- *Genèse de la programmation objet*
- *Spécifications de Java*
- *Evolutions de Java*
- *La machine virtuelle Java*
- *Le ramasse miettes*
- *Comment développer des applications*

1

Introduction

Java est un langage de programmation orienté objet. Un de plus ? Oui, mais avec des atouts que nul autre langage auparavant n'avait réussi à cumuler.

Au cours de son évolution, Java est devenu plus qu'un simple langage, c'est maintenant le synonyme de nouvelles architectures logicielles, et même matérielles.

Venu au monde au début des années quatre-vingt dix, adolescent au moment de la révolution internet, Java a maintenant atteint l'âge adulte, mais sa jeunesse nous laisse présager encore de nombreuses évolutions à venir.

Objectifs

- Connaître l'histoire de Java et des principaux langages objet
- Comprendre les spécifications de Java
- Découvrir les principales évolutions du langage
- Savoir comment fonctionne l'environnement d'exécution
- Parler des principaux environnements de développement
- Connaître les differentes éditions de Java et leurs particularités

Evolutions des langages

Erreur ! Liaison incorrecte.

L'objet n'est pas une nouveauté en soi dans l'ingénierie informatique. On trouve ses origines dès la fin des années soixante, avec le langage Simula.

Il a fallu de nombreuses années avant que ces concepts n'arrivent dans le cœur de l'industrie. Cela est principalement dû au fait que l'objet n'est pas une simple technologie, c'est une véritable philosophie, qui va considérablement révolutionner la façon de programmer, mais aussi, en amont, la façon d'analyser et même de penser les architectures logicielles.

Le premier langage objet à entrer de façon notable dans l'industrie est Smalltalk. On retrouve dans ce langage, qui date déjà des années soixante-dix, de nombreuses notions qui ont été récupérées par Java, comme par exemple l'idée de la Machine Virtuelle ou du Ramasse Miettes de Mémoire, que nous verrons ensemble un peu plus loin dans ce chapitre.

Mais le succès de ce langage a été très mitigé, plutôt universitaire mais boudé par l'industrie.

La grande vague de l'objet est arrivée plus tard, au début des années quatre-vingt-dix avec le langage C++.

C++ est une évolution du C, c'est à dire que ce n'est rien d'autre qu'un C auquel on a ajouté les opérateurs et les instructions nécessaires au développement objet.

Le C++ a conservé toutes les "possibilités dangereuses" du C, comme par exemple l'accès direct à la mémoire, ce qui n'est pas forcément compatible avec l'objet, qui cherche au contraire à s'isoler le plus possible des couches basses de la machine.

Au milieu des années quatre-vingt-dix, la société Sun Microsystems, constructeur d'équipements informatiques et éditeur d'un système d'exploitation UNIX a mis Java sur le marché. Ce langage apparaissait comme proche du C++, mais mieux conçu, plus moderne et surtout apportant des nouveautés que n'avait pas le C++.

Java a été inventé par James Gosling de la société Sun. Encore aujourd'hui, la marque Java appartient à cette société qui décide à elle seule de la normalisation et de l'évolution de ce langage.

Java : un simple langage de programmation de plus ?

La question que l'on était en droit de se poser était : Pourquoi encore un nouveau langage de programmation ?

Certes, Java est un langage de programmation très performant et puissant, mais cela ne suffisait pas à justifier le succès considérable, et son accueil chaleureux quasi unanime dans l'industrie.

Un certain nombre de facteurs ont facilité son ascension à sa place actuelle de leader, notamment une de ses particularité : **La portabilité**.

Cette propriété a été l'occasion pour Sun, son promoteur, de lancer un fameux chien dans le jeu de quilles Microsoft. Nous connaissons tous le slogan : "Write once, run anywhere" : Vous développez votre application une seule fois, et elle tourne partout.

Cette idée était peut être la clé de la fin de l'hégémonie Microsoft Windows.

Il en fut autrement, mais dès lors, toute la stratégie technologique de Microsoft consista à torpiller ce langage à tout prix. Et on peut dire aujourd'hui que Microsoft a échoué…

Java : un hasard de l'industrie

Dès le début des années quatre-vingt dix, Sun envisage de réaliser une plate-forme de développement pour les dispositifs d'informatique embarquée.

A cette époque, l'émergence du multimédia et de la domotique laissait envisager des équipements électroménagers très sophistiqués (terminaux de télévisions numériques, vidéo à la demande, télésurveillance, ordinateurs de bord des véhicules, etc.)

Sun, fabriquant de matériels informatiques, avait bien l'intention de s'y positionner avec une offre puissante : Une plateforme universelle de développement d'applications multimédias.

Le projet Oak était né.

