skyguide

ENSG Géomatique

JVM

Présenté par Yann Caron
skyguide

ENSG Géomatique

Plan du cours

Principes

Jeu d'instruction

Outils

Cas pratique

Principes

Principe

- Une machine virtuelle est un programme qui abstrait l'architecture d'une machine
- Le but étant de s'abstraire des problématiques de bas niveau
- Il en existe deux types :
  - Machine virtuelle matérielle (VMWare, Cloud computing)
  - Machine virtuelle logicielle (JVM, MSIL, Neko VM, LLVM)

JVM

- Machine virtuelle logicielle
- Créer par Sun Microsystems en 1994
- Type : A pile et à registre (Stack & Register)
- Endianness : big endian (bit de poids fort)

- Acronymes
  - JVM : Java Virtual Machine
  - JRE : Java Runtime Environment
  - JDK : Java Developpment Kit

Big endian

- Big endian (gros-boutiste)
  Octet de poids fort à gauche
  Human readable (de gauche à droite)
- Motorola 68000, SPARC, TCP/IP, JVM
- Par exemple 0x0A800410

	0	1	2	3	
...	0A	80	04	10	...

Little endian & Bi-endian

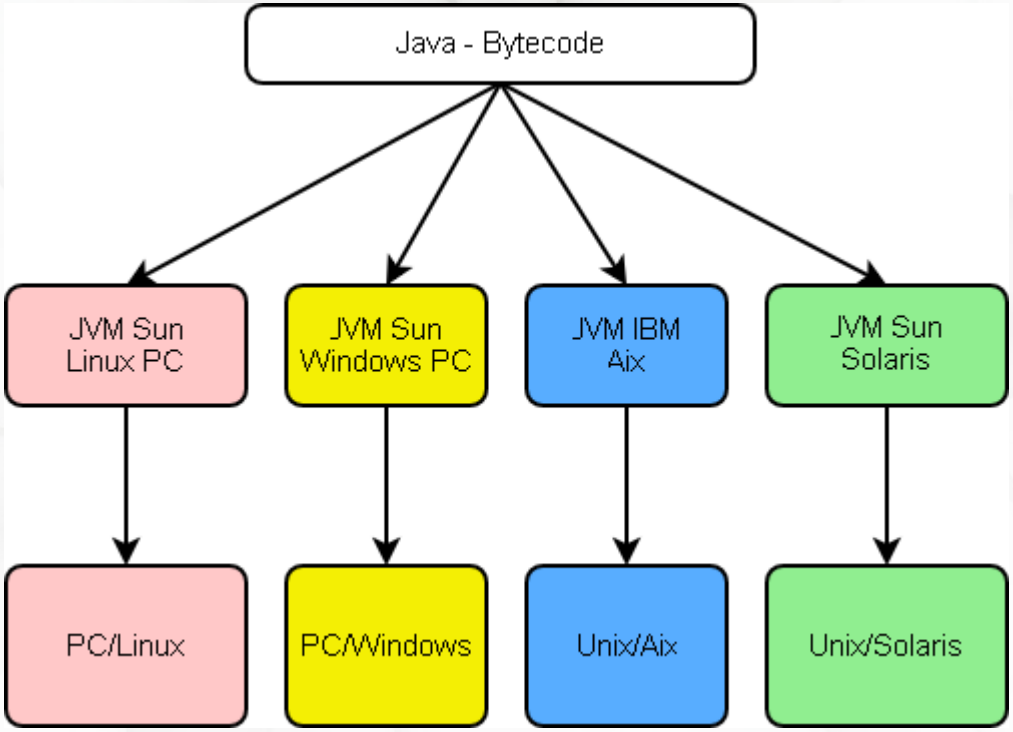
- Little endian (petit-boutiste)
  Octet de poids faible à gauche
  Machine readable (de droite à gauche)
- Processeur x86
- 0x0A800410 :

	0	1	2	3	
...	10	04	80	0A	...

- Bi-endian
- Choix de l'endianess au niveau logiciel ou materiel
- Power PC, ARM, MIPS

