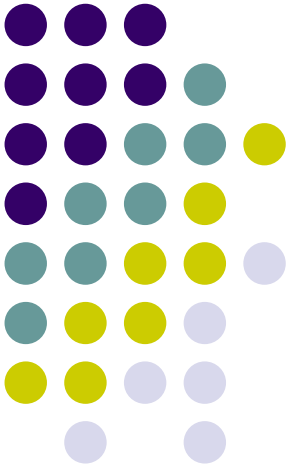
M1 – Middleware Adaptable

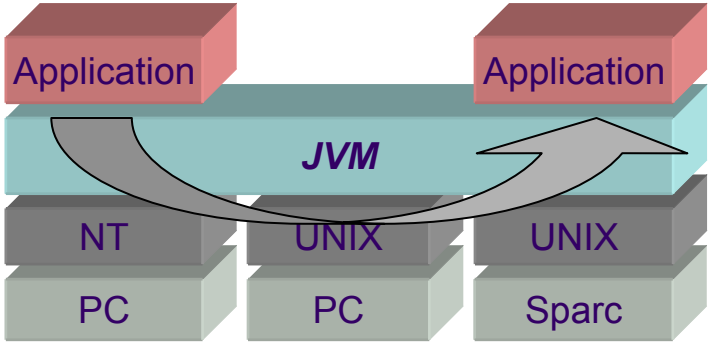
JVM
Adaptation de *bytecode*

Sara Bouchenak

Sara.Bouchenak@inria.fr
http://sardes.inrialpes.fr/~bouchena/teaching

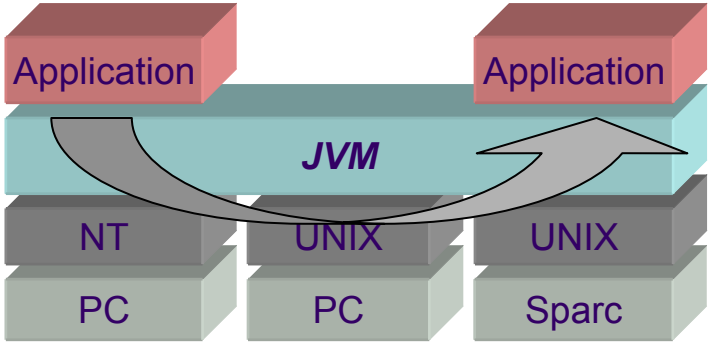


JVM



- Abstraction d'un environnement homogène
- Cacher l'hétérogénéité des
  - systèmes d'exploitation
  - systèmes de communication
  - architectures physiques sous-jacentes

Motivations d'un environnement homogène



- Portabilité des applications, du code
- Facilité de construction d'applications réparties

Plan

1. *JVM : une infrastructure middleware pour environnements hétérogènes*
   - *Qu'est-ce que la JVM ?*
   - *Spécification abstraite*
   - *Mise en œuvre concrète*
   - *Instance d'exécution*
2. Exemple d'illustration
3. BCEL : adaptation de bytecode Java
4. Synthèse et Références

Qu’est-ce que la JVM ?

- JVM (*Java Virtual Machine*)

- Machine virtuelle
  - Machine abstraite définie par une spécification

  - Nécessite une mise en œuvre concrète pour l’exécution des programmes

JVM : une des trois entités …

- Spécification abstraite
- Mise en œuvre concrète
- Instance d’exécution

Spécification de la JVM : Objectif

- Définir
  - un comportement externe
  - commun aux différentes mises en œuvre de la machine virtuelle Java

Spécification de la JVM : Objectif

- Définition de l'abstraction d'une machine de calcul :
  - Ensemble d'instructions
  - Effectuer des calculs numériques
  - Contrôler l'accès à la mémoire
  - Gérer le contrôle des flots d'exécutions (processus / threads)
  - Moyen d'accéder au matériel sous-jacent

Spécification de la JVM

- Comportement de la machine virtuelle en termes de :
  - Types de données
  - Sous-systèmes d'exécution
  - Structures de données d'exécution

Types de données (1 / 2)

- Types références
  - Exemples : référence d'instance de classe, de tableau

- Types primitifs
  - Taille des types spécifiée
  - Format de codage des types spécifié
  - Exemples : int, float, boolean, etc.

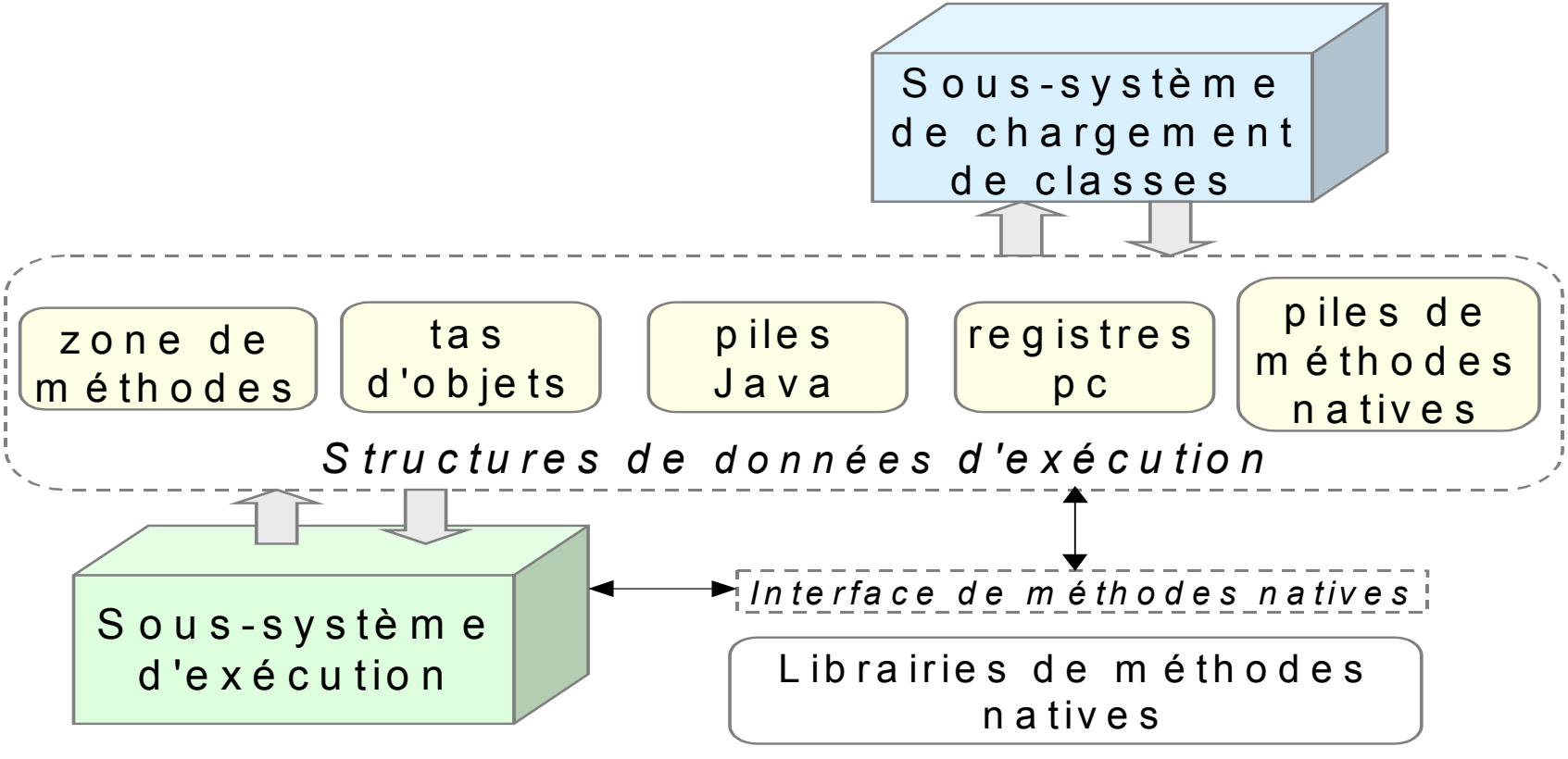
Types de données (2 / 2)

	Type	Taille	Description
Types primitifs entiers	byte	8 bits	entiers signés en complément à 2
	short	16 bits	
	int	32 bits	
	long	64 bits	
Types primitifs flottants	float	32 bits	réels au format IEEE 754
	double	64 bits	
Autres types primitifs	char	16 bits	entier non signé représentant un caractère Unicode
	boolean	non spécifié	entier 1 pour *vrai* et entier 0 pour *faux*
	returnAddress	non spécifié	pointeur vers instruction
Types références	reference	non spécifié	référence vers une instance de classe ou un tableau

Spécification de la JVM

- Comportement de la machine virtuelle en termes de :
  - Types de données
  - *Sous-systèmes d'exécution*
  - Structures de données d'exécution

Sous-systèmes d'exécution



Sous-système de chargement de classes

- Objectif :
  Lors de l'exécution d'un programme :
  - Charger en mémoire les classes et interfaces utilisées
  - Les stocker dans la *zone de méthodes* de la JVM

- Plusieurs chargeurs de classes peuvent cohabiter au sein d'une même JVM
  - Chargeur à partir du CLASSPATH (*ClassLoader*)
  - Chargeur à partir d'URL (*URLClassLoader*)

URLClassLoader

URLClassLoader

public **URLClassLoader**(URL[] urls)

Constructs a new URLClassLoader for the specified URLs using the default delegation parent ClassLoader. The URLs will be searched in the order specified for classes and resources after first searching in the parent class loader. Any URL that ends with a '/' is assumed to refer to a directory. Otherwise, the URL is assumed to refer to a JAR file which will be downloaded and opened as needed.
If there is a security manager, this method first calls the security manager's checkCreateClassLoader method to ensure creation of a class loader is allowed.

Parameters:

urls - the URLs from which to load classes and resources

Throws:

`SecurityException` - if a security manager exists and its checkCreateClassLoader method doesn't allow creation of a class loader.