L'objectif était de mettre en place un langage de programmation orienté objet simple et concis, et un architecture d'implémentation favorisant une portabilité maximale des applications, en effet, le domaine de l'informatique embarquée est une référence en matière d'hétérogénéité…

Le choix des concepteurs de Java fut purement pratique, je dirais même démagogique, et peut se résumer en deux points :

- Le langage C est de loin le plus répandu dans le monde de l'électronique, Oak va donc s'approprier sa syntaxe
- Il y a plein de bonnes choses dans SmallTalk, on va les récupérer pratiquement dans l'état, notamment la notion de machine virtuelle et celle de Garbage Collector (Ramasse Miettes) dont nous reparlerons plus loin.

Et puis il y a un certain nombre de contraintes qui seront exprimées dans ses spécifications.

Spécifications de Java

Erreur ! Liaison incorrecte.

Les contraintes très particulières au monde de l'informatique embarquée ont amené les inventeurs du langage à se fixer des règles très strictes.

Simplicité

Java est un langage simple, parce que bien conçu pour un langage pleinement objet. Cela facilite la formation du personnel, mais aussi la créativité, souvent bridée par la complexité des interfaces de programmation.

A cette époque, les bons développeurs Windows étaient très rares, donc chers. En effet, la difficulté de mise en œuvre d'une application dans cet environnement graphique nécessitait d'excellents programmeurs.

Java vient du C et du C++, ce qui permet aux nombreux développeurs pratiquant ces langages de se former rapidement et facilement.

Toutefois, un certain nombre d'éléments n'ont pas été conservés, dans un souci de simplicité, mais aussi de fiabilité comme nous le verrons plus loin :

- L'arithmétique sur les pointeurs n'existe plus. Dans les programmes informatiques, les variables des programmes sont dans la mémoire centrale de l'ordinateur, on y accède par une adresse mémoire. Cette adresse est un nombre entier, que l'on pouvait manipuler en C ou C++ pour accéder aux variables voisines (tableaux de variables). Cette opération est dangereuse, car rien n'empêche, si la valeur de l'adresse est erronée, d'accéder aux variables d'un autre programme, ou d'écraser une partie de la mémoire par erreur. Dans un souci de simplicité, mais aussi de fiabilité, cette possibilité n'existe plus en Java. Les variables sont référencées par un nombre qui n'est pas manipulable.
- Pas d'héritage multiple. Nous reparlerons dans notre prochain chapitre de cette notion.
- Pas de surcharge des types primitifs et des opérateurs. Contrairement à d'autres langages, il n'est pas possible de redéfinir ce qu'est un entier, un nombre flottant, une chaîne de caractères, une addition, une division entière, etc.

Enfin, un mécanisme de gestion automatique de la mémoire est en charge de la libération de la mémoire non utilisée.

Une application informatique orientée objet est un ensemble d'objets instanciés en mémoire, puis supprimés de la mémoire lorsqu'ils ne sont plus utiles. Cette fonction de nettoyage peut devenir très contraignante lorsque le nombre d'objets est important. Libérer le programmeur de la lourde tâche de supprimer les objets inutiles est un apport considérable dans la simplicité des programmes.

Orienté objet

Java est orienté objet, c'est fondamental. Il permet notamment l'encapsulation du code dans des classes, ce qui facilite l'implémentation d'applications analysées par la décomposition du problème en objets.

L'objet permet aussi la réutilisation du code (classes développées pour une applications, réutilisées pour une autre), la distribution d'objets (Java permet d'envoyer des objets d'une machine vers une autre).

Robuste

Java était destiné au départ pour être embarqué dans des équipements électroniques. Lorsque des milliers de ces petits équipements partent dans la nature, il est hors de question de les faire revenir pour corriger un bug.

Ne pouvant effectuer de patchs et autres services packs (comme cela se fait chez certains éditeurs…), il est nécessaire de faire les programmes les plus fiables possibles.

Pour cela, mettons le plus de chances de notre côté :

- Typage strict des données.
- Pas d'accès direct aux adresses des données en mémoire.
- Le Garbage Collector permet de récupérer toute mémoire non utilisée, c'est un gage de fonctionnement sans limite de temps, beaucoup d'applications traditionnelles nécessitaient un arrêt régulier pour cause de saturation mémoire due à des allocations non libérées après utilisation.
- La gestion des erreurs par un mécanisme d'exceptions oblige le programmeur a avoir une politique de vigilance à l'égard des erreurs possibles lors de l'exécution du programme.

Portable

Cette portabilité est réelle, et possible grâce à la notion de machine virtuelle (JVM : Java Virtual Machine). Dans les machines virtuelles, toute l'API de Java, normalisée par Sun, est implémentée pour se comporter de la même manière, quel que soit l'environnement. Nous verrons son fonctionnement.

Performant

Bien qu'interprété, ce langage est performant grâce aux optimisations du compilateur JIT (Just In Time) et à la simplicité de l'architecture.

