Machines Virtuelles

Peter Habermehl (Merci à Michele Pagani et ...)

Université de Paris Cité
UFR Informatique
IRIF

Peter.Habermehl@irif.fr

21 mars 2024

JVM (Java Virtual Machine)

Présentation

La JVM a été spécifiée par Sun MicroSystem (maintenant Oracle) pour exécuter le code-octet produit par les compilateurs Java.
Sa spécification est publique:

https://docs.oracle.com/javase/specs/

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 ocamlrun, la JVM est une machine à pile + tas (mais **sans** accumulateur).

- JVM a été pensée pour la programmation

  objet: public, private, static, . . . ,
  concurrente: threads (mémoire partagé/privée), et
  mobile: class loader (verification$\mapsto$initialisation), .

- Ce modèle de calcul est adapté à la compilation d'autres langages. On trouve des compilateurs produisant du code-octet Java pour

  Ada, Awk, C, Common Lisp, Forth, Ruby, Lua, etc. et même OCaml !

- Un défaut cependant: pas de traitement des appels terminaux.

Typage statique du bytecode Java

- Java a été pensé pour du code mobile (téléchargé, exécuté sur la même machine).
- Un bout de code non fiable chargé dynamiquement ne doit pas corrompre l'état global de la machine.
- Le vérificateur de bytecode fournit une garantie statique que:
  - les sauts se font à des adresses dans la même fonction (pas de pointeur sauvage de code)
  - les données sont toujours initialisée (pas d'accès au contenu précédent)
  - les références sont bien typées (pas d'accès au contenu adjacent)
  - l'appel de méthodes est contrôlé (pas d'accès aux champs privés)

Compilation just-in-time

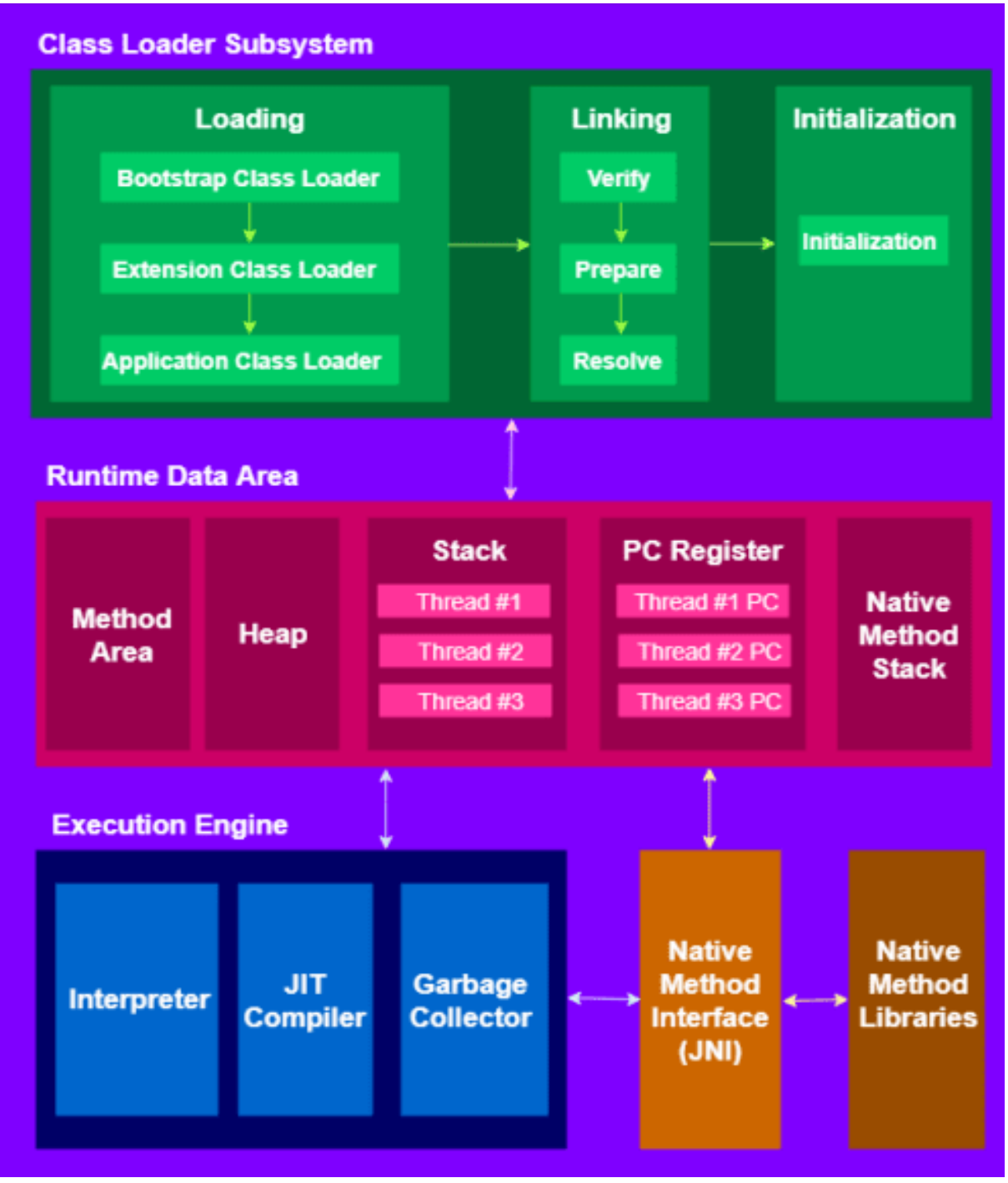
Alternativement à l'interprétation du bytecode par la M.V., on peut traduire le bytecode en assembleur natif pendant l'exécution