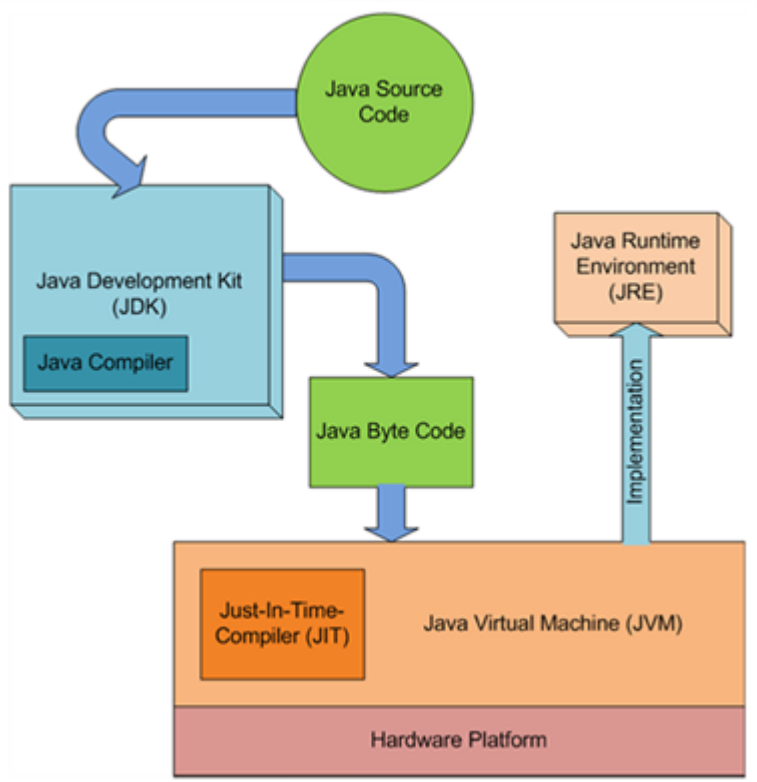
JVM

- Code once, run anywhere



JVM - Compilation

- Les classes Java sont compilés en bytecode pour être exécuter par la JVM

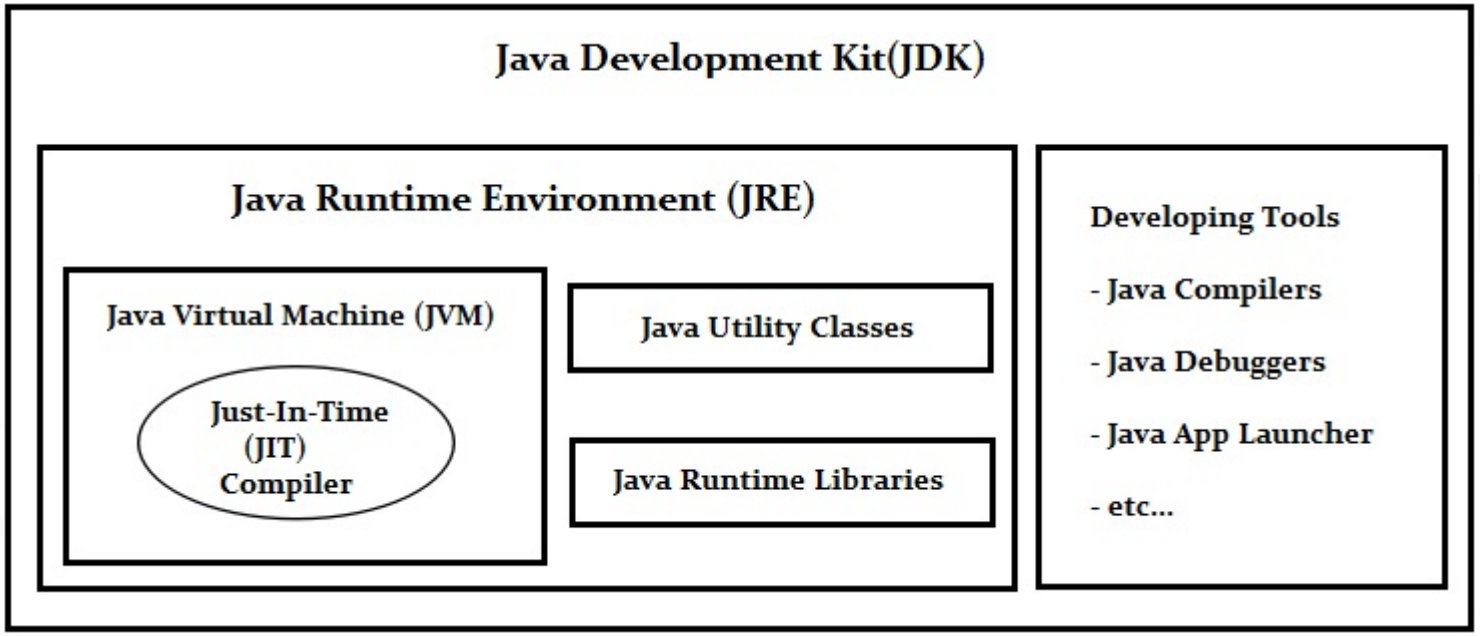


JVM - Compilation

- Le code Java est transformé en bytecode (via javac du JDK)
- Le bytecode est un langage “machine readable” au jeu d’instruction restreint (1 byte soit 8 bits)
- Le bytecode est destiné à être :
  - Interpreté par la JVM
  - Compilé par JIT (Just in Time compiler)