Multi-tâche

C'est la possibilité d'exécuter plusieurs traitements simultanément, donc d'améliorer sensiblement les performances, mais aussi et surtout la disponibilité d'une application (traitements en tâches de fond permettant à l'utilisateur de continuer à travailler).

Sécuritaire

La sécurité tient une place importante dans Java, au travers d'un certain nombre d'aspects :

- Le "bac à sable" que nous verrons lorsque nous parlerons des "applets", ces petits programmes Java distribués sur Internet, permet de limiter les risques d'infection par des virus. On constate aujourd'hui que pas un seul virus informatique n'a pu être conçu en Java (alors qu'ils fourmillent dans d'autres technologies concurrentes).
- La notion de police de sécurité permet de personnaliser les niveaux de droits des applications.
- Enfin, Java possède des API destinées au cryptage, à la gestion de l'authentification, des certificats, etc.

Riche

Java est probablement le langage de programmation le plus riche, avec plus de 3000 classes et interfaces.

Tous les domaines sont traités :

- Calculs
- Entrées/sorties
- Interface graphique
- Réseau
- Bases de données
- Distribution d'objets
- Multimédia
- etc.

De plus, de nombreuses classes spécialisées sont disponibles sur le marché, parfois gratuitement et souvent fournies avec les sources.

Java est une véritable communauté de développeurs enthousiastes.

Evolutions de Java

- **1995: Java V 1.0**
  - Langage de programmation objet et complet
- **1997: Java V 1.1**
  - Introduction des composants: Java Beans
- **1998: Java V 1.2 - Java 2**
  - Introduction de Swing et des éditions J2SE, J2ME et J2EE
- **1999: Java V 1.3**
- **2002: Java V 1.4**

Java démarre sa carrière commerciale en 1995. Il va naître à l'aube de la révolution de l'Internet, et c'est dans ce domaine d'applications qu'il va rapidement s'imposer.

D'abord par les « applets », sortes de petites applications clientes graphiques qui tournent au sein des navigateurs, et qui apportent traitements, interactivité et animations aux documents HTML, puis par les « servlets », programmes qui s'exécutent au sein des serveurs Web pour générer dynamiquement du contenu.

Mais au cours de ses évolutions, sa qualité et sa puissance va lui permettre aussi de s'imposer dans de nombreux autres domaines de l'informatique : client/serveur, serveurs de bases de données, serveurs d'objets distribués, moniteurs transactionnels, applications embarquées (téléphones cellulaires, assistants personnels) ; etc.

Java possède un très grand nombre d'API normalisées, qui couvrent de plus en plus de domaines au fil des nouvelles versions.

Version 1.0

Sun lance un langage de programmation portable pour applications Client/Serveur et applets.

Les applets sont de petites applications, téléchargées sur internet dans le navigateur, et qui s'exécutent dans celui-ci au sein même du document en cours de visualisation. Le grand intérêt des applets est d'apporter des programmes à l'intérieur même des documents.

C'est surtout avec ces applets que Java va se faire connaître. Netscape implémente une machine virtuelle Java dès la version 2 de Navigator, Microsoft suit très vite en l'implémentant dans la version 3 d'Explorer.

Java comportait alors déjà une grande richesse de classes, tout ce qu'il fallait pour développer des applications classiques :

- Entrée-sorties standard
- Structures de données (tableaux, vecteurs, hashtable…)

- Interface utilisateur graphique
- Applets
- Réseau
- JDBC (accès aux bases de données relationnelles) en option

Version 1.1

Les évolutions majeures de cette version étaient destinées à permettre l'implémentation de composants logiciels en langage Java : les JavaBeans.

- Normalisation des composants : les JavaBeans
- Introspection des classes, qui permet d'interroger les composants sur les services qu'ils offrent
- Sérialisation des objets qui permet de les stocker ou de les déplacer automatiquement
- Nouvelle gestion des événements
- RMI (Remote Method Invocation) qui permet d'invoquer les services d'un composant qui se trouve sur une autre machine du réseau
- Fichiers JARS destinés à faciliter le déploiement des composants et des applications
- JDBC en standard dans la JVM

Version 1.2

Cette version va principalement apporter :

- JFC (Java Foundation Classes) : toute une collection de composants JavaBeans graphiques pour les interfaces utilisateur
- Java 2D : le support pour la création de graphiques en deux dimensions
- Java Sound : le support du son (Midi, Wave, etc.)
- Les servlets et les pages JSP, utilisés dans les serveurs d'applications et les serveurs web J2EE
- JNDI (Java Naming and Directory Interface) qui permet de standardiser l'accès à des serveurs de nommage, notamment pour stocker des collections d'objets sérialisés
- Java IDL (Interface Description Language) pour s'ouvrir au monde CORBA
- Améliorations diverses (Sécurité, RMI, Sérialisation, JNI, JAR, JDBC…)

La liste de ces évolutions est disponible sur le site de Sun :

http://java.sun.com/products/jdk/1.2/docs/relnotes/features.html

A partir de la version 1.2 c'est aussi la déclinaison de java en trois éditions.