- (beaucoup) plus rapide que l'interprétation (malgré la phase de compilation)
- la compilation est entrelacée avec l'exécution, en Java à chaque appel de fonction (permet plus d'optimisations que la compilation batch)
- le code natif est caché (mémoisé)

Nouvelle contrainte: la compilation doit être efficace!

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



JVM runtime system

(merci `PlatformEngineer.com`)

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.

Des espaces privés et des espaces globaux

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.
- Stockage temporaire: variables locales, résultats temporaires.
- Pour passer l'adresse de retour d'une méthode + arguments effectifs.
- On accède à un seul bloc d'activation à la fois.

L'espace des méthodes

- 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.

Valeurs

Comme en Ocaml, on a deux 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, le code-octet est typé:

- 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 primitifs

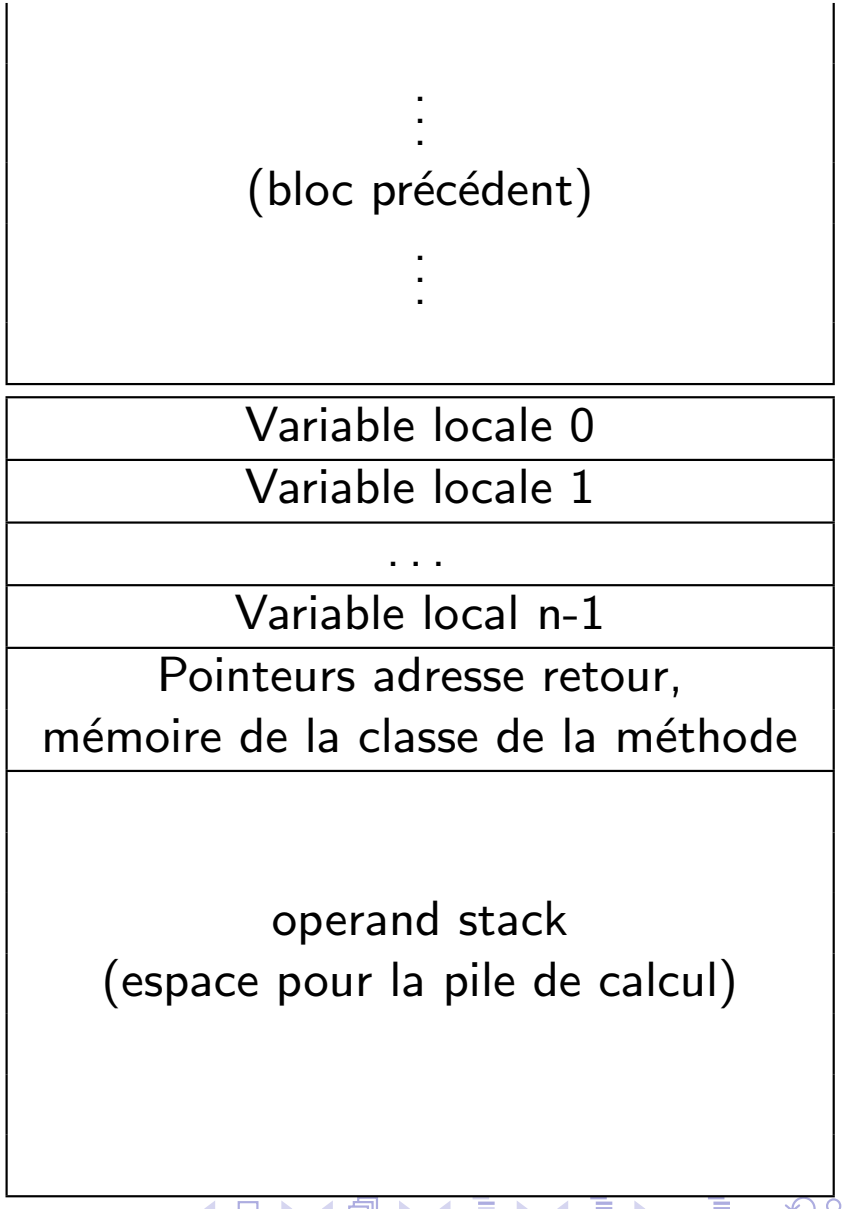
- Entiers
  `bytes` 8 bits signés
  `short` 16 bits signés
  `int` 32 bits signés
  `long` 64 bits signés
  `char` 16 bits non-signés
- Flottants
  `float` 32 bits simple-précision
  `double` 64 bits double précition
- Booléens
  - La spécification de la JVM définit un type booléen.
  - Représentés par des entiers (1 pour true et 0 pour false)
- Adresses de code: non-modifiables par le programme.

Les types des références (pointeurs)

- 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.

Appel de méthode

- variables locales sur 32 bits.
- Les données de type long ou double utilisent deux variables locales.
- Accès aux variables locales par leur position dans le bloc d'activation:
  - Indice 0: `this` (si pas static)
  - ensuite arguments effectifs des appels
  - ensuite valeurs variables locales