See Also:

SecurityManager.checkCreateClassLoader()

Définition d'un nouveau chargeur de classes (1 / 2)

For example, an application could create a network class loader to download class files from a server. Sample code might look like:

```
ClassLoader loader = new NetworkClassLoader(host, port);

Object main = loader.loadClass("Main", true).newInstance();

        . . .
```

Définition d'un nouveau chargeur de classes (2 / 2)

```
class NetworkClassLoader extends ClassLoader {
   String host;
   int port;

   public Class findClass(String name) {
      byte[] b = loadClassData(name);
      return defineClass(name, b, 0, b.length);
   }

   private byte[] loadClassData(String name) {
      // load the class data from the connection
       . . .
   }
}
```

Sous-système d'exécution

- Mécanisme d'exécution du bytecode contenu dans les méthodes des classes chargées

- Trois formes :
  - Spécification abstraite : ensemble d'instructions
  - Mise en œuvre concrète : différentes techniques
  - Instance d'exécution : thread Java

Sous-système d'exécution : Ensemble d'instructions (1 / 4)

- Compilateur
  - Traduit un programme écrit dans un langage source vers un langage cible
  - Traditionnellement, le langage cible est le langage d'un processeur physique
  - Programme compilé compris par toute machine de même architecture, pas par les autres architectures

- Compilateur Java
  - Traduit un programme écrit dans le langage source Java vers un langage cible appelé *bytecode*
  - le langage cible est le langage d'un processeur virtuel (JVM)
  - Programme Java compilé compris par toute machine virtuelle, quelle que soit l'architecture physique sous-jacente

Sous-système d'exécution : Ensemble d'instructions (2 / 4)

- Méthode Java :
  suite d'instructions de bytecode

- Instruction de bytecode :
  opcode + un ou plusieurs opérandes

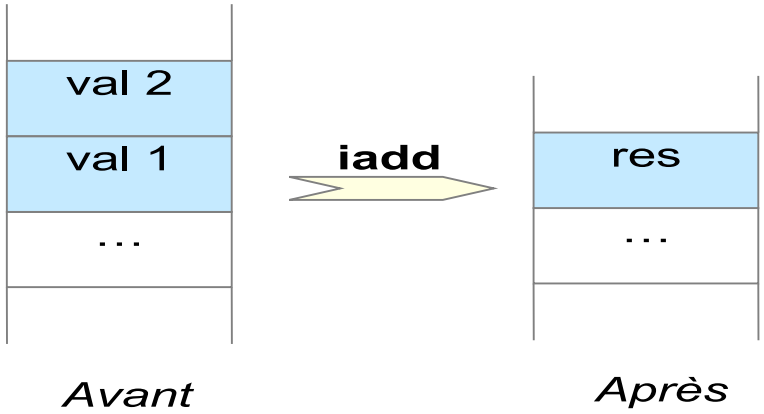
- Opcode :
  - codé sur un octet (*byte*)
  - spécifie l'opération à effectuer
  - indique s'il est suivi d'un ou plusieurs opérandes
  - indique le type des opérandes

Sous-système d'exécution : Ensemble d'instructions (3 / 4)

- Exemples d'opcodes :
  - ***i****add*
    - opération d'addition
    - opcode suivi de deux opérandes
    - opérandes de type *int*
  - ***f****sub*
    - opération de soustraction
    - opcode suivi de deux opérandes
    - opérandes de type *float*

Sous-système d'exécution : Ensemble d'instructions (4 / 4)

- Instructions : approche / structure de pile



- Avant l'exécution de *iadd*
  - val1, val2 : deux valeurs de type *int* en sommet de pile
- Exécution de *iadd*
  - val1 et val2 dépilées, additionnées, résultat de type *int* en sommet de pile

Sous-système d'exécution

- Mécanisme d'exécution du bytecode contenu dans les méthodes des classes chargées

- Trois formes :
  - Spécification abstraite : ensemble d'instructions
  - *Mise en œuvre concrète : différentes techniques*
  - Instance d'exécution : thread Java

Sous-système d'exécution : Mise en œuvre concrète (1 / 5)

- Différentes techniques
  - Interprétation
  - Compilation à la volée (*Just-In-Time compiler*)
  - Optimisation adaptative
  - Exécution native

Sous-système d'exécution : Mise en œuvre concrète (2 / 5)

- Interprétation
  - Première génération de JVM
  - Interprète Java
    - *loop*
      - lire une instruction de bytecode
      - la traduire en suite d'instructions-machine correspondant à l'architecture sous-jacente
      - exécuter les instruction machine
  - Avantage / Inconvénients
    - ☺ Simplicité de mise en œuvre
    - ☹ Lenteur de l'exécution comparée au code natif

Sous-système d'exécution : Mise en œuvre concrète (3 / 5)

- Compilation à la volée / JIT
  - Seconde génération de JVM
  - Principe
    - Si appel d'une méthode Java la première fois
      - méthode compilée vers code natif
      - exécution de la version native
    - Appel ultérieur de la même méthode Java
      - exécution de la version native
  - Avantage / Inconvénients
    - ☺ Temps d'exécution d'un code natif
    - ☹ Surcoût dû à la compilation de méthodes rarement appelées (moins coûteux d'exécuter du Java que de compiler puis exécuter du natif)

Sous-système d'exécution : Mise en œuvre concrète (4 / 5)

- Optimisation adaptative
  - Combine l'interprétation Java et la compilation à la volée
  - Principe
    - Initialement : interprétation Java
    - Surveillance de l'exécution : informations sur la fréquence d'appel des méthodes
    - Optimisation adaptative : utilisation des info.s pour choisir les méthodes à compiler puis exécuter en natif
  - Exemples de JVM proposant l'optimisation adaptative
    - JDK 1.3 et + de Sun (HotSpot VM)
    - Jikes de IBM (Jalapeño VM)

Sous-système d'exécution : Mise en œuvre concrète (5 / 5)

- Exécution native
  - Principe
    - Initialement : compilation du programme Java en programme natif
    - Exécution de tout le programme en version native
  - Avantage / Inconvénients
    - ☺ Temps d'exécution d'un code natif
    - ☺ Pas de surcoût dû à la compilation
    - ☹ Moins de portabilité

Sous-système d'exécution

- Mécanisme d'exécution du bytecode contenu dans les méthodes des classes chargées

- Trois formes :
  - Spécification abstraite : ensemble d'instructions
  - Mise en œuvre concrète : différentes techniques
  - *Instance d'exécution : thread Java*

Sous-système d'exécution : Instance d'exécution

- Instance d'exécution de la JVM = thread Java
- JVM supporte l'exécution de plusieurs threads (processus légers)
- Rôle du thread
  - exécuter le code d'une application
  - bytecode ou code natif

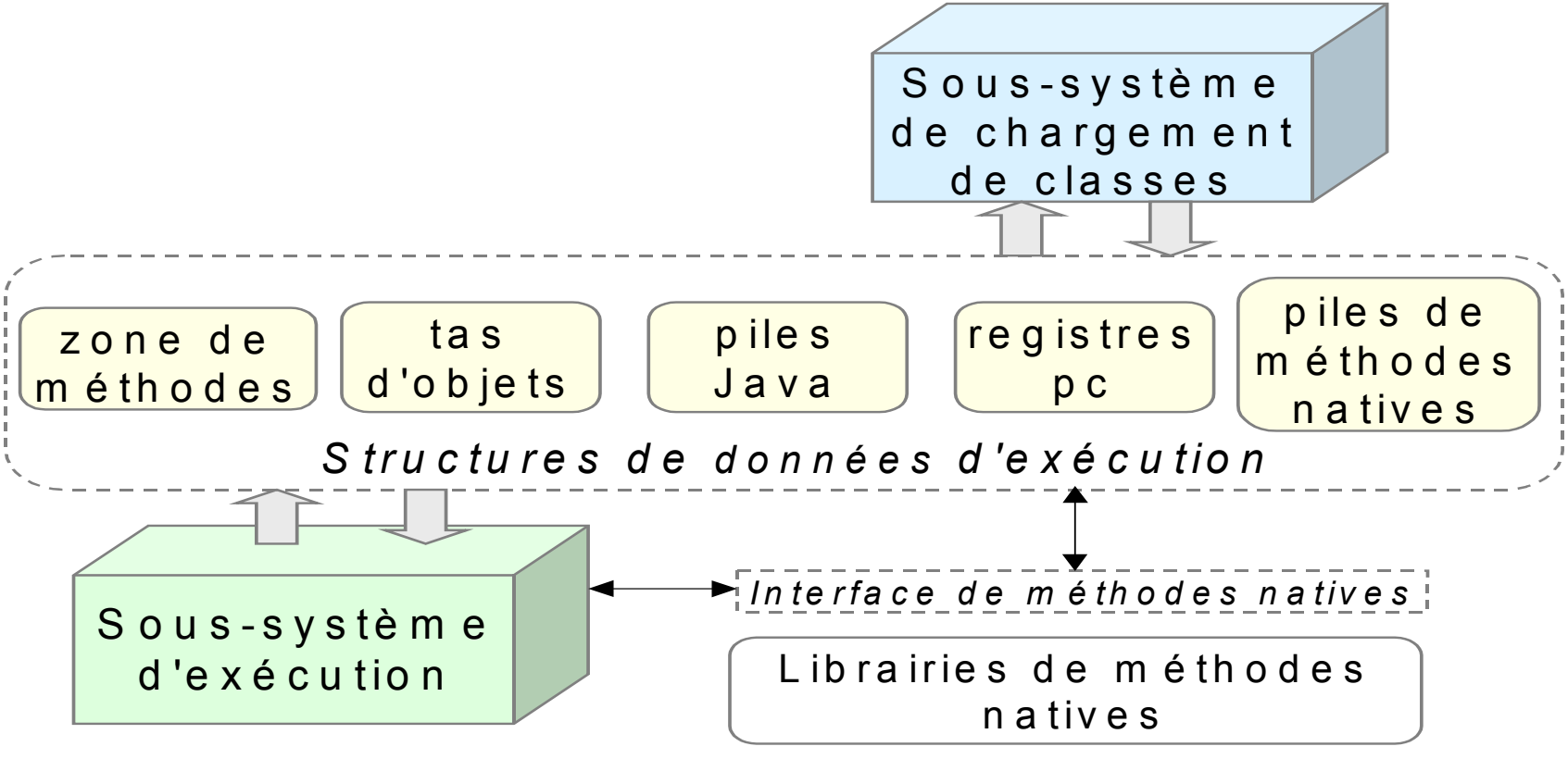
Spécification de la JVM

- Comportement de la machine virtuelle en termes de :
  - Types de données
  - Sous-systèmes d'exécution
  - *Structures de données d'exécution*

Structures de données d'exécution

- Pour exécuter un programme, la JVM a besoin de mémoire pour stocker :
  - le code des classes utilisées
  - les objets instanciés
  - les info.s des méthodes exécutées (paramètres, variables locales, résultat de calculs intermédiaires)
  - la table des symboles utilisés par le programme