Java 2 : Trois éditions

Ces éditions sont fournies avec des implémentations de référence par Sun, mais d'autres éditeurs peuvent créer leur propre implémentation et la faire valider par Sun (Java Compliance).

Des éditeurs importants vont suivre, citons notamment : Allaire, Apache, BEA Systems, Borland, IBM, Oracle, Rational, Sybase…

J2SE : Java 2 Standard Edition

Cette édition est destinée aux applications clientes. On y trouve donc à la fois le noyau, l'interface graphique, le réseau, JDBC et la partie cliente de RMI.

Nous ne traiterons dans cet ouvrage que de l'édition standard (J2SE).

J2EE : Java 2 Enterprise Edition

Cette édition comprend tous les éléments pour construire des serveurs d'applications :

- La version standard de Java (sans JFC)
- Les Servlets et les pages JSP
- JNDI
- JDBC 2 (extensions pour le pooling de connexions)
- EJB (Enterprise Java Beans) qui sont les composants métier des architectures Java

J2ME : Java 2 Micro Edition

Cette édition est destinée aux environnements embarqués ou de très petites tailles (téléphones, PDA, cartes à puce, etc…).

Elle est composée de deux configurations qui correspondent aux possibilités des différents équipements :

- Connected Limited Device Configuration (CLDC) est la configuration qui correspond aux équipements limités, aussi bien en terme de CPU (16 ou 32 bits avec un minimum de 128 kilo octets de mémoire vive) qu'en terme de connexion réseau (connexions non permanente). On peut citer les téléphones mobiles, les assistants personnels d'entrée de gamme…
- Connected Device Configuration (CDC) est la configuration pour des équipements plus performants, avec notamment une connexion réseau permanente et éventuellement de haut débit. On peut citer les ordinateurs de bord, les terminaux de télévision numérique, les assistants personnels de nouvelle génération…

A cela s'ajoute une logique de profils, qui définissent les possibilités de la machine virtuelle (en termes de réseau, d'interface graphique, de connexion aux bases de données, de calculs mathématiques, etc…). Le profil le plus connu est le "Mobile Information Device Profile" (MIDP) qui est destiné aux téléphones mobiles et aux assistants personnels d'entrée de gamme. C'est en fait le successeur du WAP.

Version 1.3

Cette version apporte principalement des corrections et des évolutions (RMI, drag & drop, Java Sound, Java 2D, Swing, etc.) ainsi que l'intégration de JNDI (Java Naming and Directory Interface) sur le client.

La liste est disponible sur le site de Sun :

http://java.sun.com/j2se/1.3/docs/relnotes/features.html

Version 1.4

La toute nouvelle version de Java apporte de nombreuses améliorations telles que :

- Un nouveau gestionnaire d'entrées/sorties (NIO : New Input / Output)
- De nouvelles fonctionnalités dans JFC, comme la gestion de la molette de défilement de la souris ou la gestion du mode graphique plein écran, le drag&drop.
- L'intégration des API pour XML (DOM, SAX et XSLT)
- Le support d'IP V.6 dans l'interface réseau
- Un outil de déploiement d'applications clientes en Java : Java WebStart

Se reporter au site de Sun :

http://developer.java.sun.com/developer/technicalArticles/releases/j2se1.4/

Remarque : **Toutes les nouveautés propres à la version 1.4 seront signalées tout au long de ce livre par le logo ci-contre.**

Compilation et exécution des programmes

Erreur ! Liaison incorrecte.

Les programmes Java sont compilés. Toutefois, le code généré n'est pas vraiment du binaire, c'est un pseudo-code qui a l'avantage d'être suffisamment générique pour être ensuite traduit en instructions d'un quelconque langage machine d'une quelconque plate-forme d'accueil, au moment de l'exécution.

La compilation

Un compilateur est un programme qui va simplement prendre un fichier **.java** (format texte) et générer un fichier **.class** (binaire) qui contiendra le pseudo-code.

Le compilateur va aussi vérifier :

- La syntaxe du programme
- Certaines erreurs algorithmiques (instructions jamais exécutées…)
- La présence et la cohérence des librairies (packages) utilisés

Remarque :

Le pseudo-code généré est normalisé par Sun. Les fichiers **.class** ont un format bien défini, dans lequel on conserve aussi les noms des méthodes et des attributs, ce qui peut faciliter le Reverse Engineering. Certains outils existent sur le marché pour décompiler des classes Java. C'est très efficace mais pas très légal…

Différents éditeurs proposent des compilateurs, Sun offre, dans le JDK, une implémentation de référence. C'est celle que nous utiliserons pour les exercices de ce cours.