- La taille maximale d'un bloc d'activation nécessaire à l'exécution d'un code donné est calculable.



Inspection d'un fichier `.class`

Avec `javap -c -verbose <nome_code-octet>`
Par exemple, sur le `.class` de:

```
class A {
    int a;

    A() { a = 32; }

     int f() {
        int i = 43 ;
        return (a + i);
    }
}
```

On obtient
(voir Chapitre 4: The `class` File Format du la spécif de jvm):

```
Classfile .....//A.class
  Last modified ......
  MD5 checksum ......
  Compiled from "A.java"
class A
  minor version: 0
  major version: 52
  flags: ACC_SUPER
Constant pool:
   #1 = Methodref          #4.#15         // java/lang/Object."<init>":()V
   #2 = Fieldref           #3.#16         // A.a:I
   #3 = Class              #17            // A
   #4 = Class              #18            // java/lang/Object
   #5 = Utf8               a
   #6 = Utf8               I
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               f
  #12 = Utf8               ()I
  #13 = Utf8               SourceFile
  #14 = Utf8               A.java
  #15 = NameAndType        #7:#8          // "<init>":()V
  #16 = NameAndType        #5:#6          // a:I
  #17 = Utf8               A
  #18 = Utf8               java/lang/Object
```

```
{
  int a;
    descriptor: I
    flags:

  A();
    descriptor: ()V
    flags:
    Code:
      stack=2, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: aload_0
         5: bipush        32
         7: putfield      #2                  // Field a:I
        10: return
      LineNumberTable:
        line 4: 0

  int f();
    descriptor: ()I
    flags:
    Code:
      stack=2, locals=2, args_size=1
         0: bipush        43
         2: istore_1
         3: aload_0
         4: getfield      #2                  // Field a:I
         7: iload_1
         8: iadd
         9: ireturn
      LineNumberTable:
        line 7: 0
        line 8: 3
}
```