Structures de données d'exécution



Zone de méthodes (1 / 3)

- Une instance de JVM possède une zone de méthodes partagée par tous les threads de la JVM
- Toute classe Java chargée est placée dans la zone de méthodes
- Tout thread s'exécutant dans la JVM peut accéder à toute classe chargée dans cette JVM

Zone de méthodes (2 / 3)

- Informations relatives à une classe dans la zone de méthodes
  - variable de classes (variables *static*)
  - Informations sur les méthodes de la classe
  - *constant pool* de la classe

Zone de méthodes (3 / 3)

- Informations relatives à une méthode de classe dans la zone de méthodes
  - nom de la méthode
  - type de son résultat
  - types de ses paramètres
  - ses modificateurs (*public*, *private*, *protected*, *static*, *final*, *synchronized*, *native*, *abstract*)

  - bytecode de la méthode
  - table d'exceptions
  - table des variables locales

Pour une méthode non abstract et non native

Constant pool

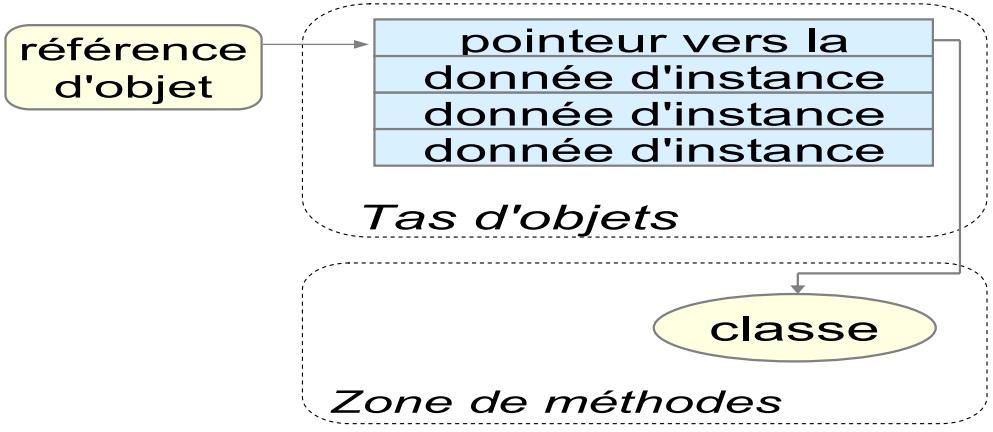
- Table de symboles de constantes utilisées par la classe
- Constante utilisée par une classe participe à la définitions de :
  - la classe
  - variables de classe ou d'instance
  - descripteur ou code des méthodes
- Forme des constantes
  - valeurs de chaînes de caractères
  - valeurs de constantes entières ou flottantes
  - noms symboliques de classes, interfaces, méthodes, variables de classe ou d'instance

Tas d'objets (1 / 2)

- Une instance de JVM possède un tas d'objets partagé par tous les threads de la JVM
- Toute instance de classe ou de tableau créée est placée dans le tas d'objets
- Tout thread s'exécutant dans la JVM peut accéder à tout objet créé dans cette JVM

Tas d'objets (2 / 2)

- Représentation d'un objet
  - valeurs des variable d'instance (attributs de l'objet)
  - possibilité d'accès aux informations sur la classe de l'objet



Pile Java

- Une pile Java est associée à chaque thread, lors de la création du thread
- Décrit l'état d'avancement des appels de méthodes Java effectués par le thread
- L'état d'un appel de méthode est représenté par une zone appelée *frame* Java

Frame Java

- Appel de méthode Java par un thread
  - un nouveau *frame* Java est créé et empilé sur la pile Java du thread
- Terminaison de méthode Java par un thread
  - frame Java associé à cette méthode est dépilé

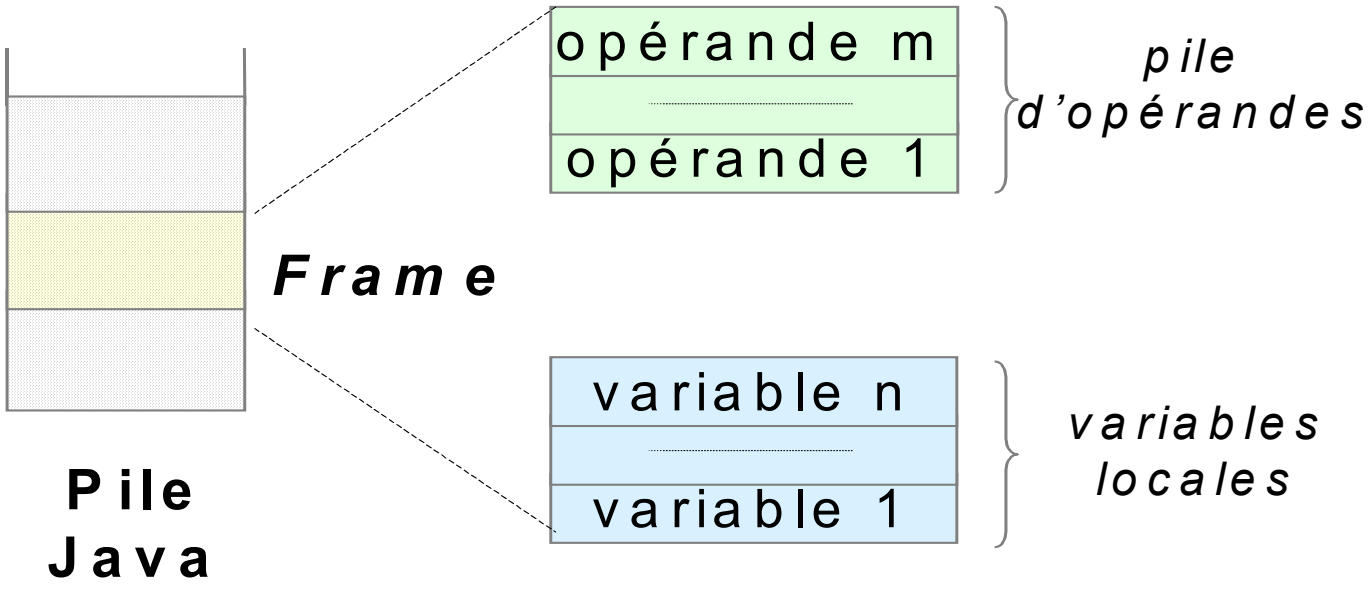
Structure d'un frame Java

- Table de variables locales
  - valeurs des paramètres de la méthode
  - valeurs des variables locales à la méthode

- Pile d'opérandes
  - résultats des calculs intermédiaires effectués lors de l'exécution de la méthode
  - manipulation de la pile d'opérandes par : empilement en sommet de pile ou dépilement de la tête de pile

- Autres informations
  - pointeur vers le constant pool de la classe de la méthode
  - autres informations dépendant de la mise en œuvre de la JVM

Frame Java sur pile Java



Registre PC

- Registre PC (*Program Counter*)
- Lorsque le thread exécute une méthode Java
  - PC indique l'adresse de la prochaine instruction de bytecode à exécuter
- Lorsque le thread exécute une méthode native
  - Valeur du PC est indéfinie

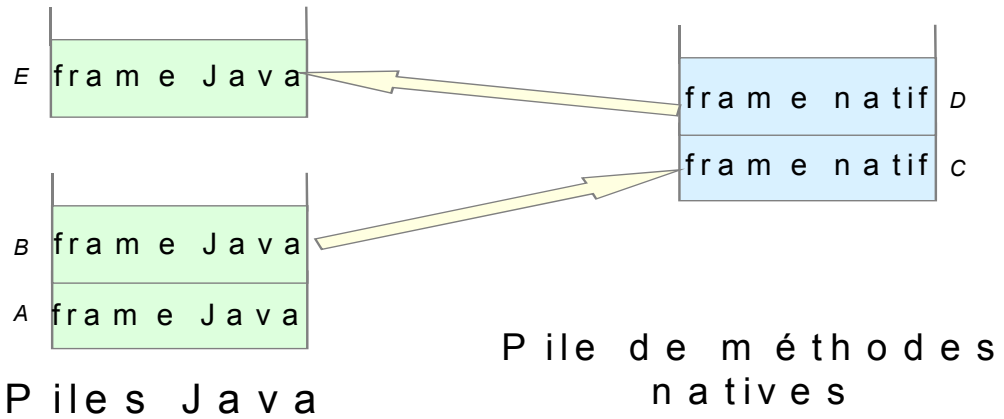
Méthodes natives vs. méthodes Java

- Méthode Java
  - Écrite dans le langage Java
  - Compilée vers du bytecode
  - Stockée dans un fichier *.class*
- Méthode native
  - Écrite dans un autre langage que Java (C/C++, assembleur)
  - Compilée vers le langage natif de la machine sous-jacente
  - Stockée dans une librairie chargée dynamiquement lors de l'invocation d'une méthode native par un programme Java
  - Interface JNI (*Java Native Interface*) : interaction entre code Java et code natif

Pile de méthodes natives

- Thread Java appelle une méthode native
  - Pile Java du thread laissée de côté
  - Utilisation de la *pile de méthodes natives*

- Structure de la pile de méthodes natives non définie par la spécification de la JVM



JVM : une des trois entités …

- Spécification abstraite
- *Mise en œuvre concrète*
- Instance d'exécution

Mise en œuvre concrète de la JVM

- Plusieurs mises en œuvre de JVM existent
- Pour plusieurs plates-formes
- Mises en œuvre
  - entièrement logicielles
  - combinaison de mises en œuvre logicielle et matérielle
- Exemples de mises en œuvre de JVM
  - JDK
  - Kaffe
  - picoJava
  - KVM
  - JavaCard

Exemples de mises en œuvre concrètes de JVM (1 / 2)

- JDK
  - fourni par Sun
  - sous-système d'exécution basé sur un interprète, compilation JIT et optimisation adaptative
  - utilisable sur plusieurs plates-formes : Linux, Solaris, Windows, Mac OS
- Kaffe
  - distribuée en logiciel libre
  - sous-système d'exécution basé sur un interprète, compilation JIT
  - utilisable sur plusieurs plates-formes : Linux, Solaris, Windows
- picoJava
  - fourni par Sun
  - mise en œuvre partiellement matérielle de la JVM
  - Microprocesseur avec le bytecode pour langage machine

