyse d'alias précise est un outil important pour comprendre un programme Java en raison de l'utilisation intensive des structures allouées dynamiquement. Le sujet a fait l'objet de plusieurs études depuis plus de 30 ans, mais il a longtemps été difficile d'atteindre une bonne précision de ces analyses sans compromettre le passage à l'échelle des gros programmes. La situation a quelque peu évolué ces dernières années avec l'apparition d'analyses d'alias s'appuyant sur des arbres binaires de décisions (en anglais, *Binary Decision Diagram* ou BDD) pour manipuler symboliquement et efficacement les millions d'informations nécessaires sur les larges programmes. Le travail le plus marquant dans ce domaine est dû à John Whaley et Monica

Lam de l'université de Stanford [31, 30] dont l'analyse est capable de manipuler implicitement (à travers une représentation symbolique compacte) 10^{14} chemins différents. Cela leur permet d'analyser avec une bonne précision les alias de programmes comprenant plus de 2000 classes.

L'existence de telles analyses a ouvert la voie à de nouvelles analyses pour les programmes Java.

Mayur Naik et Alex Aiken [41, 40, 39] ont ainsi appliqué ces analyses pour détecter la présence de *data-race* sur de larges programmes. Les informations d'alias sont nécessaires dans ce contexte afin d'avoir une bonne compréhension des différents chemins d'exécution dans un programme complet. La prise de verrous étant faite sur les objets, il est aussi important de pouvoir prédire quels objets seront en alias avec quels autres objets.

Michael Martin, Benjamin Livshits et Monica Lam [50, 38] ont, pour leur part, appliqué ces analyses d'alias pour détecter des vulnérabilités dans les applications web Java. Ils proposent un langage de requête PQL afin de réaliser des audits de sécurité sur des programmes Java complets.

Avec le langage PQL, un utilisateur qui souhaite détecter la présence de vulnérabilité vis-à-vis d'injection SQL pourra écrire la requête suivante :

```
query main()
returns
     object Object sourceObj, sinkObj;
matches {
     sourceObj := source();
     sinkObj := derived*(sourceObj);
     sinkObj := sink();
}
```

Dans cette requête, il spécifie qu'il souhaite détecter l'existence d'un chemin entre deux objets symboliques `sourceObj` et `sinkObj` tels que

- `sourceObj` vérifie la spécification PQL `source` modélisant les objets créés directement à partir d'une entrée utilisateur,
- `sinkObj` est lié à `sourceObj` par la spécification `derived*` qui spécifie essentiellement les chemins que l'utilisateur souhaite suivre depuis `sourceObj`,
- `sinkObj` vérifie la spécification `sink` qui caractérise les points où la requête SQL est effectuée.

Nous décrivons maintenant les trois spécifications `source`, `derived*` et `sink`. La spécification `source` décrit tous les points d'entrées susceptibles de recevoir une entrée utilisateur qu'il convient de surveiller, par exemple le résultat d'un appel à la méthode Java `getParameter` sur un objet de classe `HttpServletRequest`.

```
query source()
returns
    object Object sourceObj;
uses
    object HttpServletRequest req;
matches {
       sourceObj = req.getParameter(_)
     | sourceObj = req.getHeader(_)
     | ...
}
```

La spécification `sink` décrit tous les points d'appels à une requête SQL, par exemple en passant un objet en argument de la méthode Java `executeQuery` sur une instance de classe `java.sql.Statement`. Si l'objet provient d'une source non-sûre, un tel appel peut donner lieu à une attaque.

```
query sink()
returns
     object Object sinkObj;
uses
     object java.sql.Statement stmt;
     object java.sql.Connection con;
matches {
      stmt.executeQuery(sinkObj)
    | stmt.execute(sinkObj)
    | con.prepareStatement(sinkObj)
    | ...
}
```

La dernière spécification, `derived*` décrit formellement cette notion de « provenance ».

```
query derived*(object Object x)
returns
     object Object y;
uses
     object Object temp;
matches {
     !sanitizer1(x); !sanitizer2(x); ...
     y := x |
     temp := derived(x); y := derived*(temp);
}
```

Une instance `y` sera considérée comment « provenant » d'une instance `x` si :

- aucune fonction de nettoyage (`sanitizer1`, `sanitizer2`) n'a été utilisée, par exemple pour rendre une chaîne de caractères inoffensive en termes d'attaque SQL,
- `x` est directement copié dans `y` ou bien il existe une instance intermédiaire `temp` qui provient directement de `x` (décrite par la spécification `derived`) et dont `y` provient par appel récursif à la fonction `derived*`.

Pour finir, la spécification `derived` contient simplement les liens directs qui peuvent exister entre une source `x` et une cible `y`.

```
query derived(object Object x)
returns
     object Object y;
matches {
     y.append(x)
     | y = _.append(x)
     | y = new String(x)
     | y = new StringBuffer(x)
     | y = x.toString()
     | y = x.substring(_ ,_)
     | y = x.toString(_)
     | ...
}
```

À partir de cette collection de spécifications, il est possible de réaliser un audit complètement automatique sur un programme complet qui tiendra compte de tous les chemins possibles. Une particularité notable de ces travaux est qu'ils assurent une couverture totale d'un programme, sans oublier aucun chemin. Ils s'appuient pour cela sur une analyse de la réflexion [36], afin de suivre aussi les chemins liés à des primitives réflexives.

Prenons l'exemple d'une application web dont les requêtes SQL sont gérées par une séquence de commande de la forme suivante (ces instructions n'apparaissent pas forcément dans la même partie du programme) :

```
1. String userName = req.getParameter("username");
2.
3. StringBuffer buf1;
4. StringBuffer buf2;
5. ...
6. buf1.append(userName);
7. String query = buf2.toString();
8. stmt.executeQuery(query);
```

La requête PQL détectera ici une vulnérabilité potentielle si jamais les instructions réalisées à la ligne 5 permettent de créer un alias entre les chaînes mutables `buf1` et `buf2`.

Pour conclure, cette technique d'analyse représente une des méthodes les plus sophistiquées pour analyser automatiquement de larges programmes, et sans annotations de la part du programmeur. Le principal problème provient du manque d'outils autour de ces techniques. Les travaux cités précédemment ont donné lieu à des prototypes de recherches qui ne sont généralement pas maintenus. Le projet Serenitec [51] actuellement financé par la région Bretagne cherche à construire un atelier d'audit de programmes Java basé sur ces techniques, en collaboration avec l'équipe-projet Celtique de l'INRIA Rennes.

8.5.3 Software Model Checking

8.5.3.1 JavaPathFinder

JavaPathFinder est un *software model checker*. Un tel outil est une sorte de machine virtuelle, mais contrairement à une machine virtuelle standard, l'exécution sur celle-ci ne se contente pas d'explorer une voie d'exécution. Au contraire, une telle machine cherche à explorer plusieurs chemins possibles en parallèle afin de vérifier que tous satisfont une propriété donnée. La distinction entre *software model checker* et analyse statique est parfois une simple question de présentation. Un *model checker* a cependant une précision généralement assez elevée par rapport aux analyses statiques les moins sophistiquées. La contrepartie réside dans le coût de cette approche, le nombre d'états devant être exploré pouvant être au-delà des capacités des machines. On parle alors d'explosions de l'espace d'états. Différentes techniques sont mises en œuvre pour diminuer la taille de cet espace, mais la limite reste là. Par rapport à une approche basée sur du test, un tel logiciel permet d'explorer toutes les voies d'exécution. Ceci représente un avantage important dans le cadre du *multi-threading* où la rejouabilité des exécutions n'est pas garantie. Cet outil est développé par le laboratoire AMES de la NASA.

JavaPathFinder permet de détecter les interblocages (ou *deadlocks*) et les exceptions non prises en charge (par exemple, `NullPointerException` et `AssertionError`). Un des avantages majeurs de JavaPathFinder est que l'utilisateur peut étendre les propriétés qui peuvent être vérifiées en lui implémentant ses propres classes de propriétés. De telles extensions (vérification de *race condition*, de bornes mémoire) sont incluses dans la distribution de JavaPathFinder.

Cependant, nous pouvons citer un certain nombre de restrictions dans l'utilisation directe de JavaPathFinder :

- il ne peut être utilisé que sur des programmes Java n'utilisant pas de méthode native ;
- il n'y a pas de support pour des programmes utilisant les bibliothèques `java.awt` ou `java.net` ;
- il y a un support très limité pour des programmes utilisant la réflexion ou la bibliothèque `java.io` ;
- il ne peut vérifier des applications de plus de 10.000 lignes de code effectives.

51. `http://www.serenitec.fr`

Pour contourner ces restrictions, JavaPathFinder fournit une interface appelée *Model Java Interface* (MJI) qui permet de créer des abstractions de certaines classes ou méthodes. On peut citer trois principales utilisations de cette interface :

1. intercepter les méthodes natives. Sans la couche supplémentaire fournie par MJI, JavaPathFinder ne pourrait pas vérifier complètement les applications où des effets de bord peuvent apparaître au sein des méthodes natives.
2. interfacer l'application avec les fonctionnalités de bas niveau fournies par JavaPathFinder. En effet, les utilisations de certaines classes de la bibliothèque standard (comme `java.lang.Class` ou `java.lang.Thread`) doivent être interceptées pour être interfacées avec les modèles internes de classes, objet et *thread* de JavaPathFinder.
3. diminuer la taille de l'espace d'états. Cela consiste entre autre à créer des abstractions de certaines méthodes qui ne sont pas jugées comme pertinente dans la vérification de certaines propriétés.

8.6 Synthèse comparative des outils existants

Afin de synthétiser les résultats de cette étude, nous proposons de dégager plusieurs critères de comparaison qui s'appliquent aux méthodes de vérification formelle en générale.