L'exécution

Les classes sont exécutées dans la JVM (Java Virtual Machine). Ce programme, qui devra être présent sur toutes les plateformes cibles, a en charge de transformer le pseudo-code des classes en code machine pour leur exécution.

A noter que les classes utilisées lors de la compilation doivent aussi être présentes lors de l'exécution sur le poste client.

Les variables d'environnement

Pour utiliser le compilateur, comme l'interpréteur, il est nécessaire de positionner les variables d'environnement PATH et CLASSPATH :

- PATH vers le répertoire BIN du JDK
- CLASSPATH vers les répertoires des librairies

On peut positionner ces variables dans le système d'exploitation, ou bien créer un fichier de commandes de la forme :

```
set PATH=C:\JDK1.4\BIN;%PATH%
```

```
set CLASSPATH=.;C:\OutilsJava;%CLASSPATH%
```

Remarque :

Les entrées du CLASSPATH sont séparées par un point-virgule (";"). La première entrée est un point (".") qui spécifie le répertoire courant. Cette entrée est nécessaire lorsque l'on veut compiler ou exécuter une classe qui se trouve dans le répertoire courant. Bien utile dans la phase de mise au point.

La console Java

La machine virtuelle Java est un programme en mode caractère. Il sera donc exécuté sous Windows dans un invité de commande. La sortie standard permet aux programmes Java d'afficher des chaînes de caractères, pratique lors des tests des programmes. C'est ce que l'on appelle la console Java.

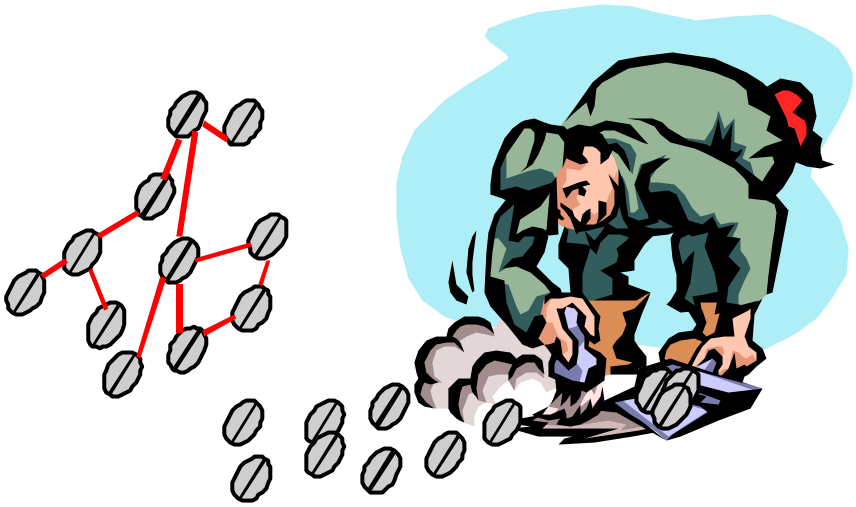
Les packages

Ils permettent de stocker les classes dans différents répertoires afin de faciliter leur classement (métier ou technique).

On peut aussi stocker les classes dans un fichier JAR (Java Archive), ce qui permet un déploiement plus facile de l'application, n'ayant qu'un seul fichier à distribuer alors que l'application est peut-être consituée de centaines de classes.

Nous verrons plus loin comment fabriquer des packages et les mettre dans des fichiers JAR.

Le Garbage Collector (ramasse-miettes)



Un atout important de Java est le Garbage Collector, aussi appelé en français le ramasse-miettes.

Derrière ce nom se cache une fonction primordiale de Java, qui a pour vocation de libérer le programmeur de la lourde et importante tâche de toujours détruire les objets qu'il a créé lorsqu'il n'en a plus besoin, et ce, sous peine d'encombrer la mémoire centrale de l'ordinateur jusqu'à une saturation complète du système.

Comment cette merveille fonctionne-t-elle ? Il s'agit en fait d'un processus s'exécutant en tâche de fond de la machine virtuelle, et qui va régulièrement faire le ménage dans la mémoire en détruisant les objets dont on n'a plus besoin.

Mais, me direz vous, comment ce programme peut-il savoir si un objet ne sert plus à rien ? Il considèrera simplement qu'un objet ne sert plus à rien lorsqu'il ne peut plus être utilisé.

Et comment savoir s'il ne peut plus être utilisé ? Lorsque sa référence est au delà de la visibilité du code de l'application. Cela correspond aux cas suivants :

- L'objet est local à une fonction, on est sorti de la fonction
- L'objet est référencé dans un autre objet qui a été détruit
- L'objet n'est plus référencé car la variable qui contenait sa référence a été initialisée à partir de la référence d'un autre objet ou a été mise à "null".