Exemples de mises en œuvre concrètes de JVM (2 / 2)

- KVM
  - une JVM "light" pour terminaux mobiles (téléphones, PDA)
  - sous-système d'exécution basé sur un interprète
  - certaines restrictions par rapport au langage Java / API Java (types *float* et *double* non supportés)

- JavaCard
  - une JVM embarquée sur une carte à puce
  - sous-système d'exécution basé sur un interprète
  - un sous-ensemble du langage Java et de la JVM
  - pas de threads multiples, chargement dynamique de classes

JVM : une des trois entités …

- Spécification abstraite
- Mise en œuvre concrète
- *Instance d'exécution*

Instance d'exécution de la JVM

- Lorsqu'une application Java est lancée, une instance d'exécution de la JVM est créée
- L'application Java s'exécute au sein de cette instance
- Instance d'exécution de la JVM commence par l'exécution de la méthode *main* d'une classe
- Méthode main : point d'entrée pour le lancement du thread initial de l'application
- Par la suite, d'autres threads peuvent être créés
- Lorsque tous les threads se terminent, l'instance d'exécution de la JVM se termine

Plan

1. JVM : une infrastructure middleware pour environnements hétérogènes
2. *Exemple d'illustration*
3. BCEL : adaptation de bytecode Java
4. Synthèse et Références

Exemple d’illustration

- Exemple
  - Code source Java
  - Bytecode correspondant
  - Exécution et structure de données associées

Code source Java

```
class IntValue {

// Variable
int value;

// Constructor
IntValue(int initialValue) {
        value = initialValue;
}

// Methods
int getIntValue() {
        return value;
}

void setIntValue(int newValue)
{
        value = newValue;
}
```

```
void add(int plus) {
0       value =
1           value
2           + plus;
}

int sub(int minus) {

0       add(-minus);
}

// Main method
public static void main(
        String[] args) {
        IntValue val;

0       val = new IntValue(5);
1       val.sub(2);
}

}
```

Bytecode

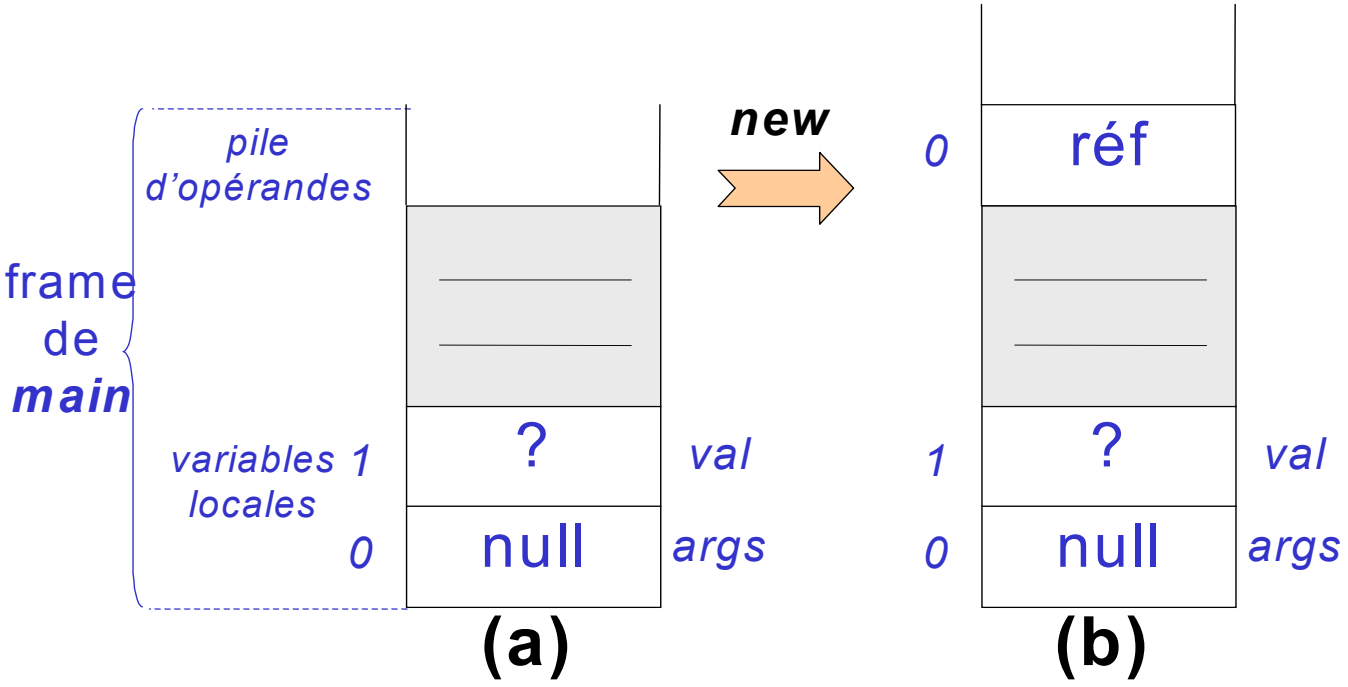
- Compilation
  - *javac IntValue.java*
    ↳ Génération du fichier *IntValue.class*

- Désassemblage
  - *javap –c IntValue.class*
    ↳ Génération du bytecode correspondant

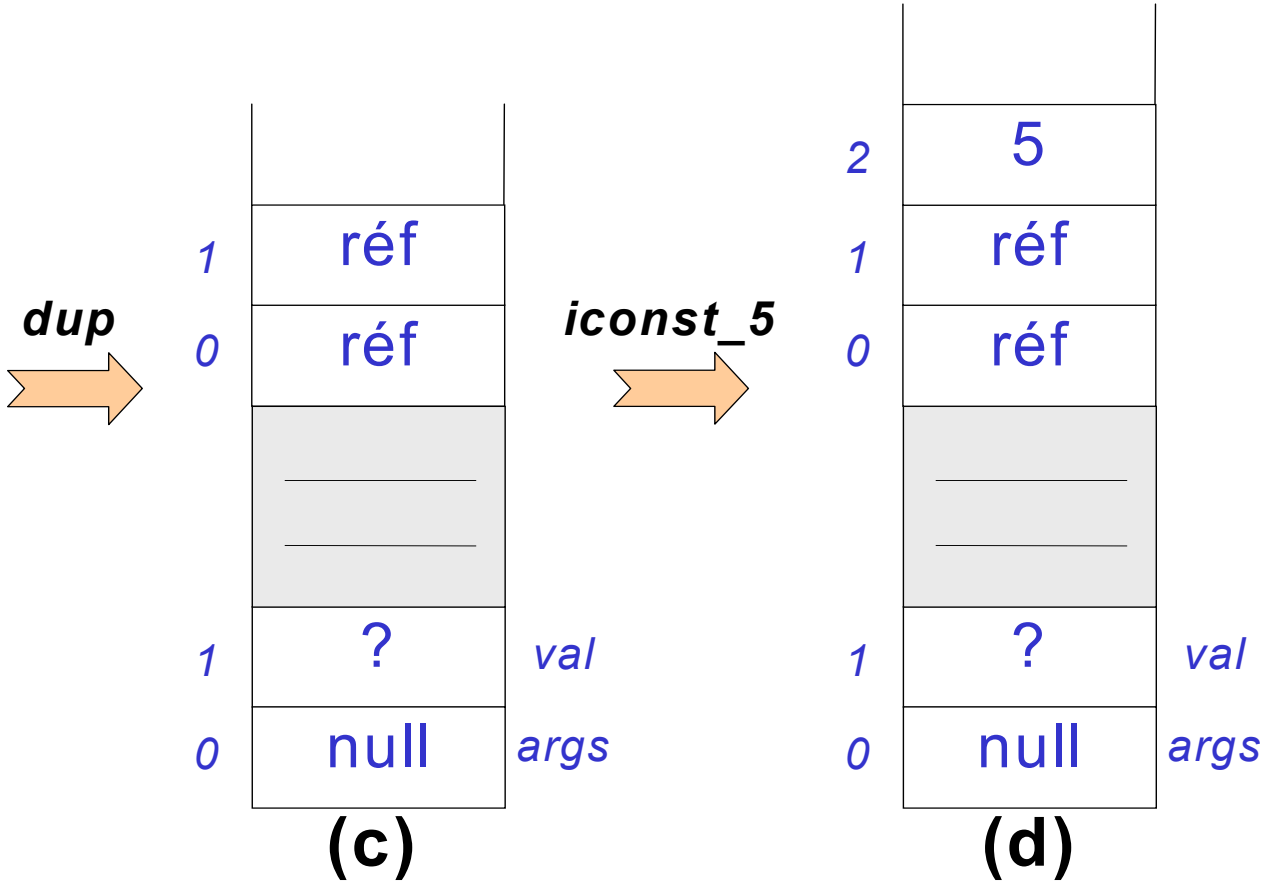
Bytecode

```
Method IntValue(int)
  0 aload_0
  1 invokespecial #3
      <Method
      java.lang.Object()>
  4 aload_0
  5 iload_1
  6 putfield #7 <Field int
      value>
  9 return

Method int
  getIntValue()
  0 aload_0
  1 getfield #7 <Field int
      value>
  4 ireturn

Method void
  setIntValue(int)
  0 aload_0
  1 iload_1
  2 putfield #7 <Field int
      value>
  5 return

Method void main(
  java.lang.String[])
  0 new #1 <Class
      IntValue>
  3 dup
  4 iconst_5
  5 invokespecial #4
      <Method
      IntValue(int)>
  8 astore_1
  9 aload_1
 10 iconst_2
 11 invokevirtual #6
      <Method
      void sub(int)>
 14 return

Method void sub(int)
  0 aload_0
  1 iload_1
  2 ineg
  3 invokevirtual #5
      <Method void
      add(int)>
  6 return

Method void add(int)
  0 aload_0
  1 dup
  2 getfield #7 <Field int
      value>
  5 iload_1
  6 iadd
  7 putfield #7 <Field int
      value>
 10 return
```