1. Quels types de propriétés peuvent être assurées par l'outil ?
   - riche : à peu près n'importe quelle spécification formelle (afin de prouver des propriétés fonctionelles fortes),
   - restreinte : seul une classe très restreinte est prise en compte (absence de pointeurs nuls, d'accès hors des bornes des tableaux, etc...).
2. Les propriétés assurées ont-elles un impact direct sur la sécurité d'un programme, autre que la correction fonctionelle ?
   - oui,
   - non.
3. Quelle est le degré de correction de l'outil ?
   - faible : l'outil a vocation à chercher des erreurs, mais pas à démontrer leur absence,
   - fort : l'outil prétend démontrer certaines propriétés, mais n'est pas à l'abri d'une erreur d'implémentation ou de spécification dans l'outil lui-même,
4. Quelle expertise est-il nécessaire d'avoir dans l'outil pour mener à bien la vérification ?
   - programmeur standard (PS) : la technique de vérification est accessible au programme Java standard, sans connaissance particulière en sécurité ou en méthodes formelles (langage de spécification, preuve formelle),

- expert sécurité (ES) : la technique demande une bonne connaissance des problématiques de sécurité,
- expert en méthodes formelles (EMF) : l'utilisateur doit maîtriser les langages de spécifications formelles et la preuve formelle.

5. Quel est le coût en temps et en personnel pour la vérification d'un programme standard ?
   - faible : la technique s'intègre facilement dans un processus de validation standard,
   - moyen : la technique demande une première mise en place de politique de sécurité, mais demande ensuite assez peu de travail pour traiter chaque nouveau programme,
   - fort : la technique demande un travail spécifique à chaque nouveau programme et augmente fortement le temps de développement.

Outil/technique de vérification	Propriétés assurées	Orientation Sécurité	Correction	Expertise requise	Effort nécessaire
JML	riche	non	faible	PS	fort
ESC/Java	restreinte	non	faible	PS	fort
Krakatoa	riche	oui	fort	EMF	fort
Jif	restreinte	oui	fort	ES	fort
FindBugs	restreinte	non	faible	PS	faible
bddbddb/PQL	restreinte	oui	fort	ES	moyen
JavaPathFinder	riche	non	faible	EMF	fort

9 IDENTIFICATION DES DÉVIATIONS POSSIBLES

9.1 Problématique de l'interfaçage avec l'environnement natif

L'interfaçage entre le code Java et les bibliothèques natives (dont la JVM) est réalisé à l'aide de JNI (*Java Native Interface*). Cette fonctionnalité nécessite un support de la JVM. Cette interface est nécessaire, car la JVM constitue une abstraction incomplète de la plate-forme d'exécution native. En effet, si elle fournit une interface avec le processeur et la mémoire vive, un certain nombre de services ne sont pas offerts. Ces services sont généralement implémentés par le système d'exploitation et sont pourtant fondamentaux en Java (fichiers, *sockets*, etc.). De plus, les JVM étant généralement développées en C ou C++ et distribuées sous la forme d'une bibliothèque partagée ou d'un exécutable natif, l'interface JNI permet également au code Java d'accéder à certaines fonctionnalitées de la JVM qui ne sont pas implémentées par un mot-clé du langage. Certaines méthodes de la bibliothèque standard doivent en effet pouvoir appeler explicitement certaines fonctions natives de la JVM (c'est par exemple le cas de la méthode `define` des chargeurs de classe). Concrètement, JNI offre les fonctionnalités suivantes :

- elle permet au code Java d'appeler des fonctions natives définies dans des bibliothèques partagées écrites en C ou en C++ ;
- elle permet à des fonctions natives de manipuler des variables de types Java (primitifs ou références), qui leurs sont passées en paramètre ou qu'elles instancient (et retournent éventuellement aux méthodes Java qui les appellent) ;
- elle permet à des fonctions natives d'interagir avec la JVM (notamment d'instancier une nouvelle instance de la JVM pour exécuter une application Java).

9.1.1 Méthodes natives

Afin d'appeler une fonction native, le code Java doit disposer d'une méthode native correspondant à cette fonction. Cette méthode est définie au sein d'une classe et sert d'interface avec la fonction native. Elle peut être privée ou publique (bien qu'il soit généralement recommandé de déclarer privée ce type de méthode). Elle correspond à la signature (en termes de types Java) de la fonction native qu'elle représente. Elle est déclarée via le mot-clé `native` et ne doit pas être implémentée (pas de bloc d'implémentation après la déclaration, même vide), car elle constitue une interface pour le code de la fonction native. Il est possible de passer des paramètres de type primitif ou référence (objet, tableau) au code natif. Lui-même peut retourner une valeur de type primitif ou référence.

Afin d'utiliser cette méthode, il convient de suivre les étapes suivantes :

1. définir la méthode native dans une classe Java ;

2. implémenter le chargement de la bibliothèque native qui fournit la fonction native interfacée. Durant l'exécution de l'application Java, le chargement doit impérativement être réalisé avant tout appel à la méthode native. Il est donc conseillé de l'implémenter dans l'initialisateur statique de la classe [52] ;
3. compiler la classe Java ;
4. générer le fichier d'en-tête correspondant à l'aide de l'outil `javah` ;
5. écrire le code natif en utilisant entre autre les fichiers d'en-tête fourni par le JDK et celui généré précédemment ;
6. compiler le code natif sous la forme d'une bibliothèque partagée.

9.1.2 Implémentation de fonctions natives

Les fonctions natives peuvent être implémentées en C ou en C++ [53]. Le développeur doit spécifier le code des fonctions natives correspondantes à chaque méthode.

Le prototype de ces fonctions suit une convention définie par la spécification de JNI (et donnée automatiquement par `javah`). Le nom de la fonction doit notamment reprendre le nom de la classe et de la méthode native, ainsi que la signature Java. Le premier paramètre est un pointeur vers une structure interface de type `JNIEnv`. Cette structure contient des pointeurs vers les fonctions JNI qui permettent au code natif d'interagir avec « le monde Java » (créer des objets Java, appeler des méthodes, convertir les types primitifs Java vers les types natifs et vice et versa, etc.). Si la méthode native est une méthode d'instance, le second paramètre est un pointeur vers l'instance d'objet à laquelle appartient la méthode native appelée (c'est l'équivalent du « `this` »). L'exemple suivant, issu de la documentation de JNI [54] illustre l'implémentation d'une méthode native :

```
package pkg;

class Cls {
  native double f(int i, String s);
  ...
}
```

Le code de la fonction correspondante :

```
jdouble Java_pkg_Cls_f__ILjava_lang_String_2 (
     JNIEnv *env,        /* interface pointer */
     jobject obj,        /* "this" pointer */
```

52. Le chargement d'une bibliothèque native est réalisé par un appel à la méthode `loadLibrary` de la classe `java.lang.System`. Le nom de la bibliothèque (sans l'extension) est passé en paramètre. Le chargement nécessite une permission adéquate.
53. JNI ne supporte que ces langages, mais les fonctions natives peuvent elles-même appeler d'autres fonctions écrites à l'aide de langages différents en utilisant des mécanismes d'interfaces ad-hoc type SWIG.
54. `http://java.sun.com/javase/6/docs/technotes/guides/jni/`

```
      jint i,              /* argument #1 */
      jstring s)           /* argument #2 */
{
      /* Obtain a C-copy of the Java string */
      const char *str = (*env)->GetStringUTFChars(env, s, 0);

      /* process the string */
      ...

      /* Now we are done with str */
      (*env)->ReleaseStringUTFChars(env, s, str);
      return ...
}
```

S'il est possible d'accéder aux objets Java depuis le monde natif et d'instancier de nouveaux objets, il n'est en revanche pas possible d'allouer de nouveaux objets directement depuis le code natif (en utilisant des primitives du système d'exploitation). Pour créer de nouveaux objets, le code natif doit appeler des fonctions de la structure `JNIEnv` qui font elles-mêmes appel aux fonctions d'allocation des objets Java de la JVM. En retour, le code natif obtient une référence sur l'objet (de la même manière que le code Java d'une méthode). Toutefois, le code natif étant susceptible d'utiliser des variables globales, situées sur le tas natif et plusieurs fonctions (ou les différents appels d'une même fonction) pouvant y accéder, JNI supporte le type de référence « native » :

- les références locales, qui sont uniquement valide pendant la durée d'un appel à une fonction native (elles sont automatiquement libérées lors du retour de la fonction) ;
- les références globales, qui restent valide jusqu'à ce qu'elles soient explicitement libérées.

Par défaut, les références locales sont utilisées. Le développeur peut transformer une référence locale en référence globale, mais il doit le faire explicitement. Les deux types de références s'utilisent de la même manière, notamment pour accéder aux champs et aux méthodes des objets.

La désallocation des objets est réalisée par le *garbage collector* de la JVM, comme pour le code Java. Le programmeur doit cependant libérer explicitement les références globales à l'aide de la fonction `DeleteGlobalRef` de la structure `JNIEnv`. Il peut également, bien que cela ne soit pas impérativement nécessaire, libérer les références locales (par exemple, pour un code natif manipulant une grande quantité d'objets, il peut être judicieux de libérer les références au fur et à mesure que le traitement sur les objets est réalisé, afin de permettre au *garbage collector* de supprimer l'objet s'il n'est plus référencé (dans le monde natif comme dans le monde Java).

9.1.3 Interactions avec la JVM

JNI définit également un ensemble de primitives permettant au code natif d'interagir avec une instance de la JVM. Ces primitives, regroupées dans l'API *invocation*, permettent d'embarquer une instance de la JVM dans une application native. Elles sont notamment utilisées par la commande `java`, mais elles permettent de définir des lanceurs spécifiques. Ces primitives permettent notamment de créer une nouvelle instance de la JVM, de modifier ces paramètres et de provoquer l'arrêt d'une instance. Il est également possible de « s'attacher » et se « détacher » d'une instance existante (créée par un autre *thread*).

9.1.4 Risques

JNI offre des fonctionnalités intéressantes et nécessaires au bon fonctionnement de la plate-forme d'exécution. Cette interface permet notamment aux méthodes natives des classes de la bibliothèque d'accéder à la plate-forme native. Toutefois, il s'agit de la principale déviation de Java qui est la cause d'un grand nombre de vulnérabilités des applications et des environnements d'exécution Java. En effet, la plupart des propriétés de Java ne sont plus vérifiées pour le code natif. Ce dernier souffre des problèmes intrinsèques liés à l'utilisation de langages tels que C ou C++ (gestion de la mémoire explicite, absence de vérifications dynamiques, etc.). À l'inverse du code Java, il est vulnérable aux dépassements de tampon, aux erreurs de formatage de chaîne de caractères, etc.