JDK – Java Development Kit

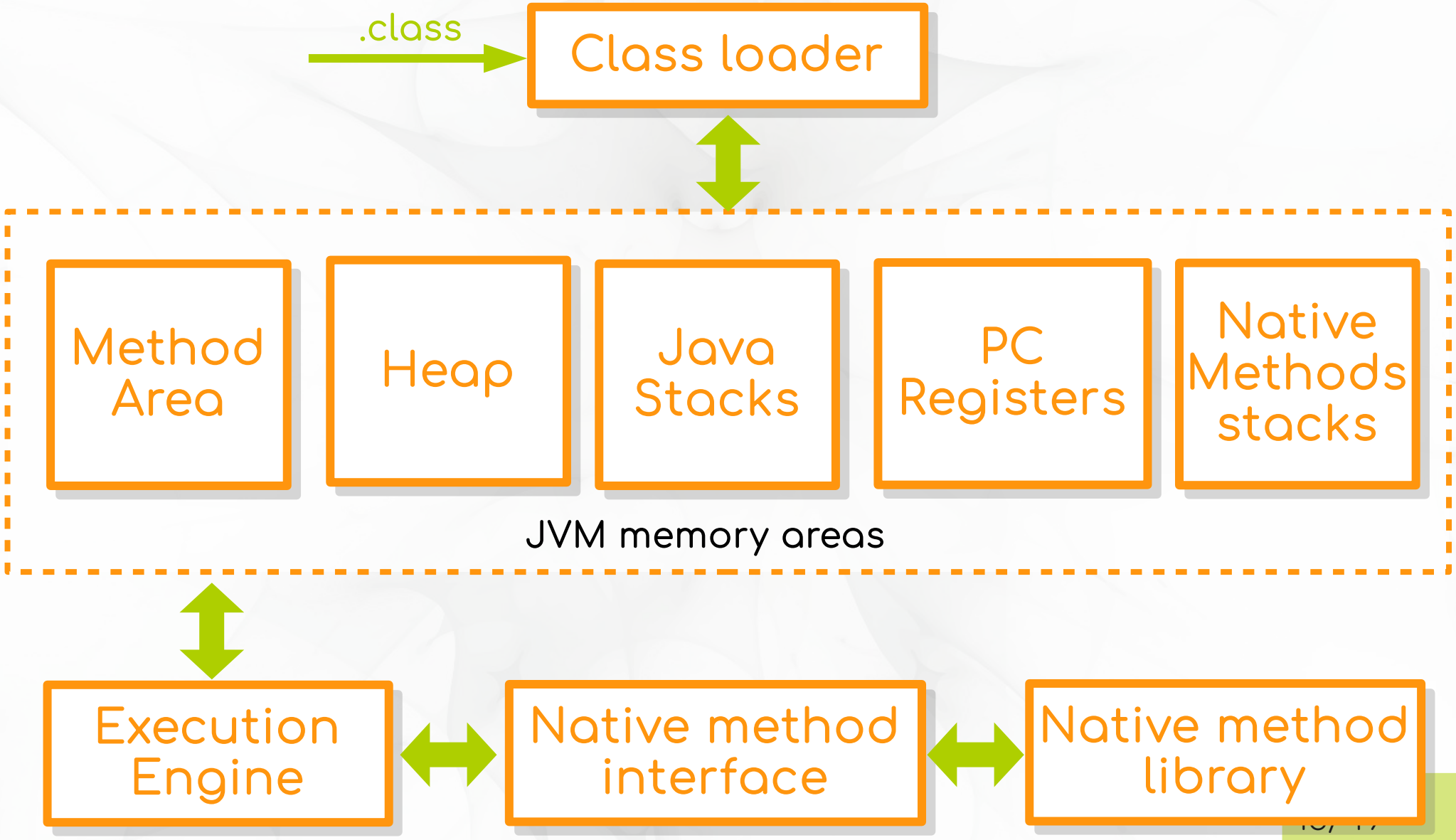
- Contient les outils nécessaires à la création de programmes Java



JDK – Java Development Kit

- JRE : Contient la JVM ainsi que toutes les bibliothèques pré-compilés (Runtime library et Java utilities) nécessaires à l'exécution d'un programme Java
- Ces bibliothèques sont contenues dans le rt.jar
- JVM : La machine virtuelle qui est en charge d'exécuter le programme (interpréter ou compiler à la volée)

Architecture de la JVM



Class loader

- Charge la classe en mémoire
- Effectue l'assemblage (linkage)
  - Effectue les vérifications de dépendances
  - Charge les classes nécessaires
- Initialise la classe
  - Alloue la mémoire nécessaire

JVM memory area

- Method area : contient les informations statiques (liés à la classe)
  - Variables statiques
- Head area : contient les informations d'instances (liés à l'instance des objets)
  - Variables d'instances
- Stack area : ensemble des piles d'exécutions (machine à pile).

