

# LA MACHINE VIRTUELLE JAVA

Pierre Letouzey<sup>1</sup>  
letouzey@irif.fr

Irif - Univ. Paris

janvier 2020

---

1. Merci à Y. Régis-Gianas pour ces transparents

## Vue d'ensemble de la JVM

---

# Présentation

- ▶ La JVM a été initialement spécifiée par SUN MICROSYSTEM pour exécuter le code-octet produit par les compilateurs JAVA.
- ▶ Sa spécification est publique :

<http://docs.oracle.com/javase/specs/jvms/se11/html/index.html>

et il en existe de (très) nombreuses implémentations :

Azul VM - CEE-J - Excelsior JET - **J9 (IBM)** - JBed - JamaicaVM - JBlend - JRockit - Mac OS Runtime for Java (MRJ) - MicroJvm - Microsoft Java Virtual Machine - OJVM - PERC - Blackdown Java - C virtual machine - Gemstone - Golden Code Development - Intent - Novell - NSIcom CrE-ME - HP ChaiVM MicrochaiVM - **HotSpot** - AegisVM - Apache Harmony - CACAO - Dalvik - IcedTea - IKVM.NET - Jamiga - JamVM - Jaos - JC - Jelatine JVM - JESSICA - Jikes RVM - JNode - JOP - Juice - Jupiter - JX - Kaffe - leJOS - Maxine - Mika VM - Mysaifu - NanoVM - SableVM - Squawk virtual machine - SuperWaba - TinyVM - VMkit - Wonka VM - Xam

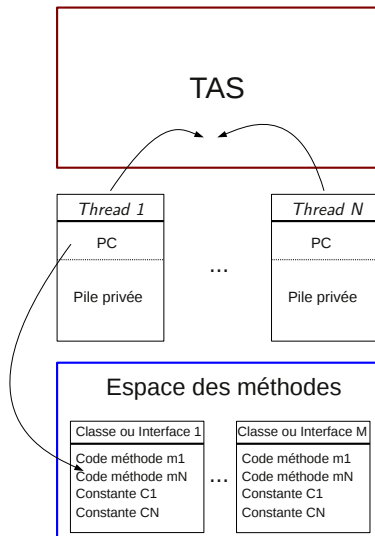
- ▶ La spécification laisse une importante liberté d'implémentation.

# Modèle de calcul

- ▶ Comme la machine d'OCAML, La JVM est une machine à pile.
- ▶ Elle a été pensée pour la programmation objet, concurrente et “mobile” :
  - ▶ Appels de méthodes.
  - ▶ Fils d'exécution (*threads*) avec mémoire locale et mémoire partagée.
  - ▶ Chargement dynamique de code et vérification de code-octets.
- ▶ Ce modèle de calcul est adapté à la compilation d'autres langages. On trouve des compilateurs produisant du code-octet JAVA pour les langages : ADA, AWK, C, COMMON LISP, FORTH, RUBY, LUA et même OCAML !
- ▶ Un gros défaut cependant : pas de traitement des appels terminaux.  
(instructions présentes dans .NET et dans OCAML)

# Composantes de la JVM

- ▶ Le tas est partagé entre les *threads*.
- ▶ Le code est partagé entre les *threads*.
- ▶ Chaque *thread* a ses propres PC et pile.
- ▶ Le contenu de toutes ses composantes évolue durant l'exécution.
- ▶ Les données au format Class permettent de peupler l'espace des méthodes.



## Les valeurs de la JVM

---

# Deux grandes familles de données

- ▶ La JVM manipule deux grandes familles de données :
  - ▶ les données de types **primitifs** :
    - ▶ les valeurs numériques : entières ou à virgule flottante ;
    - ▶ les booléens ;
    - ▶ les adresses de code.
  - ▶ Les **références** qui sont des pointeurs vers des données allouées dans le tas (des instances de classe ou des tableaux).
- ▶ Contrairement à OCAML, il n'y a pas de bit réservé pour différencier les types primitifs et les références.
- ▶ Le **typage** du code-octet garantit qu'à tout instant, le type des données est celui attendu.
- ▶ Le ramasse-miette (GC) utilise une information de typage pour déterminer si une donnée dans le tas est un pointeur ou une constante.

# Les types entiers

byte	8	bits signé
short	16	bits signé
int	32	bits signé
long	64	bits signé
char	16	bits non signé



# Les types flottants

float	32	bits simple-précision
double	64	bits double-précision

- ▶ *Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std. 754-1985.