Le risque est d'autant plus important que le code natif implémente des opérations complexes. En cas d'exploitation des éventuelles vulnérabilités, l'attaquant dispose des mêmes privilèges que la JVM (et l'application qu'elle exécute). Si différentes instances de la JVM ne sont pas cloisonnées (par exemple, exécution sous la même identité au sens de l'OS), l'attaquant peut accéder à l'espace mémoire et aux données des autres instances (et donc d'autres applications Java).

Globalement, les risque principaux sont les suivants :

- un code natif malveillant ou piégé interfère avec l'exécution d'une application Java ;
- un code natif vulnérable manipule des données qui permettent à un attaquant d'exploiter des vulnérabilités ou renvoie à la JVM des données erronées (par exemple, des références non valides).

La manipulation de données sans vérification au préalable consitue le risque majeur du code natif appelé via JNI. Il existe notamment un risque que des données acquises par le code Java et transmises au code natif permettent d'exploiter une vulnérabilité de ce dernier. C'est notamment le cas des vulnérabilités de type débordement de tampon, car le passage d'objets de type référence depuis le code Java vers le code natif nécessite une copie pour la conversion entre les types Java et les types natifs.

Même si la JVM garde un certain contrôle sur l'espace mémoire des objets Java, elle n'a en définitive que peu de moyens de contrôler le code natif. En particulier, le contrôle d'accès de Java ne s'applique pas aux actions du code natif, même s'il est possible de contrôler les appels, depuis le code Java, aux méthodes natives [55].

9.1.5 Recommandations

En raison des risques évoqués précédemment, il est recommandé de n'utiliser JNI que dans les cas qui le nécessitent explicitement (typiquement, l'accès à une ressource native pour laquelle il n'existe pas d'interface avec Java). Le code natif doit implémenter le moins de traitement possible et s'appuyer pour cela sur le code Java qui, lui, vérifie un certain nombre de propriétés évoquées dans ce document.

Lorsque le code natif implémente des mécanismes complexes, il doit être audité en profondeur afin de s'assurer de l'absence de vulnérabilité, ce qui peut s'avérer être une tâche complexe, longue et coûteuse.

Les paramètres passés depuis le code Java vers le code natif (et l'inverse, dans une moindre mesure) doivent être systématiquement vérifiés (préférentiellement dans le code Java) afin d'éviter notamment les problèmes de dépassement de tampon.

9.2 Mécanismes d'audit dynamiques

La plate-forme d'exécution Java dispose d'interfaces standards permettant la mise au point et l'audit dynamique d'une application Java s'exécutant sur une interface de la JVM. Ces mécanismes sont les suivants :

- JVM Tool Interface (JVMTI), une interface native de bas niveau qui étend JNI et permet la mise au point des applications Java ;
- Java Plateform Debugger Architecture (JPDA), une architecture haut niveau qui comprend :
  - JVMTI,
  - Java Debug Wire Protocol (JDWP), un protocole permettant la mise au point via une application distante qui se connecte à la JVM,
  - Java Debug Interface (JDI), une interface de haut niveau permettant de développer des applications de mise au point en Java ;

55. Cela nécessite cependant d'implémenter explicitement des points de contrôle. En principe, la bibliothèque standard implémente de tels points de contrôle pour les méthodes natives qu'elle définie. Lorsque le développeur implémente ses propres méthodes natives, il doit explicitement implémenter les points de contrôle correspondants.

– Java Management Extension (JMX), une API de haut niveau permettant la gestion à distance des applications Java et de la JVM.

9.2.1 JVMTI

JVM Tool Interface (JVMTI) est une API bas niveau introduite dans la version 5 de Java SE. Elle remplace à la fois les API JVM Profiler Interface (JVMPI) et JVM Debug Interface (JVMDI) qui sont maintenant obsolètes. Elle étend les possibilités de JNI et permet le développement d'agents en C/C++ qui communiquent avec une instance de la JVM. Elles offrent des services qui permettent d'inspecter l'état interne d'une instance de la JVM et notamment l'état de l'application Java qu'elle exécute. Il est également possible d'interagir avec cette instance de la JVM afin de modifier le comportement de l'application, ce qui est une fonctionnalité nécessaire à l'implémentation d'applications de mise au point. Il est également possible de modifier le code de l'application exécutée.

L'utilisation de JVMTI suppose donc l'implémentation en C/C++ d'agents qui peuvent :

– envoyer des messages de requêtes à une instance de la JVM ;

– recevoir des messages de notifications de la JVM.

À l'inverse de JNI, cette API n'offre pas de fonctionnalité au code Java, mais uniquement au code natif des agents. Ceux-ci doivent être chargés au démarrage de l'instance de la JVM via des options de la commande `java` (`-agentlib:` et `-agentpath:`) ou des variables d'environnement.

Les possibilités qu'offrent JVMTI sont donc importantes. Cette interface permet notamment d'inspecter l'ensemble du tas d'une application Java et donc d'accéder à des champs contenant des données confidentielles. De plus, depuis Java SE 6, JVMTI offre la possibilité de modifier le code des méthodes. Le risque est d'autant plus important que le code natif n'est pas soumis au contrôle d'accès Java [56]. Cependant, le risque est limité puisqu'il est nécessaire qu'un agent soit installé, ce qui ne doit pas être le cas pour une application déployée en production.

9.2.2 JPDA

Java Platform Debugger Architecture (JPDA) est une API de haut niveau qui s'appuie sur JVMTI (la JVM doit offrir les services de JVMTI afin de pouvoir implémenter JPDA). Globalement, cette API offre les mêmes possibilités que JVMTI, mais permet de développer l'application de mise au point en Java. Elle permet également que cette application soit exécutée dans un processus distinct voire sur une machine distante, via des échanges de messages entre l'instance de la JVM observée et l'application de mise au point.

56. Cependant, le code Java invoqué depuis le code natif (appel à une méthode) est soumis au contrôle d'accès.

Les risques sont les mêmes que pour JVMTI à la différence que JPDA permet à une application tierce d'interagir avec une instance de la JVM. Toutefois, s'agissant d'un mécanisme de haut niveau en Java, il est possible d'en restreindre l'utilisation via le mécanisme de contrôle d'accès. De plus, il requiert un mécanisme de communication inter-processus (IPC), dans la majorité des cas, des *sockets*, dont l'accès peut être restreint au niveau du contrôle d'accès de l'OS.

9.2.3 JMX

Depuis Java 1.5, la JVM doit disposer d'une interface permettant d'observer et de gérer une application Java et l'instance de la JVM correspondante à distance. Ces fonctionnalités sont fournies par l'API *Java Management Extension* (JMX). À l'inverse des autres API présentées précedemment, cette API de haut niveau a pour vocation d'interagir avec des applications en production. Elle permet notamment de contrôler le chargement des classes, l'état des différents *threads*, les problèmes de synchronisation, l'utilisation de la mémoire, etc. L'instance de la JVM peut être observée à distance par un agent développé en Java.

Pour permettre à un agent d'interagir avec son fonctionnement, l'instance de la JVM doit être exécutée avec un ensemble de propriétés (passées en paramètre ou dans un fichier de configuration). Il est également possible de restreindre l'utilisation de ces fonctionnalités en utilisant un mécanisme d'authentification. De plus, il est possible de chiffrer les communications via SSL entre l'agent et l'instance de la JVM observée.

9.2.4 Risques

Globalement, le risque principal lié aux différents mécanismes présentés précédemment réside dans la possibilité, pour une application tierce (éventuellement une application Java), d'accéder aux différentes informations d'une instance de la JVM et des données de l'application qu'elle exécute. Il existe également un risque qu'une application tierce modifie le comportement d'une autre application.

9.2.5 Recommandations

Afin de se prémunir contre les risques évoqués précedemment, il convient de restreindre l'utilisation de ces mécanismes aux phases de mise au point. Il est donc recommandé de désactiver ces mécanismes lorsque l'application est déployée en production. Dans le cas de JMX, le mécanisme doit être restreint de manière à ce que seul l'agent « légitime » puisse l'utiliser (utilisation du mécanisme d'authentification).

Dans la plupart des cas, la restriction est relativement simple à mettre en œuvre : il suffit de s'assurer que les paramètres passés à la commande `java` ou les fichiers de configuration n'activent pas les mécanismes en question. Lorsque le mécanisme nécessite une communication entre un agent et une instance de la JVM, il est possible d'utiliser un mécanisme de protection fourni par l'OS (pare-feu, confinement, etc.) afin de contrôler l'utilisation de ces fonctionnalités. Pour les mécanismes utilisant du code Java, il est parfois possible d'utiliser le contrôle d'accès de Java pour en limiter l'usage.

9.3 Autres risques de déviations

Il existe également d'autres mécanismes moins connus, mais qui peuvent mener à des déviations :

- le mécanisme d'instrumentation de *bytecode* « à la volée » ;
- le mécanisme permettant l'exécution d'une commande de l'OS.

9.3.1 Instrumentation

L'instrumentation de *bytecode*, apparue dans Java 1.5, repose sur une fonctionnalité fournie par la JVM et une interface de la bibliothèque Java (*package* `java.java.lang.instrument`). Elle permet à un agent développé en Java et dont la classe principale est fournie en paramètre à l'instance d'une JVM (via un paramètre de la commande `java`) de modifier à la volée le *bytecode* de toutes les classes de l'application s'exécutant sur l'instance de la JVM. Concrètement, une fois chargé, le code de la classe est passé en paramètre (sous la forme d'un tableau d'octets) à une méthode `transform` d'une classe implémentant l'interface `ClassFileTransformer`. Le code de l'agent peut également s'appuyer sur des bibliothèques de manipulation de *bytecode* telles que BCEL [57] ou ASM [58].