Les constantes

L'espace des constantes contient (presque) tout ce qui est statiquement utilisé dans la classe:

- Les valeurs flottantes
- Les chaines de caractères
- Les références aux autres classes
- Les références à toutes les méthodes utilisées
- Les références aux champs
- Les types de données
- ...

Cela permet d'y faire référence dans le code des fonctions et méthodes de façon compacte.

Jeu d'instructions

`en.wikipedia.org/wiki/Java_bytecode_instruction_listings`

Les instructions de la JVM sont typées.

- Le nom de chaque opération est préfixé 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 référence
- Par ailleurs, les opcodes sont stockés sur un seul octet.

Instructions: lecture/écriture de variables

On écrit " x" en place de i, l, s, b, c, f, d et a.
`xload n`: charge une variable au sommet de la pile.
`xload_n`: quand n est petit.
`xstore n`: écrit le sommet de la pile dans la variable n.
`xstore_n`: quand n est petit.

Constantes:
`bipush n`, `sipush n`: place l'entier n au sommet de la pile
`ldc n`, `ldc2_w n`: place le `float`/`double`/`long` rangé dans la constante n au sommet de la pile
`xconst_n`: quand n est petit.
`aconst_null`: place un pointeur nul sur 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.

Exemple (1)

```
class A {
  void f() {
      int i, j;
      i = 43;
      j = i;
    }
}
```

```
...
void f();
  flags:
  Code:
    stack=1, locals=3, args_size=1
       0: bipush 43
       2: istore_1
       3: iload_1
       4: istore_2
       5: return
...
```

Exemple (2)

```
class A {
  int f(int[] t, int i){
    return t[i];
  }
}
```

```
...
int f(int[], int);
  flags:
  Code:
    stack=2, locals=3, args_size=3
       0: aload_1
       1: iload_2
       2: iaload
       3: ireturn
...
```

- Le résultat du `ireturn` est placé en tête de la pile de calcul de la function appelant `f`.

Exemple (3)

```
class A {
    void f() {
      float i, j;
      i = 2;
      j = i;
    }
}
```

```
...
void f();
  flags:
  Code:
    stack=1, locals=3, args_size=1
       0: fconst_2
       2: fstore_1
       3: fload_1
       4: fstore_2
       5: return
...
```

Exemple (4)

```
class A {
  void f() {
    long i, j;
    i = 2;
    j = i;
  }
}
```

```
Constant pool:
 #1 = Methodref #5.#13; //java/lang/Object."<init>
 #2 = Long  2l;
 #4 = Class #14;  // A
 #5 = Class #15; //java/lang/Object
 #6 = Utf8    <init>;
...

void f();
  flags:
  Code:
    stack=2, locals=5, args_size=1
       0: ldc2_w  #2; // long 2l
       3: lstore_1
       4: lload_1
       5: lstore_3
       6: return
```

Instructions: opérations arithmétiques

`xadd`: Addition.
`xsub`: Soustraction.
`xmul`: Multiplication.
`xdiv`: Division.
`xrem`: Reste de la division.
`xneg`: Négation.

Décalage : `ishl`, `ishr`, `iushr`, `ishl`, `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 : `iinc`.
Comparaison : `dcmpg`, `dcmpl`, `fcmpg`, `fcmpl`, `lcmp`.