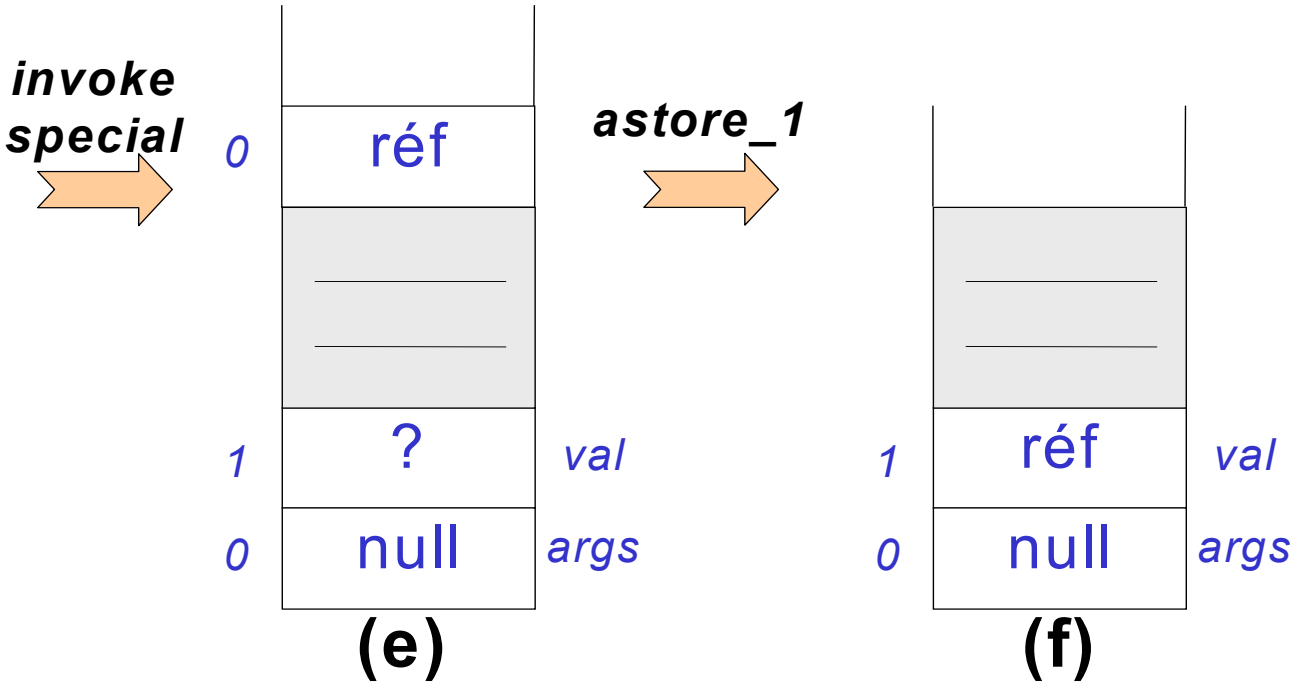
Exécution


pile
d'opérandes
frame
de
main
variables
locales
1
0
?
null
val
args
(a)
new
0
1
0
réf
?
null
val
args
(b)

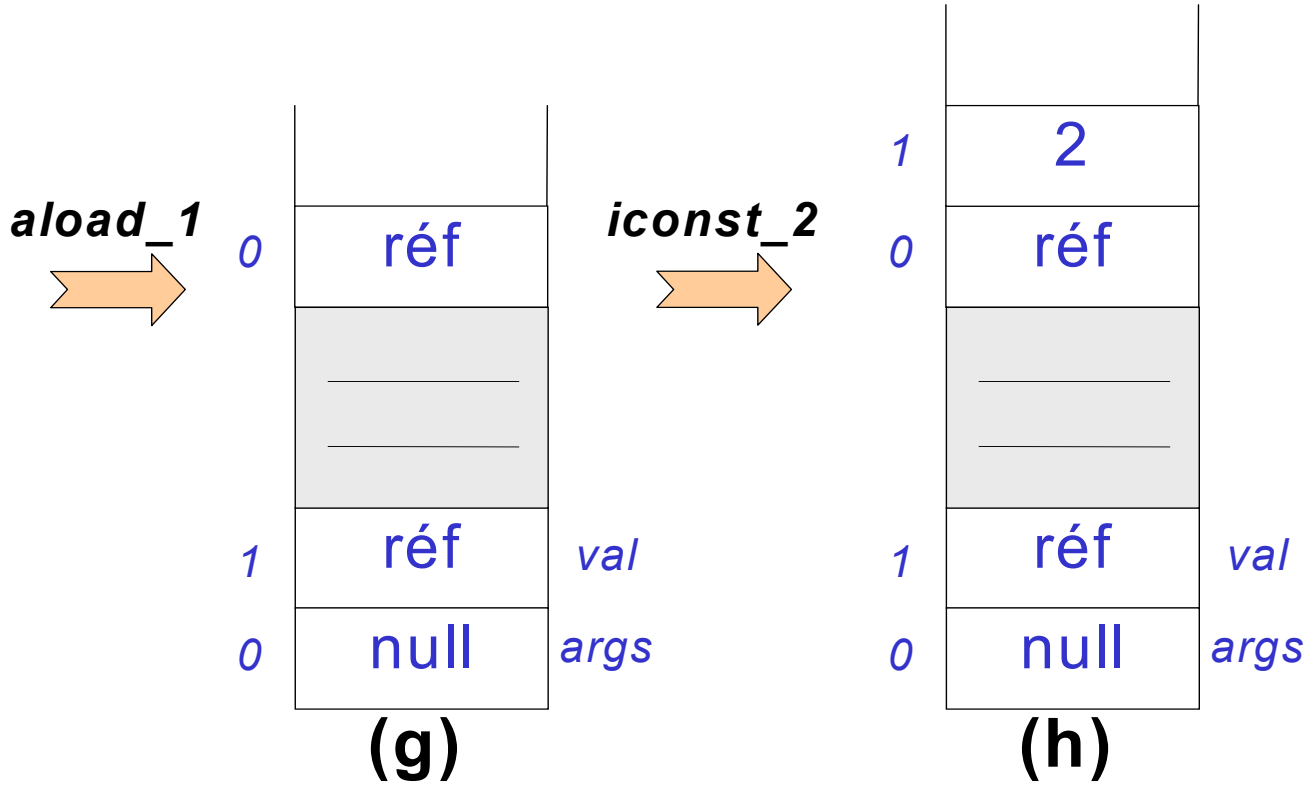
Exécution


dup
1
réf
0
réf
1
?
val
0
null
args
(c)
iconst_5
2
5
1
réf
0
réf
1
?
val
0
null
args
(d)

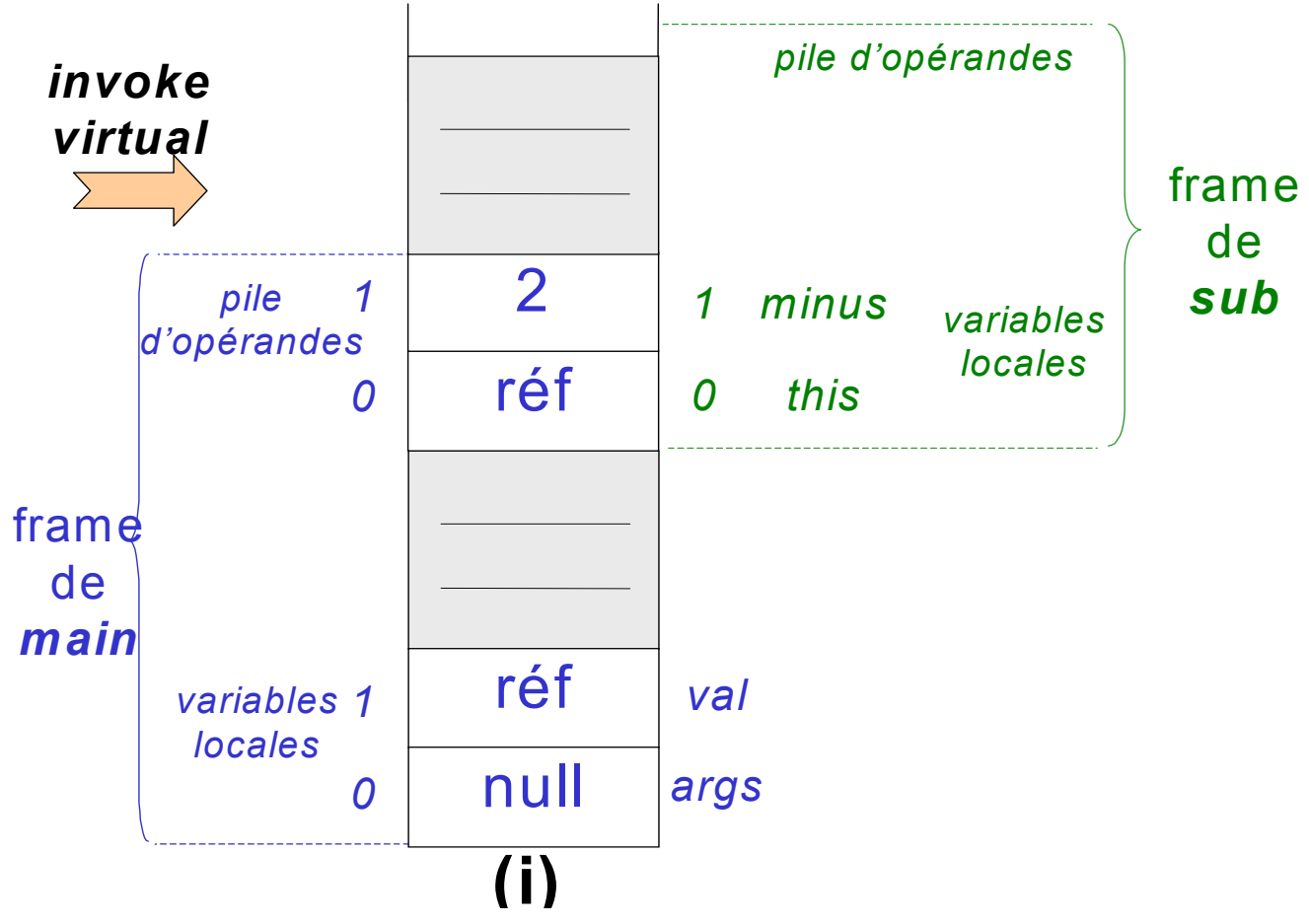
Exécution


invoke special
0
réf
1
?
val
0
null
args
(e)
astore_1
1
réf
val
0
null
args
(f)