Ce mécanisme présente un risque de piégeage de l'application par un agent malveillant. Toutefois, ce risque est limité puisque l'agent doit être déclaré explicitement dans les paramètres de la ligne de commande `java`. Il peut en revanche être utilisé pour instrumenter l'application ou implémenter des vérifications de sécurité supplémentaires. Toutefois, l'instrumentation de bytecode Java par un agent développé en Java génère un surcoût à l'exécution non négligeable qui peut limiter l'utilisation de ce type de solution en production.

57. `http://jakarta.apache.org/bcel/`
58. `http://asm.ow2.org/`

9.3.2 Exécution d'une commande

La classe `java.lang.Runtime` définit une interface entre l'application Java et le système natif sur lequel elle est exécutée. Une instance de cette classe est associée à chaque instance de la JVM. L'objet correspondant à l'instance sur laquelle s'exécute l'application peut être obtenu par un appel à la méthode statique `getRuntime()`. Parmi les méthodes de cette classe, la méthode `exec` permet comme son nom l'indique d'exécuter n'importe quelle commande sur le système natif. Elle founit en retour un objet de type `java.lang.Process` qui permet à l'application Java d'interagir avec le processus de la commande exécutée (obtenir sa valeur de retour, récupérer les flux d'erreur, d'entrée et de sortie, etc.).

Ce mécanisme permet à l'application Java d'interagir avec la plate-forme d'exécution native, comme dans le cas de l'utilisation de JNI. L'application Java peut donc accéder à tous les services offerts par l'OS, de la même manière qu'une application « native ». Toutefois, l'appel à cette méthode fait l'objet d'un point de contrôle d'accès Java. Il est donc possible d'en restreindre l'utilisation. De plus, le contrôle d'accès de l'OS peut limiter les commandes exécutables en fonction de l'identité sous laquelle est exécutée la commande `java`.

10 ANNEXES

10.1 Définitions des notions utilisées par le contrôle d'accès

La spécification de la politique de contrôle d'accès s'appuie dorénavant sur les concepts suivants :

- les **permissions** définissent un ensemble d'actions autorisées. Elles peuvent être assimilées à la notion de capacités implémentées sur certains OS ;
- les **origines du code** définissent l'origine des classes, en termes d'URL (par exemple `file:///home/user1/MyApp`) ou à l'aide de signatures ;
- les **domaines de protection** correspondent à l'association d'une ou plusieurs origines de code à un ensemble de permissions. Ils définissent un ensemble de classes qui vont s'exécuter dans un environnement confiné particulier. Dans le modèle initial du bac-à-sable (ou *sandbox*), l'environnement non-confiné et l'environnement confiné correspondent maintenant à deux domaines de protection distincts ;
- la **politique de sécurité** définit l'ensemble des permissions dynamiques associées aux différentes origines de code. Elle est implémentée sous la forme d'un objet de la classe `java.security.Policy`. Elle peut être spécifiée à l'aide de fichiers de configuration modifiables via un éditeur de texte ou l'outil `policytool` ;
- le **magasin de clés** (*Keystore*) contient l'ensemble des certificats des fournisseurs de confiance.

10.1.1 Permissions

Les permissions peuvent être génériques et concerner un ensemble d'objets (par exemple : « Tous les accès sont autorisés ») ou peuvent être au contraire spécifiques à un objet particulier (par exemple : « Le fichier `/home/user1/file.txt` peut être lu »). Une permission est caractérisée par les éléments suivants :

- le **type** qui correspond au type de la classe Java implémentant la permission. Chaque permission hérite de la classe abstraite `java.security.Permission`. Le type des permissions est donc hiérarchisé par la notion d'héritage. La bibliothèque standard définit un certain nombre de permissions, par exemple `java.security.AllPermission` ou `java.io.FilePermission` ;
- le **nom** de la permission qui est implémenté sous la forme d'un champ privé de type chaîne de caractères. Ce champ peut être utilisé, suivant le type, pour préciser la permission (par exemple en précisant l'objet sur lequel elle s'applique) ;
- les **actions** de la permission. La classe abstraite impose seulement d'implémenter l'interface `public abstract String getActions()`. Ce champ optionnel correspond

aux actions autorisées par la permission. Il peut dans certains cas préciser la permission, par exemple pour les permissions de type `java.io.FilePermission`.

Les permissions constituent des capacités qui peuvent être associées à chaque classe (représentée par un objet `java.lang.Class`) via les domaines de protection (le domaine de protection d'une classe peut être obtenu par la méthode `getProtectionDomain()` de la classe `java.lang.Class`). Cette association peut-être réalisée lors de deux étapes :

- lors du chargement de la classe (permissions statiques) ;
- lors de la vérification des permissions (permissions dynamiques).

La sémantique des permissions est définie par la méthode `implies` qui est utilisée par le contrôleur d'accès.

Initialement, le mécanisme supportait uniquement les permissions statiques qui sont attachées au domaine de protection de la classe par le chargeur de classes. Afin de prendre en compte les aspects contextuels de la politique [59], le mécanisme permet maintenant de définir des permissions dynamiques. Celles-ci ne sont plus définies lors du chargement de la classe, mais lors de la vérification des accès qui mettent en jeu des méthodes de la classe.

Les classes de la bibliothèque standard possèdent tous les droits (leur domaine de protection comprend la permission `java.security.AllPermission`). Lors de l'exécution de certaines méthodes sensibles (par exemple, la lecture d'un fichier), le mécanisme de contrôle d'accès de Java vérifie que l'application peut effectuer l'action réalisée par la méthode en prenant en compte les permissions de l'ensemble des classes des méthodes de la pile d'exécution. Cette vérification est initiée par des points de contrôle qui utilisent des permissions qui permettent alors d'exprimer les autorisations nécessaires pour réaliser l'accès contrôlé.

10.1.2 Origines du code

Les origines du code, implémentées par des objets de la classe `java.security.CodeSource`, permettent d'exprimer l'origine de la classe suivant deux paramètres complémentaires :

- l'URL de la classe, implémentée par un objet de la classe `java.net.URL`, permet de spécifier le chemin du fichier classe correspondant pour une classe « locale » (par exemple, `file:///home/user1/monapplication/`) ou l'adresse réseau pour une *applet* téléchargée (par exemple, `http://www.monsite.com/app/`). Ce paramètre correspond à l'attribut `CODEBASE` de la balise HTML `<APPLET>`. Il s'agit essentiellement d'une donnée d'identification, aucune preuve n'étant apportée sur l'origine. La confiance dans cette donnée repose essentiellement sur la confiance dans le chargeur de classes qui l'a spécifiée ;
- les certificats des fournisseurs de la classe, implémentés par des objets de la classe `java.security.cert.Certificate`. Ces derniers contiennent notamment les clés

59. Il s'agit essentiellement d'assurer l'intégration de JAAS qui permet de prendre en compte l'identité du sujet dans le contrôle d'accès.

publiques permettant de vérifier la signature de la classe si celle-ci utilise le mécanisme de signature. Ce paramètre permet d'identifier les classes provenant de fournisseurs « de confiance ». Il s'agit en quelque sorte d'une donnée d'authentification de la classe. La confiance dans cette donnée repose en partie sur la confiance dans le chargeur de classes, mais également sur le mécanisme de vérification des signatures des archives JAR.

10.1.3 Domaines de protection

Les domaines de protection de Java reprennent un concept implémenté sur les OS [42]. Ils permettent de regrouper l'ensemble des classes possédant les mêmes autorisations d'accès aux ressources du système, c'est-à-dire celles possédant les mêmes permissions (statiques et dynamiques). L'architecture initiale reposait sur deux domaines fixes : l'environnement non-confiné et l'environnement confiné (bac-à-sable). La nouvelle architecture s'appuie maintenant sur différents domaines qui peuvent être définis suivant les besoins du développeur ou de l'administateur. Deux types de domaines peuvent être distingués :

- le domaine système qui comprend l'ensemble des classes de la bibliothèque standard et qui possèdent toutes les autorisations (permission `java.security.AllPermission`) ;
- les domaines applicatifs qui correspondent aux classes des différentes applications et *applets* Java et dont les permissions sont définies par la politique.

Les règles de contrôle d'accès sont en général plus restrictives pour les domaines applicatifs que pour le domaine système. Ces restrictions portent, dans le modèle par défaut, sur l'accès aux ressources (objets et méthodes « sensibles ») de la plate-forme d'exécution et de l'OS. Il est possible d'envisager de restreindre les accès entre les ressources des applications des différents domaines. Cette restriction n'est pas implémentée dans les mécanismes de contrôle d'accès fournis par la bibliothèque standard. Le mécanisme de chargement de classes, présenté en section 6.3, permet en revanche de confiner les différentes applications entre elles.

Les domaines de protection sont implémentés par des instances de la classe `java.security.ProtectionDomain`. Chaque domaine comprend un objet `CodeSource` et un objet `PermissionCollection`, implémentés sous forme de champs privés et qui sont passés en paramètre au constructeur. Depuis l'intégration de l'API JAAS, présentée en section 6.2.3, cet objet comprend également un champ optionnel correspondant aux identités (*Principals*) des sujets pour le compte desquels les méthodes de la classe sont exécutées. L'utilité de ce champ est présentée en section 6.2.3, car il n'intervient que dans le contrôle d'accès relatif aux utilisateurs ou sujets.

Les domaines de protection sont essentiellement utilisés lors de deux étapes du contrôle d'accès :

- lors du chargement d'une classe, le chargeur de classes associe à l'objet `Class` représentant la classe chargée un domaine de protection en fonction du champ `CodeSource`

de ce dernier ;

- lors de l'accès à une ressource protégée, le mécanisme de mise en œuvre du contrôle d'accès détermine le domaine de protection associé à la classe à l'aide de la méthode `getProtectionDomain()`, puis vérifie l'ensemble des permissions associées à l'aide de la méthode `implies(Permission permission)` de la classe `ProtectionDomain`. Le comportement de cette méthode varie suivant le type de permissions utilisées (statiques ou dynamiques) :
  - si le domaine de protection utilise exclusivement des permissions statiques [60], la méthode effectue la vérification uniquement sur les permissions associées statiquement au domaine de protection,
  - si le domaine de protection utilise des permissions dynamiques (ainsi qu'éventuellement des permissions statiques), la méthode délègue la vérification au gestionnaire de la politique.