Stack area

- Stack area :

  Ensemble des piles d'exécutions (machine à pile).
- Une pile par thread
- Subdivisé en 3 parties :
  - Variables locales
  - Opérandes
  - Frame data (valeurs de retour, pointeur de méthodes etc...)

JVM memory area (suite)

- PC Register : Stocke les informations relatives au Threads, les adresses de retours
- Native Method Stack : Pile d'exécution pour les threads natifs

Jeu d'instructions

Bytecode

- Comme en langage machine, il existe une mnémonique pour chaque instruction

```
// Bytecode: 03 3b 84 00 01 1a 05 68 3b a7 ff f9
// Disassembly:
iconst_0     // 03
istore_0     // 3b
iinc 0, 1    // 84 00 01
iload_0      // 1a
iconst_2     // 05
imul         // 68
istore_0     // 3b
goto -7      // a7 ff f9
```

Bytecode

- Jeu d'instruction compact (1 byte – 8 bits)

- Comme la JVM est stack based, toute opération doit passer par la pile

Type primitifs

Type	Definition
byte	1 byte entier signé (complément à 2)
short	2 byte entier signé (complément à 2)
short	4 byte entier signé (complément à 2)
long	8 byte entier signé (complément à 2)
float	4 byte IEE 754 simple précision à virgule flottante
double	8 byte IEE 754 double-précision à virgule flottante
char	2 byte non signé caractère unicode

Complément à 2

- Notation spécifique des nombres négatifs simplifiant les opérations binaires

- Ecriture binaire des nombres négatifs
  - 1er idée : utiliser le bit de poids fort comme signe

```
00000101    //  5
10000101    // -5
```

Complément à 2

- Problème : 0 a deux représentations :

```
00000000    //  0
10000000    // -0
```

- Problème : l'addition de nombres négatifs ne fonctionne pas

```
 00000101      //      5
+10000101      // + - 5
=10001010      // = -10
```

Complément à 2

- Solution et 2eme idée : Le complément à deux :
- Inverser les bits du nombre négatif
- Lui ajouter 1

```
00000101    //  5
11111011    // -5
```

```
 00000101      //      5
+11111011      // + - 5
=00000000      // =   0
```

```
 00000101      //      5
+11111101      // + - 3
=00000010      // =   2
```