- ▶ En passant, une lecture très recommandée :

*What Every Computer Scientist Should Know About Floating Point Arithmetic*

David Goldberg (1991)

# Le type booléen

- ▶ La spécification de la JVM définit un type booléen.
- ▶ Les valeurs de type booléen sont représentées par des entiers (1 pour `true` et 0 pour `false`).

# Les adresses de code

- ▶ Les adresses de code ne sont pas modifiables par le programme.

# Les types de références

- ▶ Il y a trois types de références. Les références vers :
  - ▶ les instances de classes ;
  - ▶ les implémentations d'interface ;
  - ▶ les tableaux.
- ▶ La référence spéciale `null` ne fait référence à rien et a ces trois types.

## Les espaces de données

---

# Des espaces privés et des espaces globaux

- ▶ Lors de son lancement, la JVM initialise les espaces de données nécessaires à l'exécution du programme. Ils sont détruits lorsque la machine est stoppée.
- ▶ Le tas et l'espace des méthodes sont des espaces globaux.
- ▶ Chaque *thread* possède une pile privée et un registre PC. Ces deux espaces données sont initialisés à la création du *thread* et détruits à la fin de son exécution.

# Le registre PC

- ▶ À chaque instant, un *thread* est lié à une **méthode courante**.
- ▶ Le PC est une position à l'intérieur du code de cette méthode.

# La pile privée

- ▶ La pile privée sert à stocker des **blocs d'activation**.
- ▶ Ce sont des espaces mémoires temporaires pour les variables locales et les résultats temporaires.
- ▶ La pile privée est aussi utilisée pour passer l'adresse de retour d'une méthode ainsi que ses arguments effectifs.
- ▶ On accède à un seul bloc d'activation à la fois.



# L'espace des méthodes

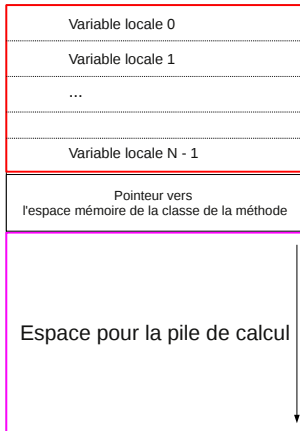
- ▶ Pour chaque classe, l'espace des méthodes contient :
  - ▶ un ensemble de constantes ;
  - ▶ des champs de classes partagés (les champs notés **static**) ;
  - ▶ des données liées aux méthodes ;
  - ▶ le code des méthodes et des constructeurs ;
  - ▶ le code de méthodes spéciales pour l'initialisation des instances de classes et de type d'interface.

## L'exécution du code

---

# L'appel de méthode

- ▶ À chaque fois qu'une méthode est invoquée, un bloc d'activation est empilé.
- ▶ Quand l'exécution de la méthode est terminée, ce bloc est dépilé.



Organisation d'un bloc d'activation.

## Calcul du bloc d'activation

- ▶ Le calcul des tailles des différentes composantes du bloc d'activation est effectué par le compilateur.
- ▶ Une variable locale a une taille de 32 bits.
- ▶ Les données de type `long` ou `double` utilisent deux variables locales.
- ▶ On accède aux variables locales *via* leur position dans le bloc d'activation.
- ▶ Ces indices débutent à 0.
- ▶ Les variables locales sont utilisées pour stocker les arguments effectifs des appels de méthode. Par convention, le premier argument (d'indice 0) contient toujours la référence vers `this`, c'est-à-dire l'instance de l'objet dont on exécute la méthode.
- ▶ La taille maximale de pile nécessaire à l'exécution d'un code donné est calculable. (C'est d'ailleurs un bon exercice pour le cours de compilation !)

## Le jeu d'instructions

---

# Résumé des instructions

- ▶ Les instructions de la JVM sont typées.
- ▶ Le nom de chaque opération est préfixée par une lettre indiquant le type des données qu'elle manipule :
  - ▶ 'i' : int
  - ▶ 'l' : long
  - ▶ 's' : short
  - ▶ 'b' : byte
  - ▶ 'c' : char
  - ▶ 'f' : float
  - ▶ 'd' : double
  - ▶ 'a' : reference
- ▶ Par ailleurs, les *opcodes* sont stockés sur un octet.
- ▶ Référence :

[http://java.sun.com/docs/books/jvms/second\\_edition/html/Instructions.doc.html](http://java.sun.com/docs/books/jvms/second_edition/html/Instructions.doc.html)

# Lecture et écriture des variables locales

- ▶ `iload, iload_<n>, lload, lload_<n>, fload, fload_<n>, dload, dload_<n>, aload, aload_<n>`  
Charge une variable locale au sommet de la pile de calcul.
- ▶ `istore, istore_<n>, lstore, lstore_<n>, fstore, fstore_<n>, dstore, dstore_<n>, astore, astore_<n>`  
Écrit le contenu du sommet de la pile dans une variable locale.
- ▶ `bipush, sipush, ldc, ldc_w, ldc2_w, aconst_null, iconst_m1, iconst_<i>, fconst_<f>, dconst_<d>`  
Empile une constante au sommet de la pile.
- ▶ `wide` :  
Modifie le sens de l'instruction suivante : la prochaine instruction devra attendre un indice de variable locale codé sur 2 octets et non 1 seul.