10.1.4 Politique de sécurité

La politique de sécurité est implémentée par un gestionnaire de politique. Par défaut [61], celui-ci est implémenté par une instance d'une classe héritant de la classe `java.security.Policy`. L'instance de la politique prise en compte par le mécanisme de contrôle d'accès du système peut être spécifiée à l'aide de la méthode statique `setPolicy` de la classe `java.security.Policy`. Une seule instance de la politique est prise en compte par le système à un instant donné.

La politique par défaut est spécifiée par le paramètre de sécurité `policy.provider`. La valeur de ce paramètre prise en compte lors du lancement de la JVM est spécifiée dans le fichier de configuration `<JAVA_HOME>/lib/security/java.security` (à titre d'exemple, l'emplacement de ce fichier de configuration sur un système Linux Debian utilisant l'implémentation de Sun du JRE est `/usr/lib/jvm/java-6-sun/jre/lib/security/java.security`).

L'implémentation de Sun du JRE utilise par défaut une instance de la classe `sun.security.provider.PolicyFile`. Cette classe permet de spécifier la politique de sécurité à l'aide de fichiers de configuration au format texte UTF8. Par défaut, la classe repose sur deux niveaux de configuration :

- une définition globale de la politique qui s'applique sur l'intégralité du système, quel que soit l'utilisateur (défini par l'OS) de l'application Java. Cette politique est spécifiée par défaut dans le fichier `<JAVA_HOME>/lib/security/java.policy` ;
- une définition de la politique propre à chaque utilisateur, qui s'applique donc uniquement aux applications exécutées par l'utilisateur en question. Cette politique est

60. Ce cas correspond à l'implémentation antérieure à la version 1.4 de Java. Il n'est plus utilisé par l'implémentation par défaut, mais est toujours supporté pour des raisons de compatibilité.

61. La configuration par défaut du contrôle d'accès suppose une délégation des vérifications du gestionnaire de sécurité au contrôleur d'accès qui lui-même s'appuie sur une instance de `java.security.Policy`.

spécifiée par défaut dans le fichier `<USER_HOME>/.java.policy`.

Ces fichiers sont modifiables à l'aide d'un éditeur de texte ou de la commande `policytool`. Il est également possible de passer en paramètre à la commande `java` un fichier de spécification de la politique. Par exemple, la commande `java -Djava.security.policy=mapolitique` permet d'utiliser une définition supplémentaire de la politique (qui complémente les définitions utilisées par défaut). La commande `java -Djava.security.policy==mapolitique` définit quant à elle la politique uniquement à partir du fichier passé en paramètre (les fichiers de configuration par défaut ne sont pas pris en compte). De manière générale, les comportements par défaut peuvent être spécifiés à l'aide de propriétés de sécurité qui sont elles-mêmes modifiables dans le fichier de configuration `java.security` ou à l'aide de paramètres de la ligne de commande `java`.

La politique de sécurité étant définie en termes d'autorisations (les permissions), la politique globale qui s'applique correspond donc à l'union des permissions définies dans les différents fichiers de politique pris en compte par le système (plus les éventuelles permissions statiques). La politique peut donc être modifiée de différentes manières :

- en modifiant un fichier de configuration de la politique utilisée par défaut ;
- en spécifiant un fichier de configuration de la politique qui complémente ou supplante les fichiers de configuration par défaut ;
- en utilisant une classe qui implémente la classe abstraite `java.security.Policy`. Cette classe peut elle-même s'appuyer sur une définition de la politique spécifiée dans des fichiers de configuration (fichiers texte, XML, etc.), dans une base de données, etc.

Ces différentes manières de modifier la politique de contrôle d'accès demeurent compatibles avec l'implémentation par défaut du mécanisme de mise en œuvre du contrôle d'accès. Il est également possible de modifier ce mécanisme de mise en œuvre et de proposer par conséquent d'autres mécanismes de spécification de la politique. Cette approche offre une plus grande liberté quant au type de mécanisme de contrôle d'accès implémenté. Elle suppose en revanche un niveau d'expertise plus conséquent. Ce type d'approche doit donc être restreint au cas ou la redéfinition est strictement nécessaire. Ce point fera l'objet de recommandations à l'intention des développeurs d'applications Java.

10.1.5 Magasin de clés

Le magasin de clés est implémenté par une instance de la classe `java.security.KeyStore` qui permet de regrouper des certificats et des clés cryptographiques. Par défaut, le JRE de Sun s'appuie sur un *keystore* au format JKS (Java Key Store) implémenté par la classe `sun.security.provider.JavaKeyStore`. Le type de *keystore* peut être modifié à l'aide du paramètre de sécurité `keystore.type` dans le fichier de configuration `<JAVA_HOME>/lib/security/java.security` (ou par passage de paramètre à la commande `java` si celui-ci est autorisé). Cette implémentation permet de stocker les certificats des fournisseurs de confiance dans un fichier. Le chemin de ce fichier est spécifié dans le fichier de configuration de la politique (par défaut,

<USER_HOME>/.keystore). Chaque entrée du *keystore* correspond à un certificat contenant la clé publique d'un fournisseur. Chaque certificat est identifié par un alias qui est notamment utilisé dans la définition de la politique de sécurité.

10.2 Architecture des *providers*

Un CSP est constitué d'un ensemble de classes Java regroupées dans une ou plusieurs archives JAR. D'un point de vue logiciel, le CSP est une classe Java qui a pour objectif de présenter et de déclarer les services qu'il offre. Cette classe est unique par CSP et hérite de `java.security.Provider`. En général, la classe représentant un CSP possède une table de hachage (classe `java.util.HashMap`) dont les entrées correspondent aux services du *provider*. Ces entrées sont formées de la manière suivante :

- Clé d'insertion :
  <*Nom de la classe haut niveau correspondant au type de service*>.<*Algorithme*> ;
- Donnée d'insertion :
  <*Nom de la classe implémentant le service*>.

Par exemple, le CSP de Sun réalise la déclaration de l'algorithme de génération de bi-clés DSA de la manière suivante : `map.put("KeyPairGenerator.DSA", "sun.security.provider.DSAKeyPairGenerator")`.

Ensuite concernant l'implémentation des services, les classes développées doivent respecter les interfaces propres au type de service implémenté. Chaque classe de haut niveau (définissant un type de service) possède une classe d'interface. Les implémentations devront donc hériter d'une de ces interfaces. Les classes d'interfaces sont les suivantes : `MessageDigestSpi`, `CipherSpi`, `SignatureSpi`, `SecureRandomSpi`, etc.

Pour mieux comprendre comment sont utilisés les services cryptographiques, il suffit d'observer le code source suivant :

```
MessageDigest md;
String message;
byte[] result;

message = new String("aaa");
md = MessageDigest.getInstance("MD5");

//Si l'on souhaite utiliser l'implementation du CSP ProviderC
//md = MessageDigest.getInstance("MD5", "ProviderC");

md.update(message.getBytes());
result = md.digest();
```

Ce code montre comment utiliser la fonction de hachage MD5. La première étape consiste à instancier le service à l'aide de l'appel `md = MessageDigest.getInstance("MD5")`. L'instanciation d'un service suit toujours le schéma suivant :

```
<Classe> service = <Classe>.getInstance("<Nom␣de␣l'algorithme>");
```

En réalisant cet appel, la liste des services de chaque *provider* est parcourue jusqu'à ce qu'une entrée correspondant au nom de l'algorithme souhaité soit trouvée. Les *providers* sont sollicités les uns après les autres en suivant un ordre de préférence. Toutefois, il est possible d'imposer l'instanciation d'un algorithme pour un *provider* donné. Pour cela, l'appel utilisé suit le schéma suivant :

```
<Classe> service = <Classe>.getInstance("<Nom␣de␣l'algorithme>",
                                        "<Nom␣du␣provider>");
```

Dans ce cas de figure, seule la liste des services du *provider* souhaité est consultée. Dans tous les cas, si un algorithme n'existe pas ou si un *provider* imposé n'existe pas, une exception est levée.

Lors de l'installation d'un environnement Java, un ou plusieurs CSP sont installés et configurés par défaut. D'autres CSP pourront être ajoutés par la suite si besoin. La déclaration des CSP se fait via le fichier de configuration `lib/security/java.security` depuis le répertoire racine de l'environnement Java. Ce fichier permet notamment de configurer l'ordre de préférence des CSP et de configurer certains paramètres de sécurité qui seront pris en compte par certains services. Les *providers* cités dans ce fichier seront chargés au démarrage de l'application.

La figure 17 illustre le fonctionnement de l'architecture de JCA-JCE.

10.2.1 Création d'un CSP

Les développeurs, s'ils le souhaitent, peuvent créer leur propre CSP et ainsi implémenter leurs propres algorithmes.