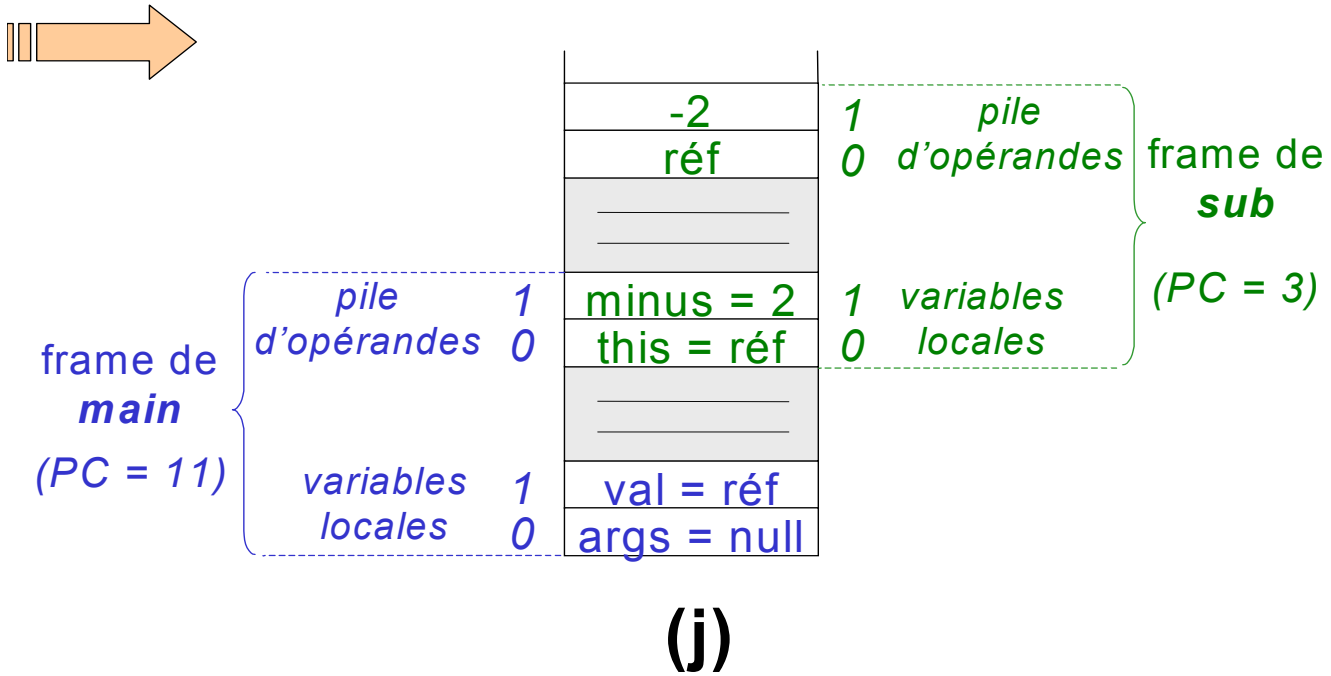
Exécution


aload_1
0
réf
1
réf
val
0
null
args
(g)
iconst_2
1
2
0
réf
1
réf
val
0
null
args
(h)

Exécution

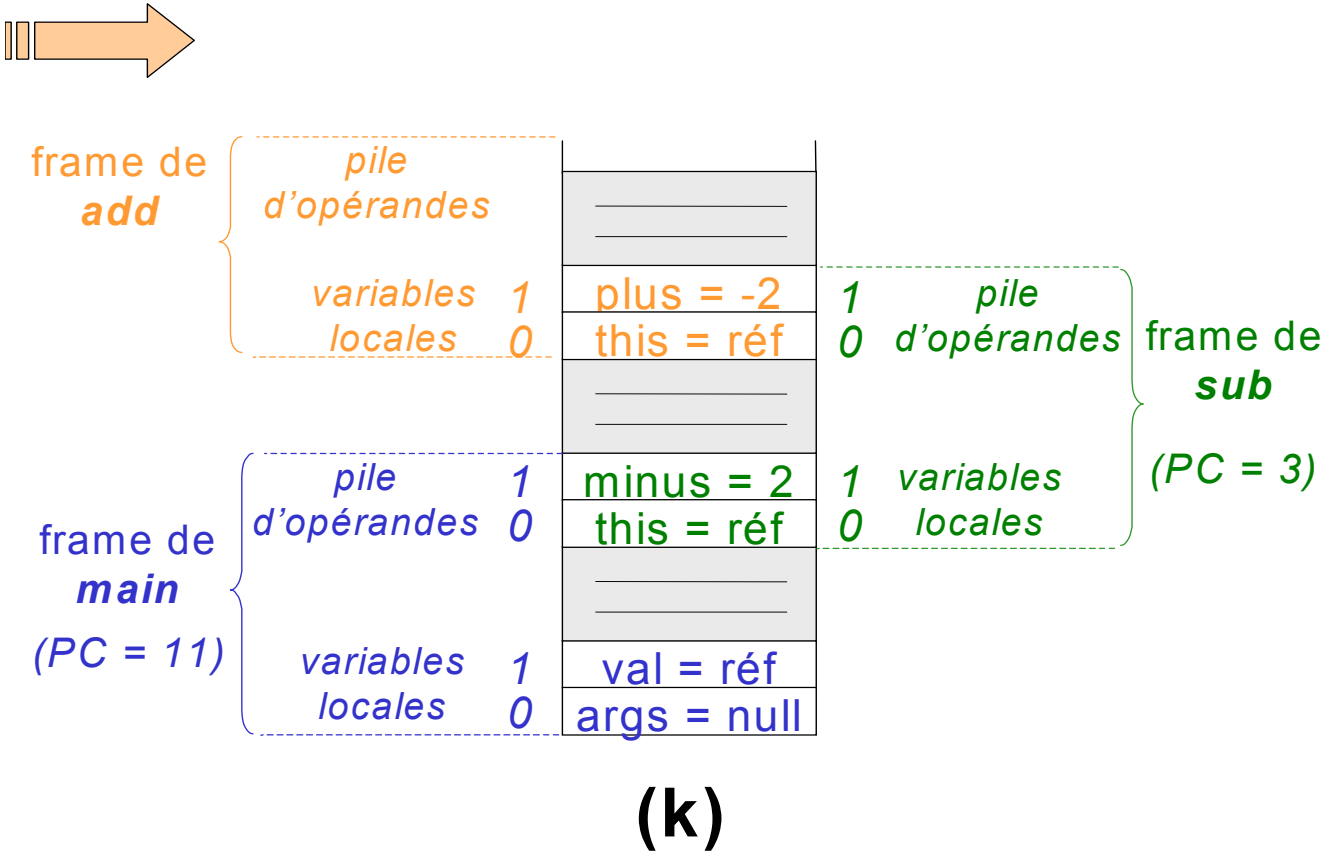

invoke
virtual
pile d'opérandes
frame
de
sub
2
réf
1 minus
0 this
variables
locales
pile
d'opérandes
1
0
frame
de
main
réf
null
variables
locales
1
0
val
args
(i)

Exécution


-2
réf
minus = 2
this = réf
val = réf
args = null
1
0
pile
d'opérandes
frame de
sub
1
0
variables
locales
(PC = 3)
pile
d'opérandes
1
0
frame de
main
(PC = 11)
variables
locales
1
0

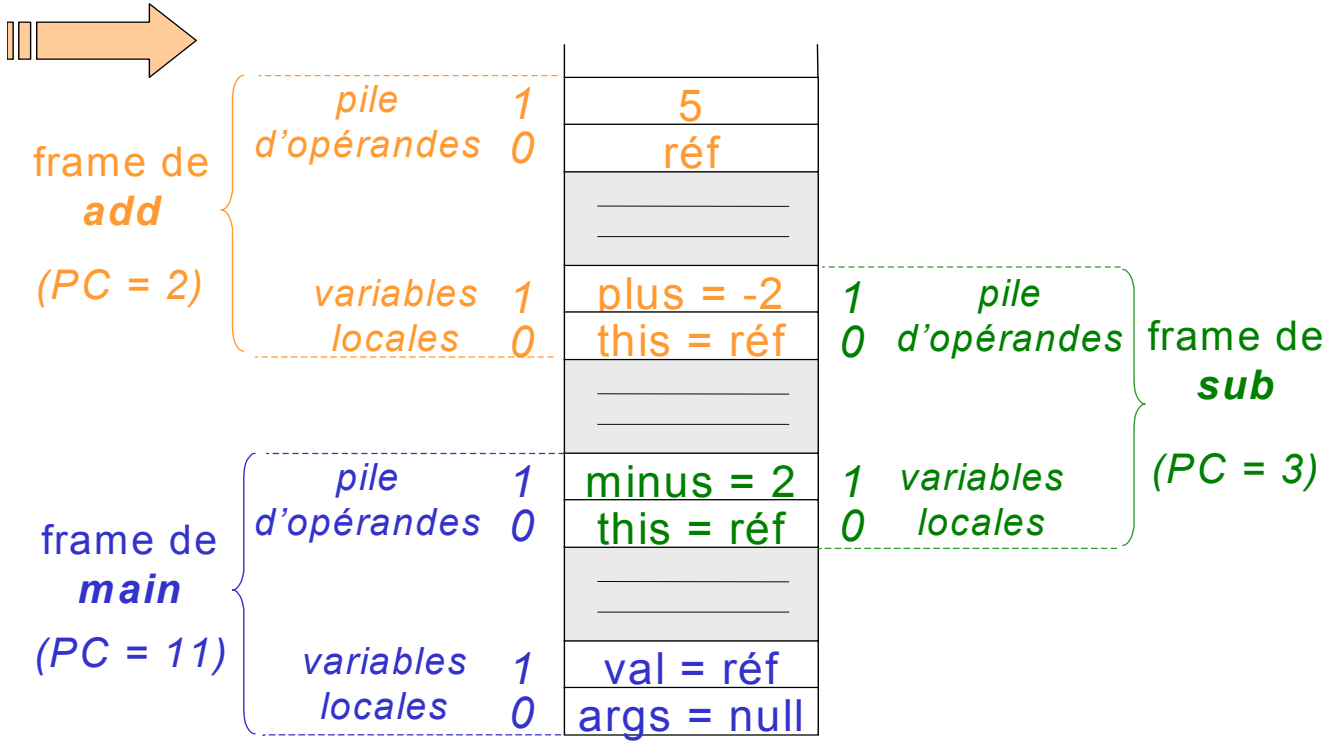
(j)

Exécution


frame de
add
pile
d'opérandes
variables
locales
1
0
plus = -2
this = réf
1
0
pile
d'opérandes
frame de
sub
(PC = 3)
minus = 2
this = réf
1
0
variables
locales
pile
d'opérandes
1
0
frame de
main
(PC = 11)
variables
locales
1
0
val = réf
args = null