Le mécanisme permettant ce prodige n'est pas nouveau, d'autres technologies l'ont déjà exploité. C'est en fait un automatisme interne à la machine virtuelle Java, qui va conserver dans une table les références de tous les objets créés, et qui va associer à chacune de ces références un compteur qui s'incrémentera à chaque nouvelle référence (assignation, passage en argument, insertion dans un tableau ou une collection…), et qui se décrémentera à chaque destruction de référence.

Par ailleurs, une tâche de fond va régulièrement consulter la table des références, et détruira de la mémoire tous les objets dont le compteur est à 0, car cela signifie qu'ils ne sont plus référencés nulle part et donc inutilisables par l'application.

Remarque

Un objet dont on n'a plus besoin peut rester référencé accidentellement, et donc ne pas être détruit.

Le développeur a peu de contrôle sur le Garbage Collector, les spécifications de Sun n’imposent ni algorithme particulier ni fréquence de nettoyage.

Les API fournies dans Java

Erreur ! Liaison incorrecte.

Les API Java sont constituées de milliers de classes.

Elles sont organisées en deux parties :

- Java Core API : c'est l'API de base que l'on retrouve implémentée dans toute machine virtuelle. Ses classes font partie du package "java".
- Java Standard Extention API : ce sont les extensions, normalisées par Sun au travers d'interfaces, et qui sont implémentées par les éditeurs de solutions (serveurs d'applications, e-business, web…). Elles font partie du package "javax".

Java Core API

Elle contient les packages suivants :

- java.lang : toutes les classes de base pour le langage (String, Thread…)
- java.util : divers utilitaires (Date, Vector, Hashtable…)
- java.math : si les fonctions mathématiques sont déjà implémentées dans le package java.lang, ce package permet les calculs sur des numériques de très grande taille
- java.io : toutes les entrées/sorties (console, fichiers, filtres, pipelines…)
- java.awt : Abstract Window Toolkit : Tout ce qui concerne l'interface graphique
- java.applet : le support des applets
- java.sql : l'accès aux bases de données (JDBC : Java DataBase Connector)
- java.beans : La construction de composants JavaBeans
- java.rmi : l'invocation de méthodes d'objets distants (RMI : Remote Method Invocation)
- java.net : le support du réseau TCP/IP (sockets, UDP…)

Nous étudierons l'ensemble de ces packages dans ce cours.

Java Standard Extension

Cette partie de l'API de Java est la plus évolutive. Elle correspond à des besoins nouveaux qui ont été integrés progressivement. La particularité de ces API est qu'elles sont fondées sur des interfaces créées et donc normalisées par Sun, et dont l'implémentation est à la charge d'éditeurs tiers.

Certaines de ces API seront appelées à rejoindre la Java Core API.

Le JDK 1.4 de la version standard (J2SE) est proposé avec, en plus de la Java Core API, des extensions suivantes :

- javax.swing : l'interface graphique avancée de Java
- javax.print : le support des impressions
- javax.crypto : le support de la cryptographie
- javax.security : le support de la sécurité (au niveau client)

L'ensemble de ces API sont dans des packages situés dans le fichier RT.JAR qui se trouve dans le sous-répertoire /JRE/LIB du répertoire d'installation du JDK.

Les autres API font partie principalement de la version Enterprise (J2EE) :

- javax.servlet : concerne les programmes serveurs
- javax.naming : support du nommage et des annuaires (JNDI : Java Naming and Directory Interface). Contient une implémentation LDAP.
- javax.xml : gestion des documents XML
- javax.mail : support des protocoles pour le courrier électronique (SMTP, POP3, IMAP)
- javax.jms : JMS : Java Messaging Services. Gestion de services de messages pour applications s'appuyant sur des MOMs (Message Oriented Middleware)
- javax.transaction : support des transactions, notamment les transactions distribuées (Commit à deux phases)
- javax.ejb : support des EJB (Enterprise Java Beans), composants métiers bénéficiant d'un environnement transactionnel et sécurisé.

Environnements de développement

Erreur ! Liaison incorrecte.

Le JDK

Sun propose un kit de développement minimum : Le Java Development Kit (JDK).

On y trouve :

- Toutes les classes du Java (J2SE)
- Un compilateur : javac.exe
- Une machine virtuelle Java : java.exe
- Un débugger
- Un générateur de documentation technique : javadoc.exe
- Un générateur de fichiers JAR (Java ARchive) : jar.exe
- Une visionneuse d'applets : appletviewer.exe
- Divers outils pour le RMI (Remote Method Invocation)
- etc.

Si l'on ajoute au JDK un éditeur de texte (à coloration syntaxique de préférence, comme par exemple Ultra-Edit ou Zeus), on a tout ce qu'il faut pour développer des applications, et même de grosses applications.

Si, par contre, on tient à un certain confort, on cherchera à acquérir un IDE (Integrated Development Environment), environnement de développement integré dans lequel on trouvera beaucoup plus d'outils et de convivialité (aide en ligne, assistants, génération automatique de code…).