Lors de la création d'un nouveau provider, il faut dériver le nouveau *provider* de la classe `java.security.Provider`. Nous présentons ci-après un exemple de création d'un CSP.

```
import java.security.Provider;
public final class NewProvider extends Provider {

  private static final long serialVersionUID =
      -4272847733899350653L;
  private static final String INFO =
      "NEWPROVIDER␣:␣SecureRandom␣+␣SHA1";
```

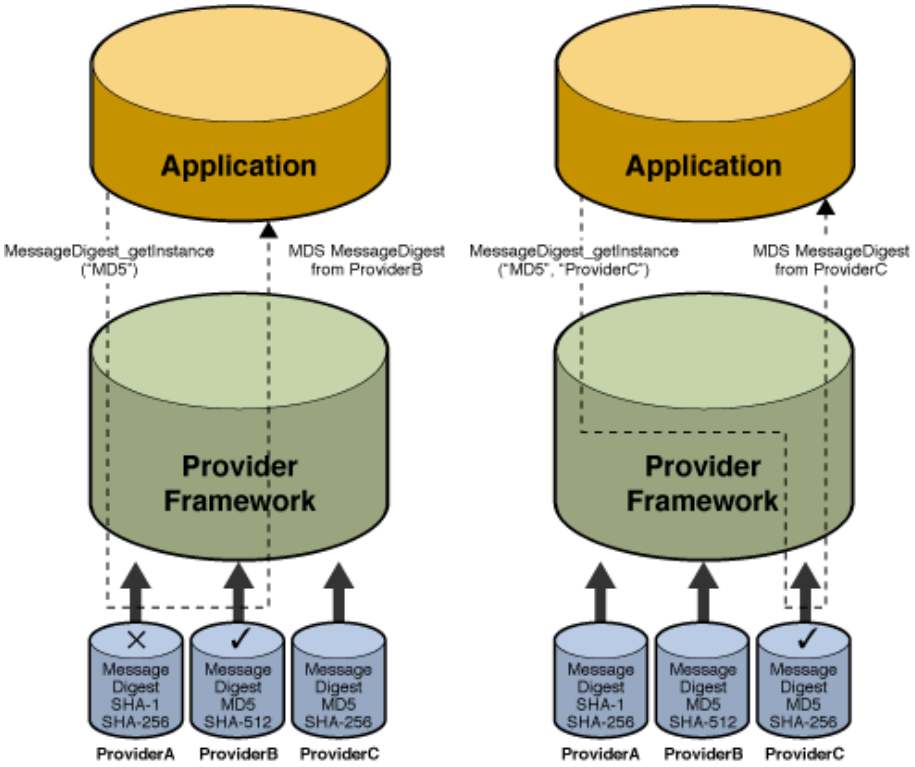


FIGURE 17 – Architecture CSP - JCE (source Sun)

```
  public NewProvider() {
    super("BADPROVIDER", 1.6, INFO);
    final Map map = (System.getSecurityManager() == null) ?
        (Map)this : new HashMap();
    map.put("SecureRandom.SHA1PRNG",
        "sun.security.new.newSecureRandom");
    map.put("MessageDigest.SHA1", "sun.security.new.NewSha");
    map.put("MessageDigest.SHA-1", "sun.security.new.NewSha");
    map.put("MessageDigest.SHA", "sun.security.new.NewSha");
    if (map != this)
      AccessController.doPrivileged(new PutAllAction(this, map));
  }
}
```

Ce nouveau CSP possède un nom (*newProvider*) ainsi que des services associés (dans l'exemple ci-dessus, un algorithme de génération de nombres aléatoires et un algorithme de signature). Il convient de préciser qu'une implémentation d'un algorithme peut être assignée à plusieurs noms (SHA1, SHA-1, SHA renvoient sur le même algorithme).

10.3 Critères d'évaluation de JCA

10.3.1 Indépendances des algorithmes

```
private SecureRandom(SecureRandomSpi secureRandomSpi,
                     Provider provider,
                     String algorithm) {
  super(0);
  this.secureRandomSpi = secureRandomSpi;
  this.provider = provider;
  this.algorithm = algorithm;
}
```

Dans cet exemple, l'argument `algorithm` correspond à l'algorithme utilisé pour le service.

L'API JCA n'implémente pas d'algorithme. L'API utilise les implémentations d'algorithmes provenant d'un CSP via l'utilisation de la méthode `getInstance`. L'exemple ci-dessous (issu des sources de la classe `java.security.SecureRandom`) présente une implémentation de la méthode `getInstance` pour un service de génération de nombres aléatoires :

```
public static SecureRandom getInstance(String algorithm,
    String provider) throws NoSuchAlgorithmException,
        NoSuchProviderException {
  Instance instance = GetInstance.getInstance("SecureRandom",
      SecureRandomSpi.class, algorithm, provider);
  return new SecureRandom((SecureRandomSpi)instance.impl,
      instance.provider, algorithm);
}
```

Comme le montre le code précédent, la méthode `getInstance` prend en paramètres un algorithme et un provider et renvoie un objet de type `SecureRandom`.

10.3.2 Exemple d'utilisation de ces conventions de nommage et commentaires

```
/**
  * Generates an integer containing the user-specified number of
  * pseudo-random bits (right justified, with leading zeros).
  * This method overrides a <code>java.util.Random</code> method,
  * and serves to provide a source of random bits to all of the
  * methods inherited from that class (for example,
  * <code>nextInt</code>, <code>nextLong</code>, and
  * <code>nextFloat</code>).
  *
  * @param numBits number of pseudo-random bits to be generated,
```

```
   * where 0 <= <code>numBits</code> <= 32.
   *
   * @return an <code>int</code> containing the user-specified
   * number of pseudo-random bits (right justified,
   * with leading zeros)
   */
final protected int next(int numBits) {
  int numBytes = (numBits+7)/8;
  byte b[] = new byte[numBytes];
  int next = 0;
  nextBytes(b);
  for (int i = 0; i < numBytes; i++)
    next = (next << 8) + (b[i] \& 0xFF);
  return next >>> (numBytes*8 - numBits);
}
```

Les commentaires informent du service rendu par la fonction en indiquant ses données d'entrées et ce qu'elle retourne. Les paramètres et variables utilisés sont compréhensibles et facilement utilisables. L'utilisation de l'API JCA est intuitive pour un développeur Java. Le guide d'utilisation de l'API est très clair et propose des exemples concrets et simples d'approche.

10.3.3 Code source de java.security.SecureRandom

```
/**
   * Constructs a secure random number generator (RNG) implementing
   * the default random number algorithm.
   * The SecureRandom instance is seeded with the specified
   * seed bytes.
   *
   * <p> This constructor traverses the list of registered security
   * Providers, starting with the most preferred Provider.
   * A new SecureRandom object encapsulating the
   * SecureRandomSpi implementation from the first
   * Provider that supports a SecureRandom (RNG) algorithm is
   * returned.
   * If none of the Providers support a RNG algorithm,
   * then an implementation-specific default is returned.
   *
   * <p> Note that the list of registered providers may be
   * retrieved via the {@link Security\#getProviders() Security.
   * getProviders()} method.
   *
   * <p> See Appendix A in the <a href=
   * "../../../technotes/guides/security/crypto/CryptoSpec.html">
   * Java Cryptography Architecture API Specification \&amp;
   * Reference </a>
   * for information about standard RNG algorithm names.
```

```
 *
 * @param seed the seed.
 */
public SecureRandom(byte seed[]) {
  super(0);
  getDefaultPRNG(true, seed);
}

private void getDefaultPRNG(boolean setSeed, byte[] seed) {
  String prng = getPrngAlgorithm();
  if (prng == null) {
    // bummer, get the SUN implementation
    prng = "SHA1PRNG";
    this.secureRandomSpi =new sun.security.provider.SecureRandom();
    this.provider = new sun.security.provider.Sun();
    if (setSeed)
      this.secureRandomSpi.engineSetSeed(seed);}
    else {
      try {
        SecureRandom random = SecureRandom.getInstance(prng);
        this.secureRandomSpi = random.getSecureRandomSpi();
        this.provider = random.getProvider();
        if (setSeed) {
          this.secureRandomSpi.engineSetSeed(seed);}
        }
    }
    catch (NoSuchAlgorithmException nsae) {
      // never happens, because we made sure the algorithm exists
      throw new RuntimeException(nsae);
  }
}
```

10.4 Description pseudo-formelle du générateur de pseudo-aléa de Sun

En prenant pour i variant de 0 à 19, prenons les conventions suivantes :

- Soit $S_t[i]$ l'octet de rang i de l'état interne ;
- Soit $R_t[i]$ l'octet de rang i de la sortie ;
- Soit $T[i]$ l'entier de rang i de la variable temporaire T ;
- Soit $\delta[i]$ l'entier de rang i de la variable temporaire δ.

La description pseudo-formelle de l'algorithme devient :

Algorithme : Loi d'évolution du générateur de pseudo-aléa de Sun

Entrées : $\delta[0] = 1$, l'état interne S_t et la sortie R_t au temps t

Pour $i = 0$ **à** 19 **Faire**

$$T[i] = S_t[i] + R_t[i] + \delta[i]$$

$$S_{t+1}[i] = T[i] \bmod 256$$

$$\delta[i+1] = \frac{1}{256} T[i]$$

Fin Pour

Sortie : $R_{t+1} = \text{SHA-1}(S_{t+1})$

Avec :

$$R_0 = \text{SHA-1}(S_0)$$

$$S_0 = \text{SHA-1}(seed)$$

10.5 Code source du générateur d'aléa SHA1PRNG de Sun

L'accès aux spécifications du générateur d'aléa est réalisé via une revue du code source.

```
/**
 * <p>This class provides a crytpographically strong pseudo-random
 * number generator based on the SHA-1 hash algorithm.
 *
 * <p>Note that if a seed is not provided, we attempt to provide
 * sufficient seed bytes to completely randomize the internal
 * state of the generator (20 bytes). However, our seed
 * generation algorithm has not been thoroughly studied or
 * widely deployed.
 *
 * <p>Also note that when a random object is deserialized,
 * <a href="#engineNextBytes(byte[])">engineNextBytes</a> invoked
 * on the restored random object will yield the exact same (random)
 * bytes as the original object. If this behaviour is not desired,
 * the restored random object should be seeded, using
 * <a href="#engineSetSeed(byte[])">engineSetSeed</a>.
 *
 * @author Benjamin Renaud
 * @author Josh Bloch
 * @author Gadi Guy
 */
```