Outils

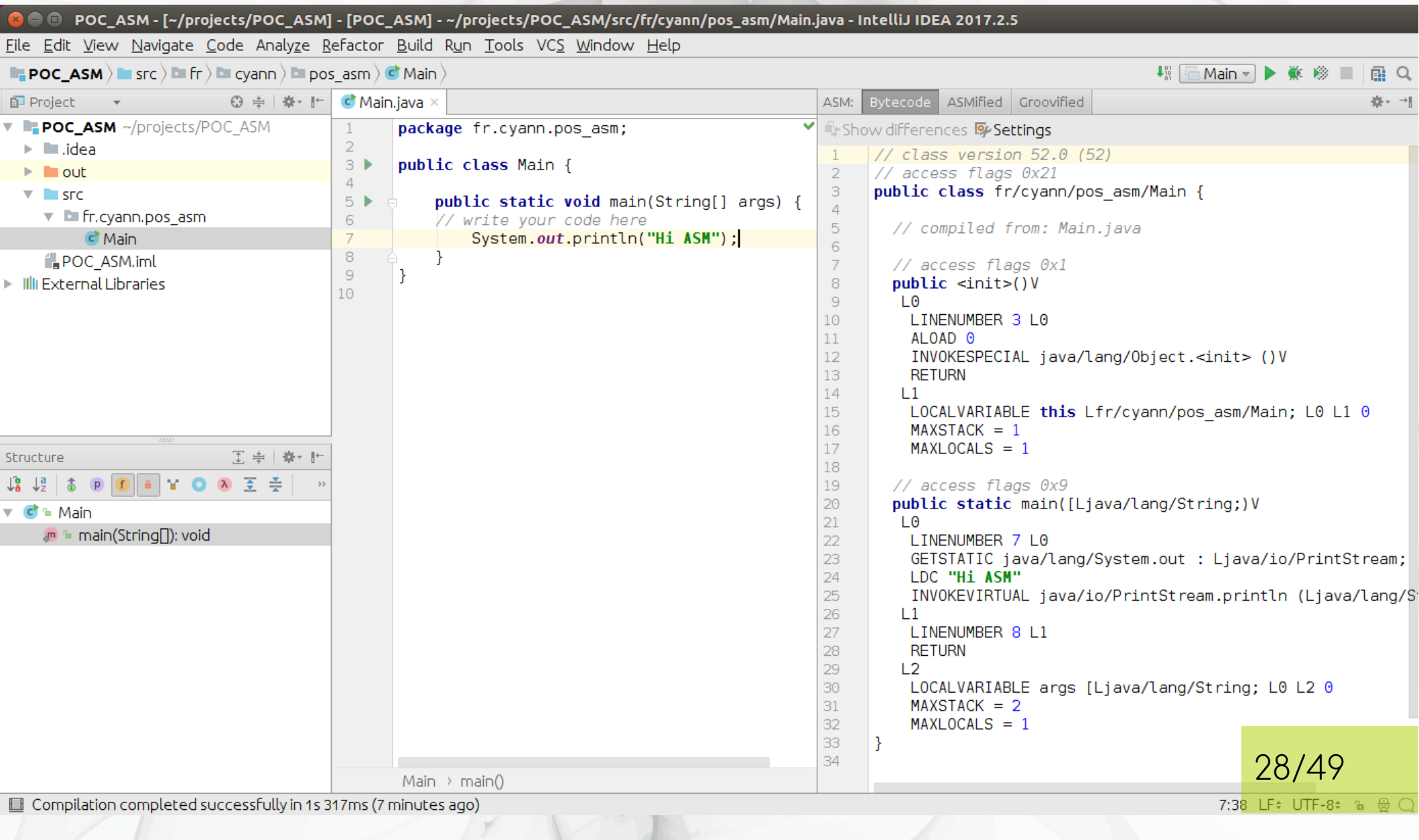
Java

- javac [file]
  - compilateur java
- java [class]
  - exécuteur de classes java
- javap -v [class]
  - dé-compilateur de classes java

Bibliothèques ByteCode

- Apache BCEL
- ASM
- Byte Buddy (basé sur ASM)
- Javaassist
  - Utilisé pour de l'AOP (insertion de code avant, pendant et après l'appel de méthodes déjà définies)
- JBL
- Jasmin (abandonné depuis 2010)

Plugin ASM


POC_ASM - [~/projects/POC_ASM] - [POC_ASM] - ~/projects/POC_ASM/src/fr/cyann/pos_asm/Main.java - IntelliJ IDEA 2017.2.5
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
POC_ASM src fr cyann pos_asm Main
Main
Project
POC_ASM ~/projects/POC_ASM
.idea
out
src
fr.cyann.pos_asm
Main
POC_ASM.iml
External Libraries
Structure
Main
main(String[]): void
Main.java
1 package fr.cyann.pos_asm;
2
3 public class Main {
4
5 public static void main(String[] args) {
6 // write your code here
7 System.out.println("Hi ASM");
8 }
9 }
10
Main › main()
ASM:
Bytecode
ASMified
Groovified
Show differences
Settings
1 // class version 52.0 (52)
2 // access flags 0x21
3 public class fr/cyann/pos_asm/Main {
4
5 // compiled from: Main.java
6
7 // access flags 0x1
8 public <init>()V
9 L0
10 LINENUMBER 3 L0
11 ALOAD 0
12 INVOKESPECIAL java/lang/Object.<init> ()V
13 RETURN
14 L1
15 LOCALVARIABLE this Lfr/cyann/pos_asm/Main; L0 L1 0
16 MAXSTACK = 1
17 MAXLOCALS = 1
18
19 // access flags 0x9
20 public static main([Ljava/lang/String;)V
21 L0
22 LINENUMBER 7 L0
23 GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
24 LDC "Hi ASM"
25 INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/S
26 L1
27 LINENUMBER 8 L1
28 RETURN
29 L2
30 LOCALVARIABLE args [Ljava/lang/String; L0 L2 0
31 MAXSTACK = 2
32 MAXLOCALS = 1
33 }
34
Compilation completed successfully in 1s 317ms (7 minutes ago)
7:38 LF UTF-8