Exemple (5)

```
class A {
  public int f(int i) {
    int j = 12;
    ++j;
    return (i + 34);
  }
}
```

```
int f(int);
  flags: AC_PUBLIC
  Code:
    stack=2, locals=3, args_size=2
       0: bipush    12
       2: istore_2
       3: iinc      2, 1
       6: iload_1
       7: bipush    34
       9: iadd
      10: ireturn
```

Instructions: autres opérations

Conversion de types de donnée:

- élargissement : `i2l`, `i2f`, `i2d`, `l2f`, `l2d`, `f2d`.
- projection : `i2b`, `i2c`, `i2s`, `l2i`, `f2i`, `d2i`, `d2f`

Opération sur la pile:

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

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`, ...

Exemple (6)

```
class A {
  void f() {
     int i;
     float j;
     i = 20;
     j = 10 + i;
  }
}
```

```
...
 void f();
    flags:
    Code:
      stack=2, locals=3, args_size=1
         0: bipush 20
         2: istore_1
         3: bipush 10
         5: iload_1
         6: iadd
         7: i2f
         8: fstore_2
         9: return
...
```

Exemple (7)

```
class A {
  boolean f(int i) {
    if (i == 1) {
      return true;
    }
    else {
      return false;
    }
  }
}
```

```
...
  boolean f(int);
    flags:
    Code:
      stack=2, locals=2, args_size=2
         0: iload_1
         1: iconst_1
         2: if_icmpne 7
         5: iconst_1
         6: ireturn
         7: iconst_0
         8: ireturn
...
```

Exemple (8)

```
class A {
    int f(int i) {
        int j = 0;
        int r = 0;
        while (j != i) {
            r += j;
        }
        return r;
    }
}
```

```
...
  int f(int);
    flags:
    Code:
      stack=2, locals=4, args_size=2
         0: iconst_0
         1: istore_2
         2: iconst_0
         3: istore_3
         4: iload_2
         5: iload_1
         6: if_icmpeq   16
         9: iload_3
        10: iload_2
        11: iadd
        12: istore_3
        13: goto    4
        16: iload_3
        17: ireturn
...
```

Instructions: objets, tableaux, méthodes

- Création d'une nouvelle instance de classe : `new`.
- Création d'un tableau : `newarray`, `anewarray`, `multianewarray`.
- Accès aux champs d'une classe : `getfield`, `setfield`, `getstatic`, `putstatic`.
- Chargement d'un tableau sur la pile de calcul : `xaload`,
- Affectation d'une case d'un tableau : `xastore`,
- Empile la taille d'un tableau : `arraylength`.
- Vérification dynamique : `instanceof`, `checkcast`.
- Invoquer une méthode avec liaison tardive : `invokevirtual`.
- Invoquer une méthode de classe statique : `invokestatic`.
- Invoquer une méthode d'interface : `invokeinterface`.
- Invoquer une initialisation d'instance, une méthode privée ou une méthode d'une classe mère : `invokespecial`.

Exemple (9)

```
class B {
  int x;

  B() { x = 67; }

  int get() {
    return x;
  }
}

class A {
  void f() {
     int i;
     B b = new B();
     i = b.get();
   }
}
```

le fichier A.class contient:

```
Constant pool:
 #1 = Methodref #6.#14;//java/lang/Object."<init>":()V
 #2 = Class   #15; // B
 #3 = Methodref #2.#14; // B."<init>":()V
 #4 = Methodref #2.#16; // B.get:()I
 #5 = Class   #17; // A
 #6 = Class   #18; // java/lang/Object
 #7 = Utf8 <init>;
...
void f();
  flags:
   Code:
    stack=2, locals=3, args_size=1
      0: new #2;
      3: dup
      4: invokespecial #3;
      7: astore_2
      8: aload_2
      9: invokevirtual #4;
     12: istore_1
     13: return
```

Ocamlrun versus JVM

OCamlrun	JVM
fonctions de 1ère classe	méthodes + dispatch dynamique
données structurées	objets
bytecode interprété	compilation just-in-time
bytecode non typé	bytecode typé statiquement
distinction pointeur/valeur directe	objets portent leur type
optimisation appels terminaux	pas d'optimisation
GC non concurrent	primitives de concurrence