Le code source ci-après a servi à rétro-spécifier ce générateur de pseudo-aléa.

```
/**
```

```
     * Reseeds this random object. The given seed supplements,
     * rather than replaces, the existing seed. Thus, repeated
     * calls are guaranteed never to reduce randomness.
     *
     * @param seed the seed.
     */
    synchronized public void engineSetSeed(byte[] seed) {
      if (state != null) {
        digest.update(state);
        for (int i = 0; i < state.length; i++)
          state[i] = 0;
      }
      state = digest.digest(seed);
    }

    private static void updateState(byte[] state, byte[] output) {
      int last = 1;
      int v = 0;
      byte t = 0;
      boolean zf = false;

      // state(n + 1) = (state(n) + output(n) + 1) % 2160;
      for (int i = 0; i < state.length; i++) {
        // Add two bytes
        v = (int)state[i] + (int)output[i] + last;
        // Result is lower 8 bits
        t = (byte)v;
        // Store result. Check for state collision.
        zf = zf | (state[i] != t);
        state[i] = t;
        // High 8 bits are carry. Store for next iteration.
        last = v >> 8;
      }

      // Make sure at least one bit changes!
      if (!zf)
        state[0]++;
    }

    /**
     * Generates a user-specified number of random bytes.
     *
     * @param bytes the array to be filled in with random bytes.
     */
    public synchronized void engineNextBytes(byte[] result) {
      int index = 0;
      int todo;
      byte[] output = remainder;
```

```
  if (state == null) {
    if (seeder == null) {
      seeder = new SecureRandom(SeedGenerator.getSystemEntropy());
      seeder.engineSetSeed(engineGenerateSeed(DIGEST_SIZE));
    }
    byte[] seed = new byte[DIGEST_SIZE];
    seeder.engineNextBytes(seed);
    state = digest.digest(seed);
  }

  // Use remainder from last time
  int r = remCount;
  if (r > 0) {
    // How many bytes?
    todo = (result.length - index) < (DIGEST_SIZE - r) ?
          (result.length - index) : (DIGEST_SIZE - r);
    // Copy the bytes, zero the buffer
    for (int i = 0; i < todo; i++) {
      result[i] = output[r];
      output[r++] = 0;
    }
    remCount += todo;
    index += todo;
  }

  // If we need more bytes, make them.
  while (index < result.length) {
    // Step the state
    digest.update(state);
    output = digest.digest();
    updateState(state, output);

    // How many bytes?
    todo = (result.length - index) > DIGEST_SIZE ?
      DIGEST_SIZE : result.length - index;
    // Copy the bytes, zero the buffer
    for (int i = 0; i < todo; i++) {
      result[index++] = output[i];
      output[i] = 0;
    }
    remCount += todo;
  }

  // Store remainder for next time
  remainder = output;
  remCount %= DIGEST_SIZE;
}
```

10.6 Code source du générateur d'aléa Windows-PRNG du *provider* SunMSCAPI

La fonction native utilisée est la suivante :

```
JNIEXPORT jbyteArray JNICALL
  Java_sun_security_mscapi_PRNG_generateSeed (
    JNIEnv *env, jclass clazz, jint length, jbyteArray seed)
```

La méthode de liaison côté Java est la suivante :

```
private static native byte[] generateSeed(int length, byte[] seed);
```

L'exemple suivant de code source Java illustre les appels au code natif :

```
protected void engineSetSeed(byte[] seed) {
  if (seed != null) {
    generateSeed(-1, seed);
  }
}

protected void engineNextBytes(byte[] bytes) {
  if (bytes != null) {
    if (generateSeed(0, bytes) == null) {
      throw new ProviderException("Error␣generating␣random␣bytes");
    }
  }
}

protected byte[] engineGenerateSeed(int numBytes) {
  byte[] seed = generateSeed(numBytes, null);

  if (seed == null) {
    throw new ProviderException("Error␣generating␣seed␣bytes");
  }
  return seed;
}
```

Dans tous les cas, un appel à `generateSeed` se traduit par un appel à l'API Windows :

```
BOOL WINAPI CryptGenRandom(__in HCRYPTPROV hProv, __in DWORD dwLen,
     __inout BYTE *pbBuffer).
```

La documentation de cette fonction est disponible sur le site Web de Microsoft [62]. La sortie de cette fonction ne fait pas l'objet de retraitement algorithmique et est directement remontée à l'application Java.

62. http://msdn.microsoft.com/en-us/library/aa379942(VS.85).aspx

10.7 Paramètres DSA du générateur de clés du *provider* Sun

```
/*
* We support precomputed parameter for 512, 768 and 1024 bit
* moduli. In this file we provide both the seed and counter
* value of the generation process for each of these seeds,
* for validation purposes. We also include the test vectors
* from the DSA specification, FIPS 186, and the FIPS 186
* Change No 1, which updates the test vector using SHA-1
* instead of SHA (for both the G function and the message
* hash.
*/

/*
* L = 512
* SEED = b869c82b35d70e1b1ff91b28e37a62ecdc34409b
* counter = 123
*/
BigInteger p512 =
new BigInteger("fca682ce8e12caba26efccf7110e526db078b05edecb" +
         "cd1eb4a208f3ae1617ae01f35b91a47e6df63413c5e1" +
         "2ed0899bcd132acd50d99151bdc43ee737592e17", 16);

BigInteger q512 =
new BigInteger("962eddcc369cba8ebb260ee6b6a126d9346e38c5", 16);

BigInteger g512 =
new BigInteger("678471b27a9cf44ee91a49c5147db1a9aaf244f05a43" +
         "4d6486931d2d14271b9e35030b71fd73da179069b32e" +
         "2935630e1c2062354d0da20a6c416e50be794ca4", 16);

/*
* L = 768
* SEED = 77d0f8c4dad15eb8c4f2f8d6726cefd96d5bb399
* counter = 263
*/
BigInteger p768 =
new BigInteger("e9e642599d355f37c97ffd3567120b8e25c9cd43e" +
         "927b3a9670fbec5d890141922d2c3b3ad24800937" +
         "99869d1e846aab49fab0ad26d2ce6a22219d470bc" +
         "e7d777d4a21fbe9c270b57f607002f3cef8393694" +
         "cf45ee3688c11a8c56ab127a3daf", 16);

BigInteger q768 =
new BigInteger("9cdbd84c9f1ac2f38d0f80f42ab952e7338bf511",
         16);

BigInteger g768 =
```

```
new BigInteger("30470ad5a005fb14ce2d9dcd87e38bc7d1b1c5fac" +
         "baecbe95f190aa7a31d23c4dbbcbe06174544401a" +
         "5b2c020965d8c2bd2171d3668445771f74ba084d2" +
         "029d83c1c158547f3a9f1a2715be23d51ae4d3e5a" +
         "1f6a7064f316933a346d3f529252", 16);

/*
* L = 1024
* SEED = 8d5155894229d5e689ee01e6018a237e2cae64cd
* counter = 92
*/
BigInteger p1024 =
new BigInteger("fd7f53811d75122952df4a9c2eece4e7f611b7523c" +
         "ef4400c31e3f80b6512669455d402251fb593d8d58" +
         "fabfc5f5ba30f6cb9b556cd7813b801d346ff26660" +
         "b76b9950a5a49f9fe8047b1022c24fbba9d7feb7c6" +
         "1bf83b57e7c6a8a6150f04fb83f6d3c51ec3023554" +
         "135a169132f675f3ae2b61d72aeff22203199dd148" +
         "01c7", 16);

BigInteger q1024 =
new BigInteger("9760508f15230bccb292b982a2eb840bf0581cf5",
         16);

BigInteger g1024 =
new BigInteger("f7e1a085d69b3ddecbbcab5c36b857b97994afbbfa" +
         "3aea82f9574c0b3d0782675159578ebad4594fe671" +
         "07108180b449167123e84c281613b7cf09328cc8a6" +
         "e13c167a8b547c8d28e0a3ae1e2bb3a675916ea37f" +
         "0bfa213562f1fb627a01243bcca4f1bea8519089a8" +
         "83dfe15ae59f06928b665e807b552564014c3bfecf" +
         "492a", 16);

dsaCache.put(Integer.valueOf(512),
          new DSAParameterSpec(p512, q512, g512));
dsaCache.put(Integer.valueOf(768),
          new DSAParameterSpec(p768, q768, g768));
dsaCache.put(Integer.valueOf(1024),
          new DSAParameterSpec(p1024, q1024, g1024));

// use DSA parameters for DH as well
dhCache.put(Integer.valueOf(512), new DHParameterSpec(p512, g512));
dhCache.put(Integer.valueOf(768), new DHParameterSpec(p768, g768));
dhCache.put(Integer.valueOf(1024),
   new DHParameterSpec(p1024, g1024));
```

Le développeur préférera le code suivant :

```
KeyPairGenerator gen;
KeyPair keys;
DSAParameterSpec spec;

gen = KeyPairGenerator.getInstance("DSA", "SUN");
spec =
  ParameterCache.getNewDSAParameterSpec(1024, new SecureRandom());
gen.initialize(spec);
keys = gen.generateKeyPair();
```

10.8 Détail des packages de l'API Bouncycastle

JCE Utility and Extension Packages :

- `org.bouncycastle.jce` : classes d'outils à utiliser avec JCE ;
- `org.bouncycastle.jce.examples` : exemples de classes pour JCE ;
- `org.bouncycastle.jce.exception` ;
- `org.bouncycastle.jce.interfaces` : interfaces de support pour les attributs des clés de courbes elliptiques, d'El Gamal et de PKCS#12 ;
- `org.bouncycastle.jce.netscape` ;
- `org.bouncycastle.jce.spec` : spécification des paramètres pour El Gamal et courbes elliptiques.

OCSP and OpenSSL PEM Support Packages :

- `org.bouncycastle.ocsp` : classes pour la gestion des OCSP - RFC 2560 ;
- `org.bouncycastle.ocsp.test` : classes de test pour les messages OCSP ;
- `org.bouncycastle.openssl` : classes pour la gestion des fichiers PEM avec OpenSSL ;
- `org.bouncycastle.openssl.test` : classes de test pour OpenSSL.