Plugin ASM

- Disponible dans les IDE :
  - Intellij
  - Eclipse

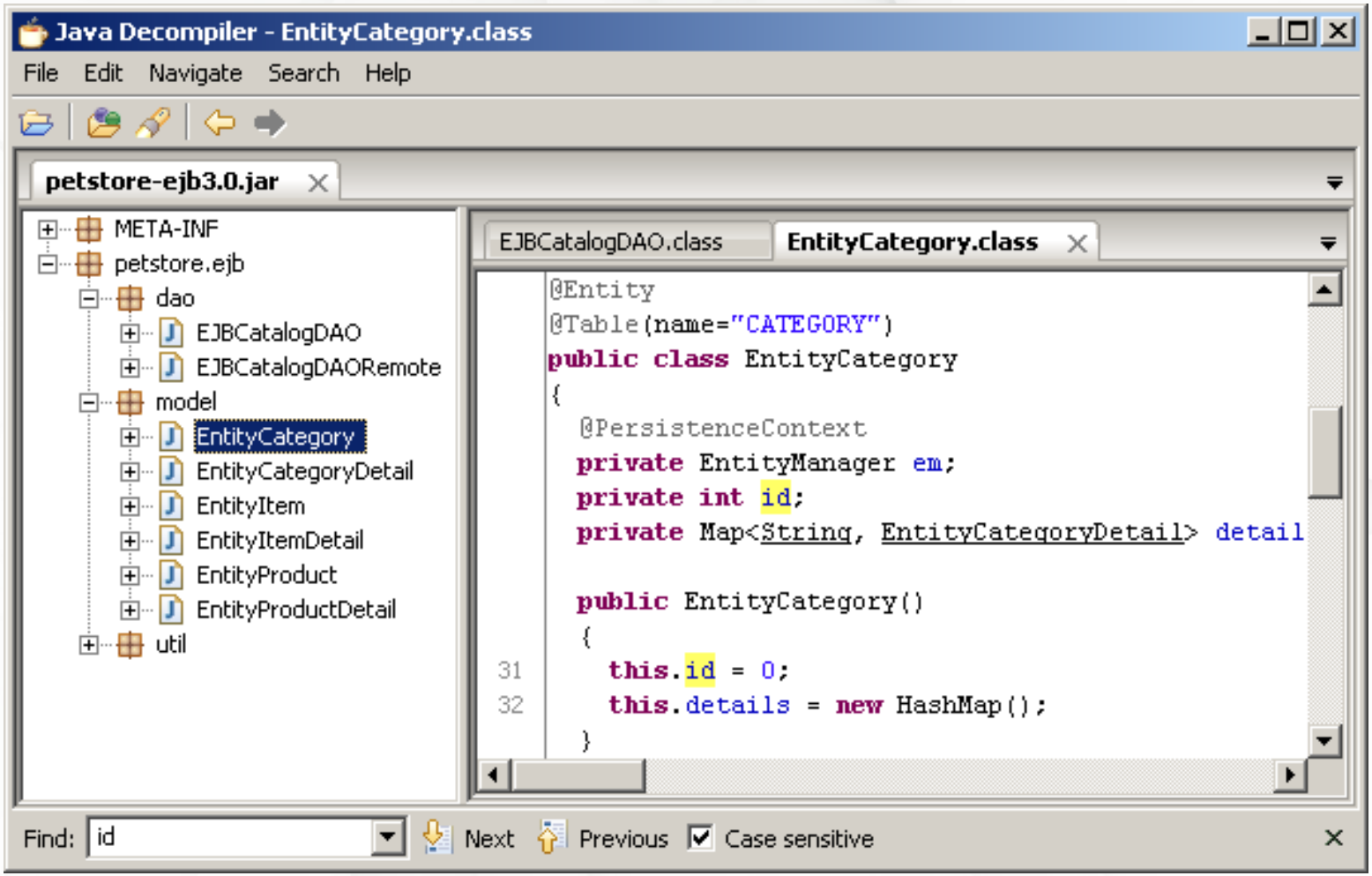
- Donne une lecture en OpCode ou directement en code Java pour la bibliothèque ASM

Java Decompiler

- Outil autonome JD-GUI
- Ou plugins Eclipse / Intelij

- Permet de lire du code source Java à partir de .class (bytecode compilés)