Les fichiers .class

Le développement d'applications consiste à écrire des fichiers sources dans un langage de programmation (par exemple Java), puis à les compiler afin de les traduire dans un langage compréhensible par l'ordinateur.

Ce dernier, appelé langage machine, permettra à l'ordinateur d'exécuter les instructions définies par le programmeur.

Le problème que l'on a depuis toujours, est que chaque marque de microprocesseur possède son propre langage machine. Les programmes en langage machine ne sont donc pas portables.

L'idée exploitée dans Java est d'inventer un nouveau langage machine, suffisamment générique pour s'adapter facilement à tous les microprocesseurs. Les programmes Java sont compilés dans ce langage.

Au moment de l'exécution de l'application, l'interpréteur (appelé machine virtuelle Java) n'a plus qu'à convertir les instructions génériques en instructions propres au microprocesseur de la plate-forme d'exécution (le "Run-Time").

Ces instructions génériques s'appellent des "op-codes". Les fichiers .class contiennent uniquement de instructions de ce type.

Pour permettre à tous les compilateurs et toutes les machines virtuelles Java d'effectuer cette traduction, le format des fichiers .class et les codes opérations sont normalisés par Sun. Ces spécifications sont détaillées dans des documents téléchargeables depuis le site de Sun : http://java.sun.com

Une application est composée d'au moins un fichier .class, sorte de programme principal, dans lequel on trouvera le point de démarrage : la méthode "main".

Un fichier .class contient une seule classe. Par conséquent, une application sera constituée d'un grand nombre de fichiers.

JavaDoc

Le problème de la documentation des programmes n'est pas nouveau. Afin de faciliter le travail de groupe et la maintenance des applications, tout bon programmeur se doit de documenter le source qu'il écrit, c'est à dire expliquer clairement, dans des zones appropriées (les zones de commentaire) ce qu'il fait, comment il le fait et pourquoi il le fait ainsi.

JavaDoc est un outil qui permet, en analysant les sources, de remonter certains des commentaires et de les mettre en forme dans une documentation propre et présentable (au format HTML).

C'est un outil très intéressant, aussi bien pour les développeurs que pour les chefs de projets. Pour ces derniers, la consultation des documentations générées automatiquement par JavaDoc est un bon complément à une revue de code.

Un guide d'utilisation de JavaDoc se trouve en annexe de ce livre. Je vous conseille de vous y reporter lorsque vous commencerez à faire des programmes afin de prendre rapidement de bonnes habitudes.

Debugging d'applications

Un débugger mode caractère est fourni dans le JDK (programme jdb.exe).

Il peut paraître spartiate, et pourtant il est complet. D'ailleurs, certains outils graphiques s'appuient entièrement dessus, n'offrant qu'une interface graphique permettant de tracer directement dans le source lors de l'exécution pas à pas.

Les IDE du marché

Un IDE (Integrated Development Environment) est un logiciel permettant d'écrire des applications rapidement et efficacement.

On trouvera notamment dans ce type de logiciel :

- Un éditeur de sources à coloration syntaxique
- Des assistants de préfabrication de code
- Un explorateur de classes
- Une aide en ligne
- Un gestionnaire de versions
- Un debugger graphique
- Un composeur de fenêtres graphiques
- Etc.

Il existe un grand nombre d'environnements integrés de développement Java sur le marché. Les produits les plus utilisés sont Borland JBuilder , IBM Visual Age et son successeur : Websphere Studio Application Developer.qui s'appuie sur Eclipse, IDE développé par IBM en open source.

On trouvera en annexe un tutorial pour prendre en main Eclipse.

Mais il existe bien d'autres produits. Citons notamment les produits proposés par de grands éditeurs :

- Forté de Sun
- Oracle JDeveloper
- Microsoft Visual J++
- Symantec Visual Café
- Sybase Power J

On peut aussi remarquer des environnements de développement créés par de petites sociétés ou des développeurs isolés, notamment :

- IDEA de la société IntelliJ (http://www.intellij.com)
- JCreator de Wendel de Witte (http://www.jcreator.com)

Technologies des compilateurs et interpréteurs

Erreur ! Liaison incorrecte.

Afin d'améliorer les performances de l'exécution des programmes, il existe actuellement trois solutions :

- La compilation Just-In-Time (JIT)
- La technologie HotSpot de Sun
- La compilation native

La compilation Just-In-Time

C'est une sur-couche de la machine virtuelle Java classique. Le principe d'un JIT Compiler est d'exécuter un compilateur binaire en même temps que l'interpréteur de l'application. Ainsi au pire, le code passe par l'interpréteur, au mieux, s'il est déjà compilé, c'est le code binaire qui est exécuté, donnant à l'application de bien meilleures performances.