ASN.1 Support Packages (33 classes) :

- `org.bouncycastle.asn1` : bibliothèque permettant d'analyser la structure et l'écriture d'objet ASN.1 ;
- `org.bouncycastle.asn1.cmp` ;
- `org.bouncycastle.asn1.cms` ;
- `org.bouncycastle.asn1.crmf` ;
- `org.bouncycastle.asn1.ocsp` ;
- `org.bouncycastle.asn1.smime` ;
- `org.bouncycastle.asn1.x509` : classes permettant d'encoder et de traiter des certificats X.509 ;

Lightweight Crypto Packages :

- `org.bouncycastle.crypto` : classes de base du *package* ;
- `org.bouncycastle.crypto.agreement` : classes de négociation de clés ;
- `org.bouncycastle.crypto.agreement.kdf` ;
- `org.bouncycastle.crypto.digests` ;
- `org.bouncycastle.crypto.encodings` ;
- `org.bouncycastle.crypto.engines` ;
- `org.bouncycastle.crypto.examples` : exemples d'utilisation des classes cryptographiques ;
- `org.bouncycastle.crypto.generators` : générateur de clés, de bi-clés, de mots de passe ;
- `org.bouncycastle.crypto.io` ;
- `org.bouncycastle.crypto.macs` : classes de génération de hachage ;
- `org.bouncycastle.crypto.modes` ;
- `org.bouncycastle.crypto.paddings` ;
- `org.bouncycastle.crypto.params` ;
- `org.bouncycastle.crypto.prng` : générateur de nombres pseudo-aléatoires ;
- `org.bouncycastle.crypto.signers` : générateur de signatures ;
- `org.bouncycastle.crypto.tls` : classes pour utiliser le protocole TLS ;
- `org.bouncycastle.crypto.util` : boite à outils pour les différents algorithmes (conversions, ...).

Utility Packages :

- `org.bouncycastle.util` ;
- `org.bouncycastle.util.encoders` : classes pour la génération et la lecture de données en Base64 et Hex ;
- `org.bouncycastle.util.io` ;
- `org.bouncycastle.util.test` : classes de test.

JCE Provider and Test Classes :

- `org.bouncycastle.jce.provider` ;
- `org.bouncycastle.jce.provider.asymmetric` ;
- `org.bouncycastle.jce.provider.asymmetric.ec` ;
- `org.bouncycastle.jce.provider.symmetric` ;
- `org.bouncycastle.jce.provider.test` ;
- `org.bouncycastle.jce.provider.test.nist` ;
- `org.bouncycastle.jce.provider.test.rsa3`.

Other Packages :

- `org.bouncycastle.mozilla` : classes pour la gestion des clés publiques signées de Mozilla et les divers challenges ;
- `org.bouncycastle.x509` : classes de génération et de gestion de certificats X.509 ;
- `org.bouncycastle.x509.examples` : classes d'exemples de génération et de gestion de certificats X.509 ;
- `org.bouncycastle.x509.extension` ;
- `org.bouncycastle.x509.util`.

RÉFÉRENCES

[1] D. Ghindici and G. Grimaud and I. Simplot-Ryl . Embedding Verifiable Information Flow Analysis . In *Proc. Annual Conference on Privacy, Security and Trust* , pages 343–352 , 2006 .

[2] A. Askarov and A. Sabelfeld. Security-typed Languages for Implementation of Cryptographic Protocols : a Case Study. In *Proc. of ESORICS'05*, LNCS, pages 197–221. Springer Verlag, 2005.

[3] A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *Proc. of SOSP'97*, pages 129–142. ACM, 1997.

[4] A. C. Myers and L. Zheng and S. Zdancewic and S. Chong and N. Nystrom. Jif : Java information flow (software release). Technical report, `http://www.cs.cornell.edu/jif`, 2001.

[5] Antony Edwards and Trent Jaeger and Xiaolan Zhang. Runtime verification of authorization hook placement for the linux security modules framework. In *CCS '02 : Proceedings of the 9th ACM conference on Computer and communications security*, pages 225–234, New York, NY, USA, 2002. ACM Press.

[6] Bertelsen, P. Dynamic semantics of Java bytecode. *Future Gener. Comput. Syst.*, 16(7) :841–850, 2000.

[7] C. Colby and P. Lee and G. C. Necula and F. Blau and M. Plesko and K. Cline. A certifying compiler for Java. In *In Proc. of PLDI'00*, volume 35 of *ACM Sigplan Notices*, pages 95–107. ACM Press, 2000.

[8] Cohen, R. M. An Operational Semantics of Java. Technical report, Computational Logic Inc, Austin, Texas, 1997.

[9] Consortium JAVASEC. Cible de sécurité JVM. Technical Report Livrable 2.4 dans le CCTP JAVASEC, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique - SGDN, 2009.

[10] Consortium JAVASEC. Comparatif des compilateurs. Technical Report Livrable 2.1 dans le CCTP JAVASEC, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique - SGDN, 2009.

[11] Consortium JAVASEC. Comparatif des JVM. Technical Report Livrable 2.2 dans le CCTP JAVASEC, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique - SGDN, 2009.

[12] Consortium JAVASEC. Guide de déploiement et de configuration d'une JVM sous Linux. Technical Report Non identifié dans le CCTP, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique - SGDN, 2009.

[13] Consortium JAVASEC. Guide de développement. Technical Report Livrable 1.3 dans le CCTP JAVASEC, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique - SGDN, 2009.

[14] Consortium JAVASEC. Rapport sur les modèles d'exécution Java. Technical Report Livrable 1.2 dans le CCTP JAVASEC, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique - SGDN, 2009.

[15] Consortium JAVASEC. Stratégie d'évolution et d'implémentation d'une JVM pour l'exécution d'applications de sécurité. Technical Report Livrable 2.3 dans le CCTP JAVASEC, Silicom Région Ouest - Amossys - INRIA Rennes Bretagne Atlantique - SGDN, 2009.

[16] D. Pichardie. Bicolano – Byte Code Language in Coq. `http://mobius.inria.fr/bicolano`. Summary appears in Deliverable 3.1 : Bytecode Specification Language and Program Logic, `http://mobius.inria.fr`, 2006.

[17] David Hovemeyer and Jaime Spacco and William Pugh. Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs. *SIGSOFT Software Engineering Notes*, 31(1) :13–19, 2006.

[18] Don Syme. Proving Java Type Soundness. In *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 83–118. Springer, 1999.

[19] E. Albert and G. Puebla and M. V. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'05*, number 3452 in LNCS, pages 380–397. Springer Verlag, 2005.

[20] E. Rose. Lightweight Bytecode Verification. *Journal of Automated Reasoning*, 31(3-4) :303–334, 2003.

[21] F. Besson and T. Jensen and D. Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science*, 364(3) :273–291, 2006.

[22] F. Besson and T. Jensen and D. Pichardie and T. Turpin. Result Certification for Relational Program Analysis. Research Report 6333, IRISA, 2007.

[23] FindBugs. `http://findbugs.sourceforge.net/`.

[24] Freund, Stephen N. and Mitchell, John C. A Type System for the Java Bytecode Language and Verifier. *J. Autom. Reason.*, 30(3-4) :271–321, 2003.

[25] G. Barthe and D. Pichardie and T. Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. In *Proc. of ESOP'07*, LNCS, pages 125–140. Springer Verlag, 2007.

[26] G. C. Necula. Proof-Carrying Code. In *Proc. of POPL'97*, pages 106–119. ACM, 1997.

[27] G. C. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proc. of OSDI'96*, pages 229–243. USENIX Assoc., 1996.

[28] G. Morrisett and D. Walker and K. Crary and N. Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3) :527–568, 1999.

[29] Gerwin Klein and Tobias Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler. *TOPLAS*, 28(4) :619–695, 2006.

[30] J. Whaley and D. Avots and M. Carbin and Monica S. Lam. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Proc. of the 3d Asian Symposium on Programming Languages and Systems*, volume 3780 of *LNCS*, pages 97–118, 2005.

[31] J. Whaley and M. S. Lam. Cloning-based Context-Sensitive Pointer Alias Analysis using Binary Decision Diagrams. In *Proc. of the ACM conference on Programming language design and implementation (PLDI '04)*, pages 131–144. ACM, 2004.

[32] Jaeger,, Trent and Edwards,, Antony and Zhang,, Xiaolan. Consistency analysis of authorization hook placement in the Linux security modules framework. *ACM Trans. Inf. Syst. Secur.*, 7(2) :175–205, 2004.

[33] James Gosling and Bill Joy and Guy Steele and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[34] Joshua Bloch. *Effective Java, 2nd edition*. Addison Wesley, 2008.

[35] L. Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.

[36] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *LNCS 3780*, pages 139–160),, November 2005.

[37] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76 :138–164, 1988.

[38] M. Martin and B. Livshits and M. S. Lam. Finding Application Errors and Security Flaws using PQL : a Program Query Language. In *Proc. of OOPSLA '05*, pages 365–383, 2005.

[39] M. Naik. *Effective Static Data Race Detection For Java*. PhD thesis, Standford University, 2008.

[40] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Proc. of POPL '07*, pages 327–338, New York, NY, USA, 2007. ACM Press.

[41] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proc. of PLDI '06*, pages 308–319, New York, NY, USA, 2006. ACM Press.

[42] Saltzer, J.H. and Schroeder, M.D. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9) :1278–1308, Sept. 1975.

[43] Sophia Drossopoulou and Susan Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 41–82. Springer, 1999.

[44] Stärk, Robert F. and Börger, E. and Schmid, Joachim. *Java and the Java Virtual Machine : Definition, Verification, Validation with Cdrom*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[45] Stephen Freund. The Costs and Benefits of Java Bytecode Subroutines. In *In Formal Underpinnings of Java Workshop at OOPSLA*, 1998.

[46] Stephenson, K. Towards an Algebraic Specification of the Java Virtual Machine. In *Proceedings of the ESPRIT Working Group 8533 on Prospects for Hardware Foundations*, pages 236–277. Springer-Verlag, 1998.

[47] Sun Microsystems. Secure Coding Guidelines Version 2.0 for the Java Programming Language. `http://java.sun.com/security/seccodeguide.html`.

[48] Sun Microsystems. JSR 133 : Java Memory Model and Thread Specification Revision. `http://jcp.org/en/jsr/detail?id=133`, 2004.

[49] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[50] V. B. Livshits and M. S. Lam. Finding Security Errors in Java Programs with Static Analysis. In *Proc. of the 14th Usenix Security Symposium*, pages 271–286, 2005.

[51] Zhenyu Qian. A formal specification of Java Virtual Machine instructions for objects, methods and subroutines. In *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 271–311, 1999.