Java Decompiler

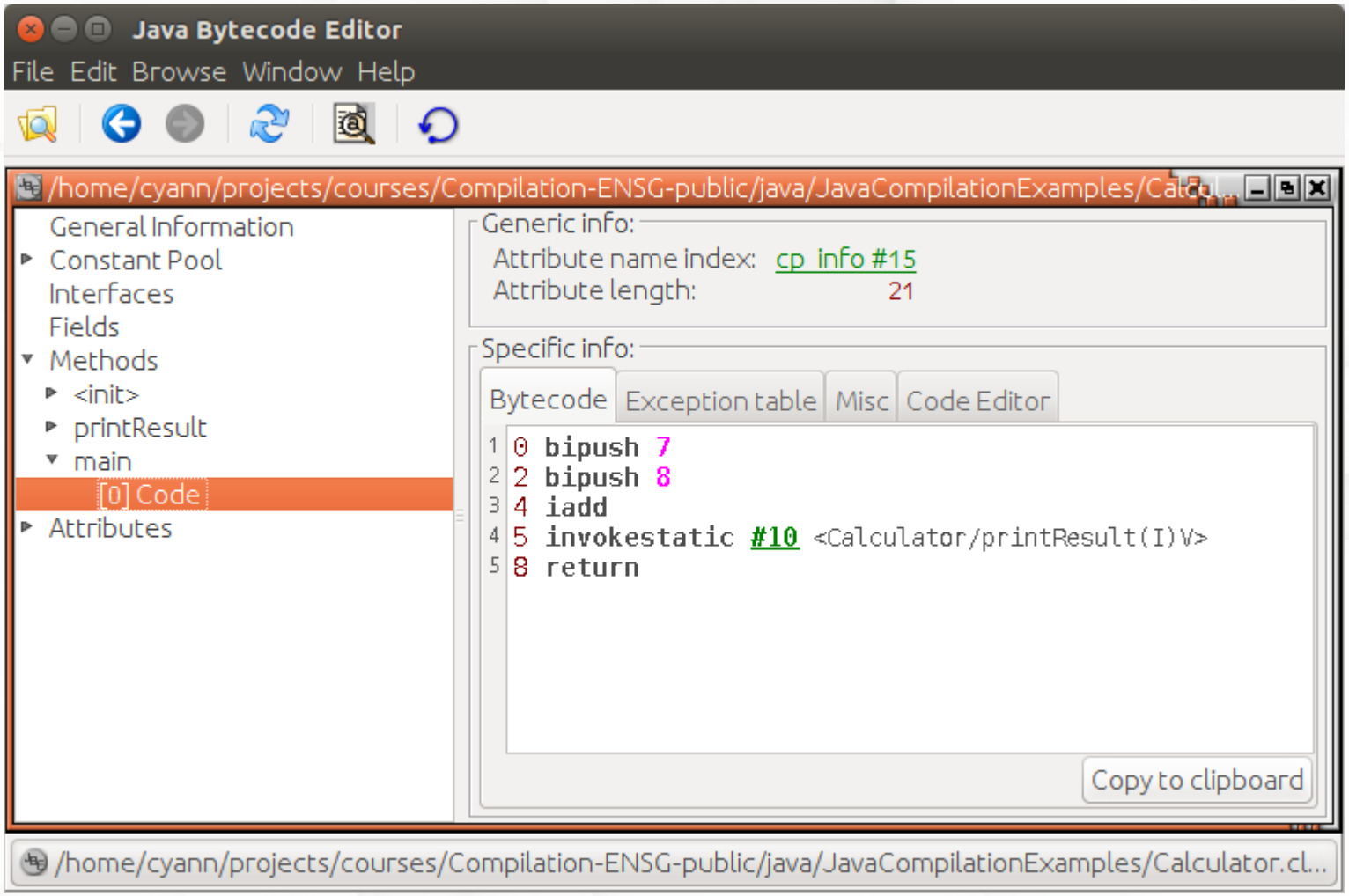


JBE

- Java bytecode editor
- Outil basé sur BCEL & jclasslib
- Attention, bug sur linux, il faut modifier le fichier jbe.sh

```
#!/bin/bash
cd bin
java ee.ioc.cs.jbe.browser.BrowserApplication
```

JBE


Java Bytecode Editor
File Edit Browse Window Help
/home/cyann/projects/courses/Compilation-ENSG-public/java/JavaCompilationExamples/Calc
General Information
Constant Pool
Interfaces
Fields
Methods
<init>
printResult
main
[0] Code
Attributes
Generic info:
Attribute name index: cp_info #15
Attribute length: 21
Specific info:
Bytecode
Exception table
Misc
Code Editor
1 0 bipush 7
2 2 bipush 8
3 4 iadd
4 5 invokestatic #10 <Calculator/printResult(I)V>
5 8 return
Copy to clipboard
/home/cyann/projects/courses/Compilation-ENSG-public/java/JavaCompilationExamples/Calculator.cl...

Cas pratique

Cas pratique

- Analysons le bytecode du code suivant

```
public class Calculator {

   public static void main(String[] args) {
      int a = 7;
      int b = 8;
      int c = a + b;

      System.out.println(c);
   }

}
```