Par contre, la compilation en code natif demande beaucoup de ressources. De plus, les performances ne sont pas aussi bonnes qu'avec des programmes compilés en code natif, comme les programmes écrits en C, C++, Cobol, etc. car la compilation en code natif doit être effectuée à chaque lancement du programme.

Un autre inconvénient est que, pour améliorer les performances lors de la compilation en code natif, certaines optimisations coûteuses en temps, que l'on trouve dans tout bon compilateur C ou C++ , ne sont pas effectuées. Le code natif généré n'est donc pas aussi bien optimisé.

Enfin, le JIT Compiler ne sait pas se passer de la JVM classique, car un certain nombre de portions de code, qu'il estime inutile de compiler en natif, sont toujours interprétées par la JVM.

On peut dire aujourd'hui que la compilation Just-In-Time est une amélioration sensible, mais reste malgré tout en deçà de la compilation classique native des programmes.

La technologie HotSpot de Sun

Cette technologie développée par Sun Microsystems part du principe que seules certaines parties des programmes, appelés points chauds (Hot Spots en anglais), nécessitent une compilation native conservée en mémoire (une sorte de cache mémoire de code).

On s'apercevra que les points chauds sont relativement peu nombreux dans des applications clientes, par contre beaucoup plus visibles dans des applications serveur, comme des conteneurs de servlets ou d'EJB (Enterprise Java Beans).

C'est la raison pour laquelle on utilisera beaucoup plus souvent la technologie HotSpot pour faire tourner des programmes serveurs, comme les serveurs d'applications J2EE par exemple.

La compilation des points chauds en code natif est faite pendant le déroulement du programme, ce qui se traduit par des performances qui s'améliorent progressivement pendant l'exécution du programme. Là encore, cette technologie est plus adaptée à des programmes serveur qui sont faits pour tourner pendant des jours ou des semaines sans s'arrêter.

La compilation native

La compilation native consiste à compiler le code source directement en code natif, pour une cible déterminée. Cette compilation n'est faite qu'une seule fois par le développeur de l'application, ce qui permet de prendre un peu plus de temps pour bénéficier des meilleurs algorithmes d'optimisation de code.

Les performances des programmes Java compilés de cette manière sont aussi bonnes que celles des programmes écrits en C ou C++ .

Un autre avantage de la compilation native est une meilleure protection contre le reverse engineering, très facile à faire en Java car le format des fichiers des classes étant normalisé et bien connu.

L'application générée ne pourra tourner que dans l'environnement cible du compilateur. Si l'on souhaite lui permettre de tourner sur plusieurs plate-formes, il sera nécessaire d'avoir un compilateur natif pour chaque cible et de compiler l'application. On aura donc autant de programmes compilés que de cibles supportées, ce qui compliquera un peu la distribution du logiciel.

Il exite deux produits intéressants pour faire de la compilation native :

- Microsoft Visual J++ (Version 6) est un excellent outil pour créer des clients Windows en langage Java. Microsoft propose en plus, dans leur environnement, des classes spécifiques au monde Windows (et fort utiles). Attention : portabilité condamnée, au grand dam de Sun, ce qui a valu à Microsoft un retentissant procès qu'il a perdu.
- JET de la société Excelsior (http://www.excelsior-usa.com/) est un compilateur binaire intéressant, puisqu'il génère du code natif à partir de toutes les classes Java, y compris celles du JDK, ce qui le rend, en théorie, complètement transparent vis à vis de la version du JDK utilisée.

Et la portabilité ?

La compilation binaire n'est pas très appréciée dans le monde Java, car elle semble contraire à l'idée de portabilité : « Write once, run anywhere ». En effet, un programme compilé pour un environnement ne pourra s'exécuter que dans cet environnement.

Toutefois, je pense que cette crainte est infondée. En effet, la portabilité de Java ce n'est pas de dire que les classes générées et déployées peuvent s'exécuter sur toute plate-forme disposant d'une JVM, mais plutôt que les sources développés et utilisant les classes de la JVM peuvent tourner, sans aucune adaptation du code, sur n'importe quelle plate-forme.

La richesse de Java est avant tout axée sur de deux points :

- C'est un excellent langage orienté objet, apportant aux développeurs une productivité très importante.
- C'est un environnement de développement d'une richesse considérable de classes permettant de faire pratiquement n'importe quel type d'applications : graphiques, multimédias, orientées réseau, connectées à des bases de données, etc.

Les convertisseurs C++

Certaines solutions de compilations natives sont en fait des convertisseurs C++. Cela consiste à traduire le code Java en code C++.

La similarité de ces deux langages permet de le faire, mais cette traduction va générer un code C++ un peu lourd et pas forcément aussi bien écrit que si cela avait été fait par un bon programmeur C++.

L'avantage est que les compilateurs C++ existent dans pratiquement tous les environnements, la portabilité est donc assurée pourvu que l'on fournisse le run-time de bas niveau nécessaire aux classes de la JVM, elles aussi traduites en C++.