(k)

Exécution


frame de
add
(PC = 2)
pile
d'opérandes
1
0
variables
locales
1
0
5
réf
plus = -2
this = réf
1
0
pile
d'opérandes
frame de
sub
(PC = 3)
1
0
variables
locales
frame de
main
(PC = 11)
pile
d'opérandes
1
0
variables
locales
1
0
minus = 2
this = réf
val = réf
args = null

(I)

Plan

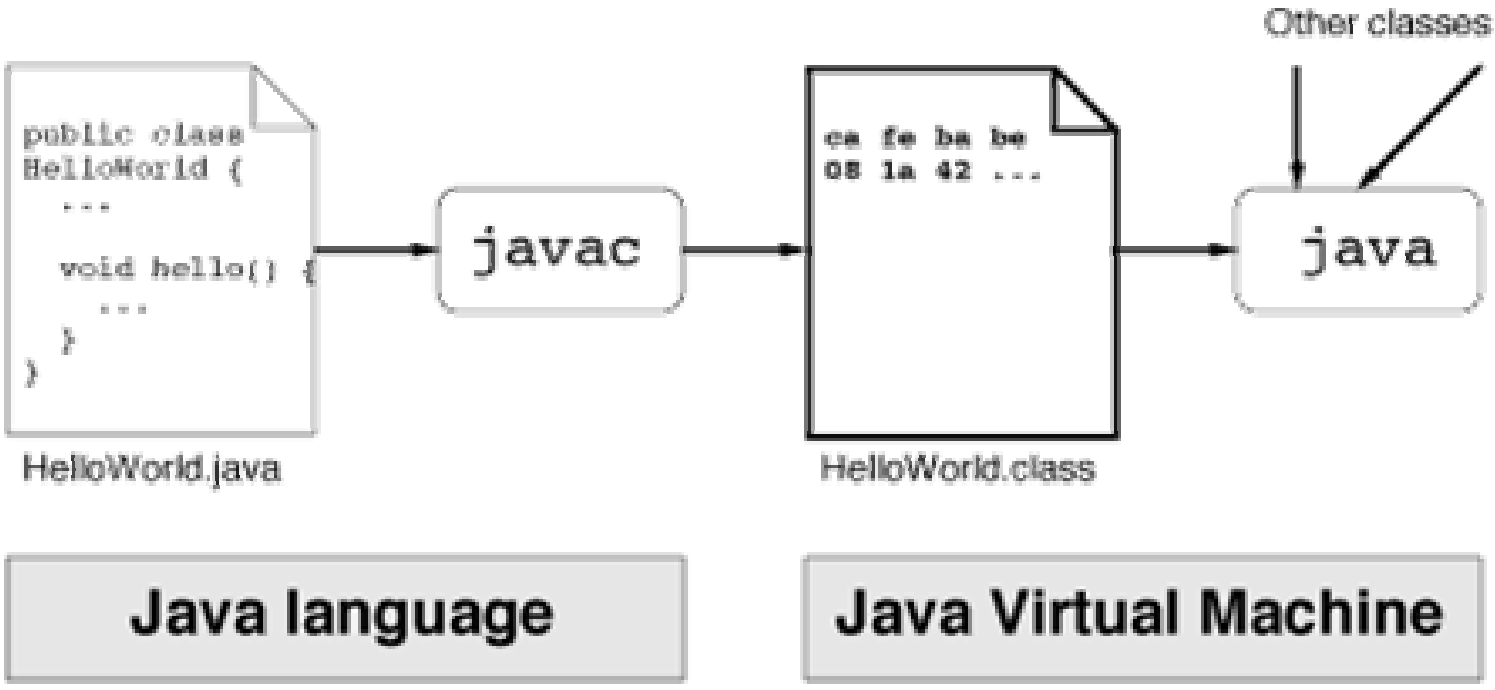
1. JVM : une infrastructure middleware pour environnements hétérogènes
2. Exemple d'illustration
3. *BCEL : adaptation de bytecode Java*
4. Synthèse et Références

BCEL : Motivations

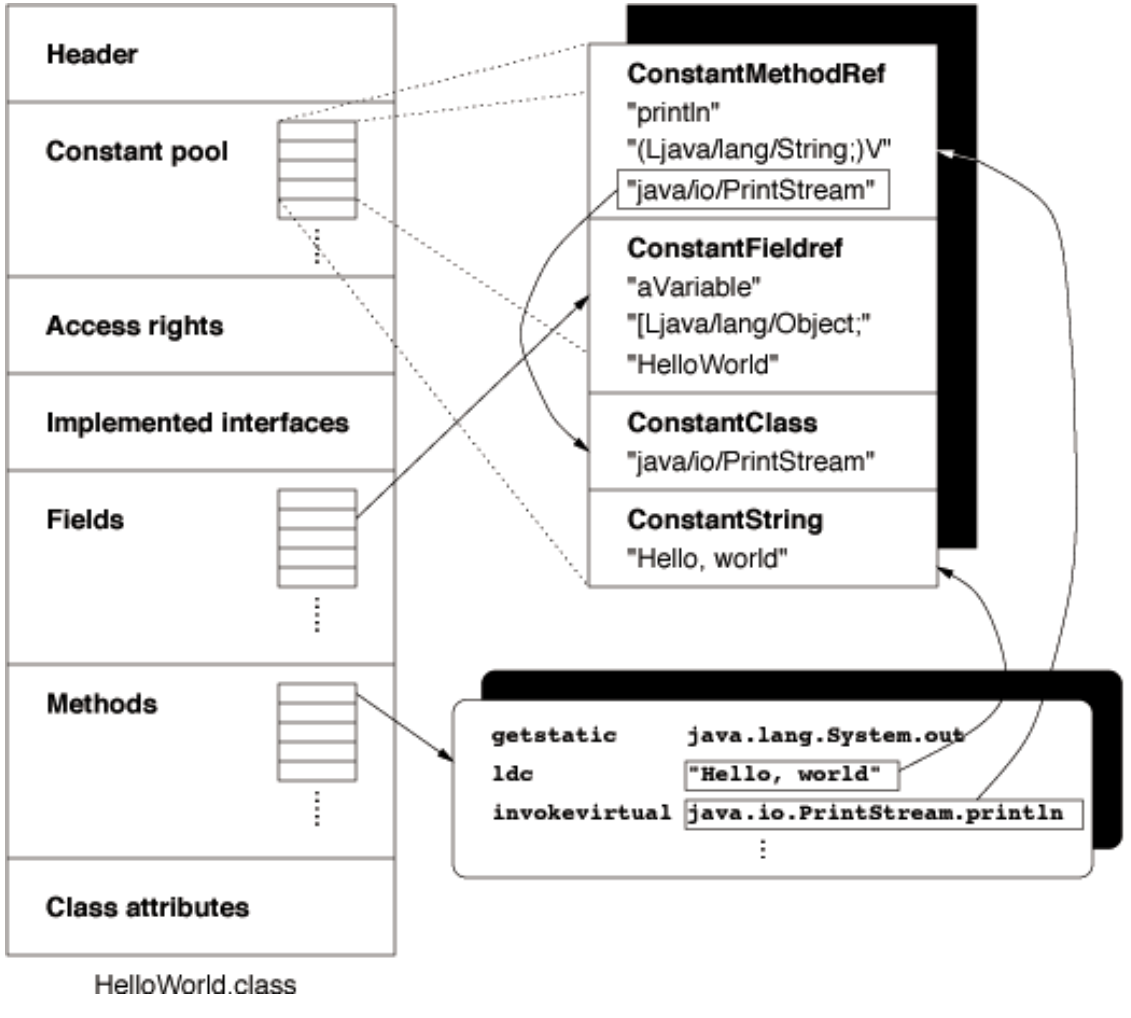
- Extension et amélioration des programmes Java
- Manipulation et transformation de bytecode
  - après la compilation Java
  - ou lors du chargement dynamique des classes
- BCEL fournit une API qui permet de mettre en œuvre ses propres transformations de bytecode

Exemple

```
System.out.println("Hello, world");
```



Exemple


Header
Constant pool
Access rights
Implemented interfaces
Fields
Methods
Class attributes
HelloWorld.class
ConstantMethodRef
"println"
"(Ljava/lang/String;)V"
"java/io/PrintStream"
ConstantFieldref
"aVariable"
"[Ljava/lang/Object;"
"HelloWorld"
ConstantClass
"java/io/PrintStream"
ConstantString
"Hello, world"
getstatic java.lang.System.out
ldc "Hello, world"
invokevirtual java.io.PrintStream.println

API de BCEL

- Trois parties principales
  - Package de description statique
  - Package de modification dynamique
  - Package d'exemples et d'utilitaires