# Opérations arithmétiques

- ▶ Addition : **iadd**, **ladd**, **fadd**, **dadd**.
- ▶ Soustraction : **isub**, **lsub**, **fsub**, **dsub**.
- ▶ Multiplication : **imul**, **lmul**, **fmul**, **dmul**.
- ▶ Division : **idiv**, **ldiv**, **fdiv**, **ddiv**.
- ▶ Reste de la division : **irem**, **lrem**, **frem**, **drem**.
- ▶ Négation : **ineg**, **lneg**, **fneg**, **dneg**.
- ▶ Décalage : **ishl**, **ishr**, **iushr**, **lshl**, **lshr**, **lushr**.
- ▶ “Ou” sur la représentation binaire : **ior**, **lor**.
- ▶ “Et” sur la représentation binaire : **iand**, **land**.
- ▶ “Ou exclusif” sur la représentation binaire : **ixor**, **lxor**.
- ▶ Incrémentation d'une variable locale : **iincr**.
- ▶ Comparaison : **dcmpg**, **dcmpl**, **fcmpg**, **fcmpl**, **lcmp**.



# Conversion des types de données

- ▶ Conversions d'élargissement : `i2l`, `i2f`, `i2d`, `l2f`, `l2d`, `f2d`.
- ▶ Conversions par projection : `i2b`, `i2c`, `i2s`, `l2i`, `f2i`, `d2i`, `d2f`.  
(Se reporter à la spécification pour les détails.)

# Opérations sur la pile

- ▶ Dépiler : `pop`, `pop2`.
- ▶ Dupliquer le sommet de la pile : `dup`, `dup2`, `dup_x1`, `dup2_x1`, `dup_x2`, `dup2_x2`, `swap`.

# Opérateur de flot de contrôle

- ▶ Branchement conditionnel : `ifeq`, `iflt`, `ifle`, `ifne`, `ifgt`, `ifnull`, `ifnonnull`, `if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmpgt`, `if_icmple`, `if_icmpge`, `if_acmpne`.
- ▶ Table de saut : `tableswitch`, `lookupswitch`.
- ▶ Branchement inconditionnel : `goto`, `goto_w`, `jsr`, `jsr_w`, `ret`.

# Manipulation d'objets et de tableaux

- ▶ Création d'une nouvelle instance de classe : `new`.
- ▶ Création d'un nouveau tableau : `newarray`, `anewarray`, `multianewarray`.
- ▶ Accès aux champs d'une classe : `getfield`, `setfield`, `getstatic`, `putstatic`.
- ▶ Chargement d'un tableau sur la pile de calcul : `baload`, `caload`, `saload`, `iaload`, `laload`, `faload`, `daload`, `aaload`.
- ▶ Affectation d'une case d'un tableau : `bastore`, `castore`, `sastore`, `iastore`, `lastore`, `fastore`, `dastore`, `aastore`.
- ▶ Empile la taille d'un tableau : `arraylength`.
- ▶ Vérification dynamique : `instanceof`, `checkcast`.

# Invocation de méthode

- ▶ Invoquer une méthode avec liaison tardive (*i.e.* en prenant en compte le type exact de l'instance considérée) : **invokevirtual**.
- ▶ Invoquer une méthode d'une interface dans une instance qui l'implémente : **invokeinterface**.
- ▶ Invoquer une initialisation d'instance, une méthode privée ou une méthode d'une classe mère : **invokespecial**.
- ▶ Invoquer une méthode de classe statique : **invokestatic**.

## Les outils utiles

---

# Hexedit

- ▶ Pour observer le contenu d'un fichier au format `CLASS`, on peut toujours utiliser `hexedit`.
- ▶ Mieux : `javap -c -verbose`

# Jasmin

- ▶ `jasmin` produit du code-octet `JAVA`.
- ▶ Le langage d'entrée est un langage assembleur qui est plus facile à utiliser que le langage de sortie de `javap`.



## Synthèse

---

# Synthèse

- ▶ L'architecture de la JVM et ses principales instructions.
- ▶ Mis sous le tapis : les exceptions.
- ▶ En TD, nous allons écrire du code à l'aide de `jasmin`.