API de BCEL : Package de description statique

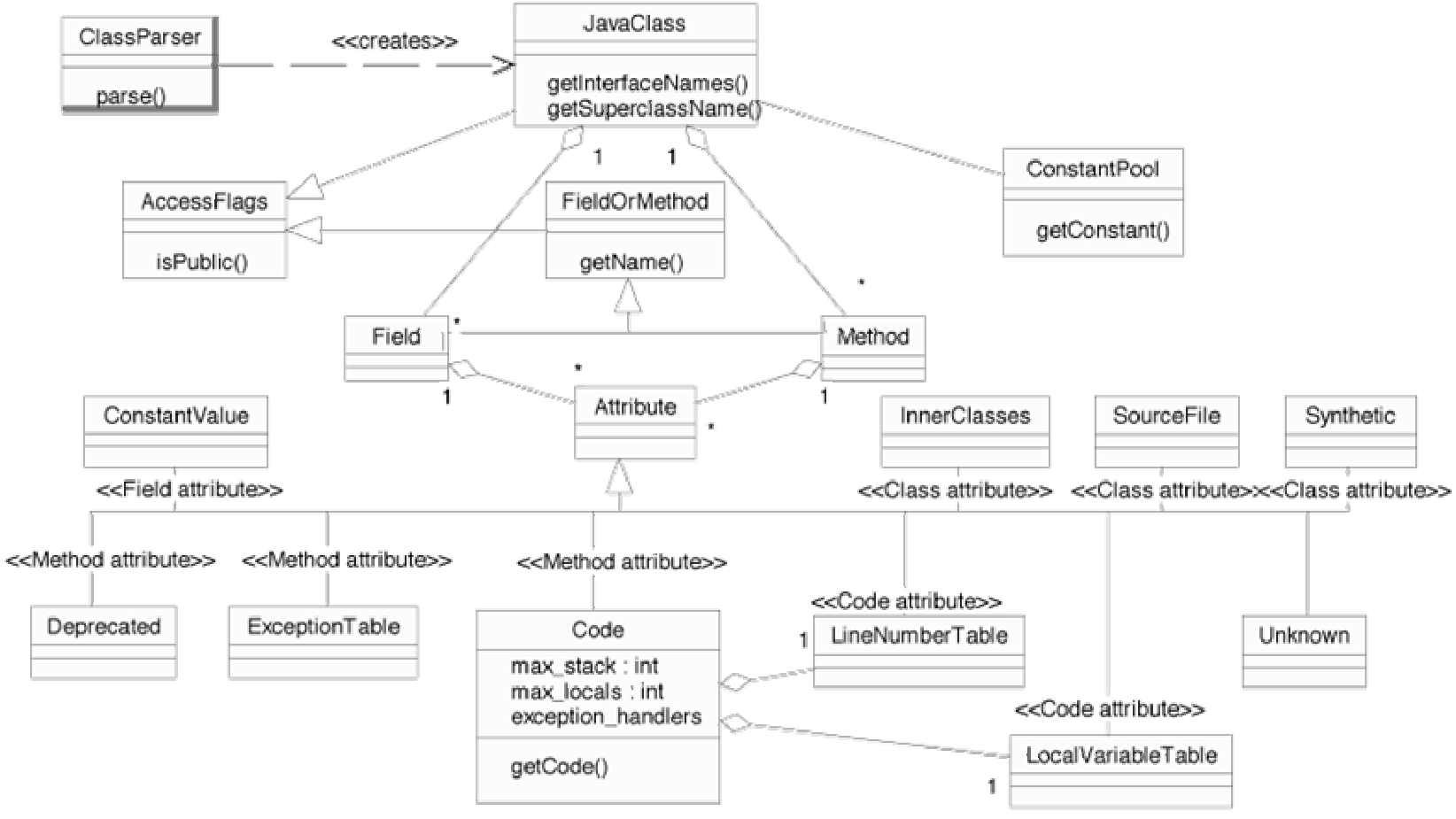
- décrit le format des fichiers de classes
- pour la lecture/écriture de fichiers de classes
- pas pour la modification du bytecode

↳ utile pour l'analyse de classes
↳ principale classe *JavaClass*

Exemples de classes

- JavaClass
  - public String getClassName()
  - public ConstantPool getConstantPool()
  - public Field[] getFields()
  - public Method[] getMethods()
- Method
  - public LocalVariableTable getLocalVariableTable()
  - public ExceptionTable getExceptionTable()

JavaClass


ClassParser
parse()
<<creates>>
JavaClass
getInterfaceNames()
getSuperclassName()
1
1
ConstantPool
getConstant()
AccessFlags
isPublic()
FieldOrMethod
getName()
Field
Method
1
1
ConstantValue
Attribute
InnerClasses
SourceFile
Synthetic
<<Field attribute>>
<<Class attribute>>
<<Class attribute>>
<<Class attribute>>
<<Method attribute>>
<<Method attribute>>
<<Method attribute>>
<<Code attribute>>
Deprecated
ExceptionTable
Code
max_stack : int
max_locals : int
exception_handlers
getCode()
1
LineNumberTable
Unknown
<<Code attribute>>
LocalVariableTable
1

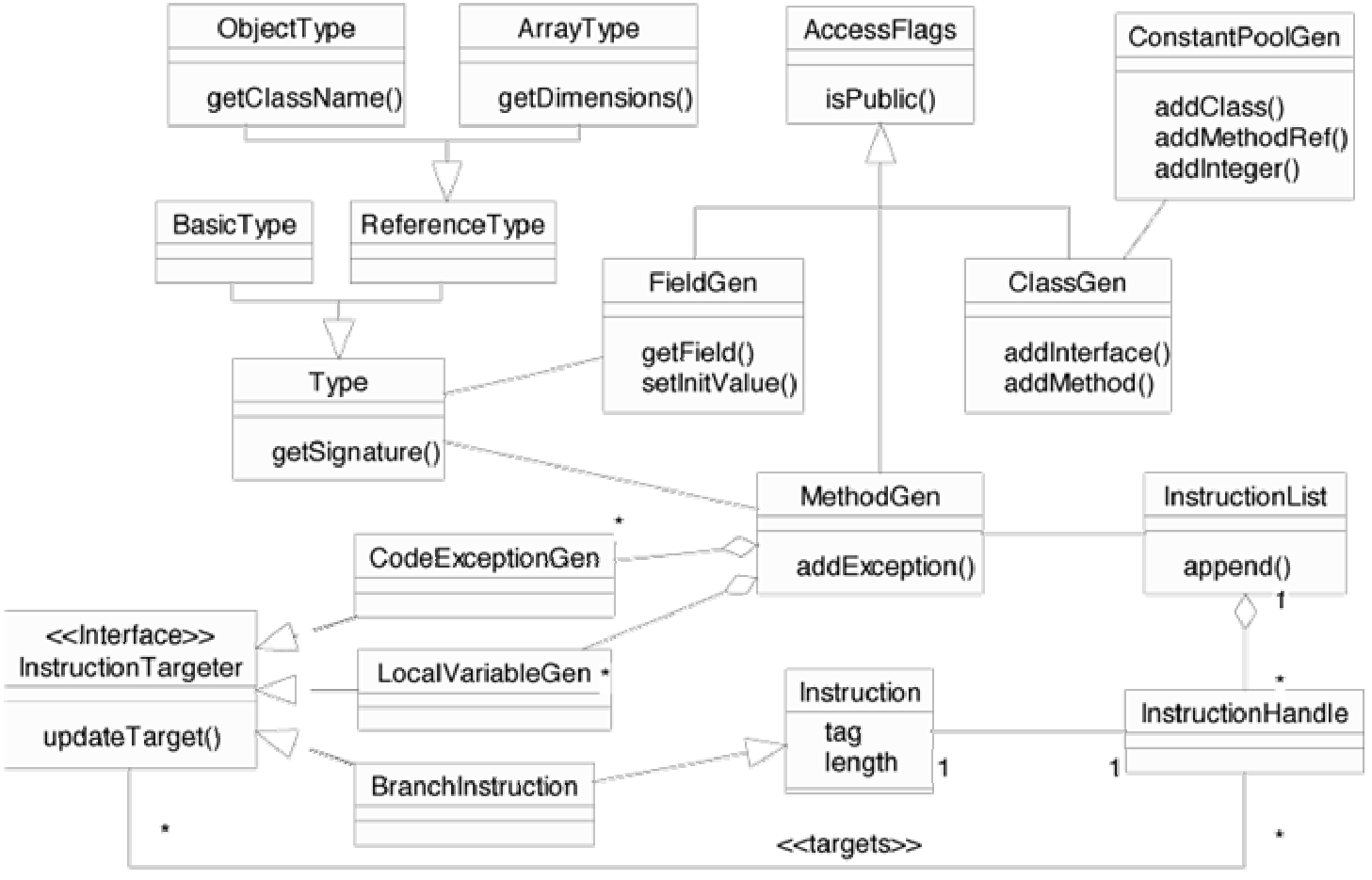
Lecture d'un fichier de classe

```
JavaClass clazz = Repository.lookupClass("java.lang.String");
System.out.println(clazz);
printCode(clazz.getMethods());
...
public static void printCode(Method[] methods) {
  for(int i=0; i < methods.length; i++) {
    System.out.println(methods[i]);
    Code code = methods[i].getCode();
    if(code != null)
      // Non-abstract method
      System.out.println(code);
    }
}
```

API de BCEL : Package de modification dynamique

- modification dynamique d'objets *JavaClass* ou *Method*
- pour l'insertion de code d'analyse
- pour la mise en œuvre de back-end de compilateur Java

Modification de classe



API de BCEL : Package d'exemples et d'utilitaires

- plusieurs exemples de code utilisant BCEL
- plusieurs utilitaires
  - *viewer* de fichier de classe
  - outil de conversion d'un fichier de classe en HTML

Synthèse

- JVM
  - une infrastructure middleware pour environnements hétérogènes
- Adaptation de bytecode
  - BCEL : une librairie de manipulation de bytecode

Synthèse : Réflexivité Java, AspectJ, BCEL

- Réflexivité Java (package *java.lang.reflect*)
  - Accès aux types des paramètres d'une méthode
    - public Method[ ] java.lang.Class.getMethods()
    - public Class[ ] java.lang.reflect.Method.getParameterTypes()
- AspectJ
  - Accès aux types et valeurs des paramètres d'une méthode
- BCEL
  - Accès aux types et valeurs des paramètres et variables locales à une méthode

Synthèse

- Comparaison de BCEL et AspectJ
  - BCEL
    - librairie de bas niveau
    - manipulation de bytecode
    - ☹ nécessite connaissance de la JVM
    - ☺ permet de construire différents outils de plus haut niveau (optimisation des programmes, analyse statique, AOP, etc.)
  - AspectJ
    - technique de plus haut niveau
    - abstractions (join points, pointcuts, advices)
    - ☺ utilisation plus aisée (ne nécessite pas la connaissance de la JVM)
    - ☹ mais moins de liberté qu'avec BCEL

Références Web

- Spécification de la machine virtuelle Java : http://java.sun.com/docs/books/vmspec/

- Manipulation de bytecode :
  - BCEL : http://jakarta.apache.org/bcel/ ou http://bcel.sourceforge.net/
  - JikesBT : http://www.alphaworks.ibm.com/tech/jikesbt/
  - Javassist : http://www.csg.is.titech.ac.jp/~chiba/javassist/
  - ASM : http://asm.objectweb.org/
  - JOIE : http://www.cs.duke.edu/ari/joie/

Références bibliographiques

- **The Java Virtual Machine Specification (2nd Edition)**, Tim Lindholm, Franck Yellin, Addison Wesley, 1999.
- **Inside the Java Virtual Machine**, Bill Venners, McGraw-Hill, 2000.
- **Programming for the Java Virtual Machine**, Joshua Engel, Addison-Wesley, 1999
- **Java Virtual Machine**, Jon Meyer, Troy Downing, O'Reilly & Associates, 1997
- **Mobilité et Persistance des Applications dans l'Environnement Java – Chapitre 2**, Sara Bouchenak, Thèse de Doctorat de l'INPG, 2001.