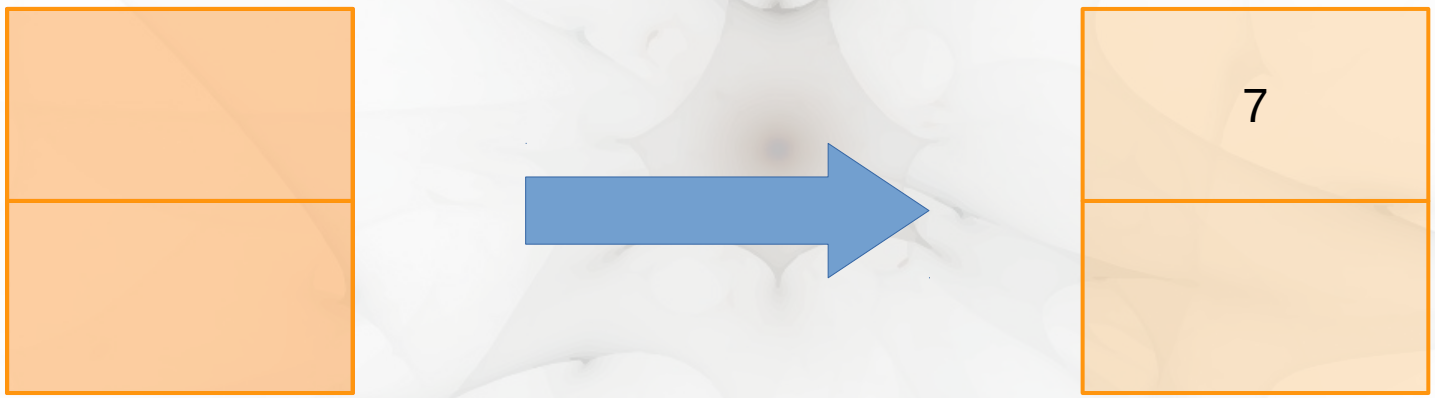
Bytecode

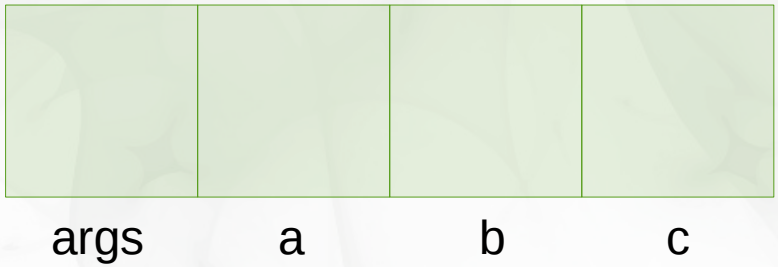
- Ouvrons le avec JBE

```
 0 bipush 7
 2 istore_1
 3 bipush 8
 5 istore_2
 6 iload_1
 7 iload_2
 8 iadd
 9 istore_3
10 getstatic #2 <java/lang/System/out Ljava/io/PrintStream;>
13 iload_3
14 invokevirtual #3 <java/io/PrintStream/println(I)V>
17 return
```

Bytecode

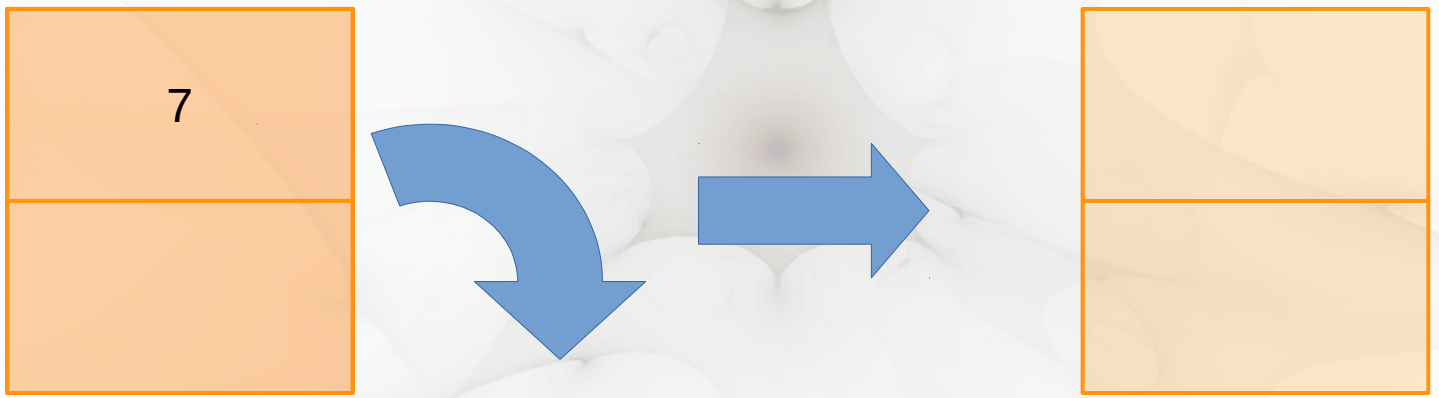
- bipush 7

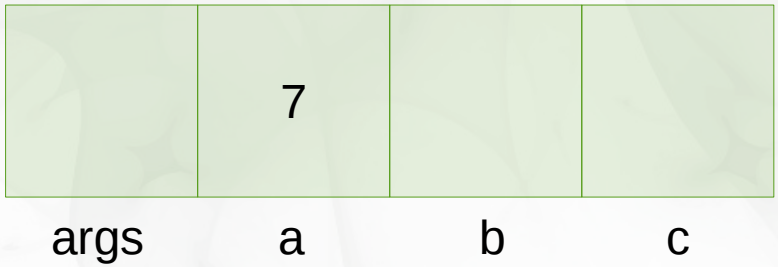




Bytecode

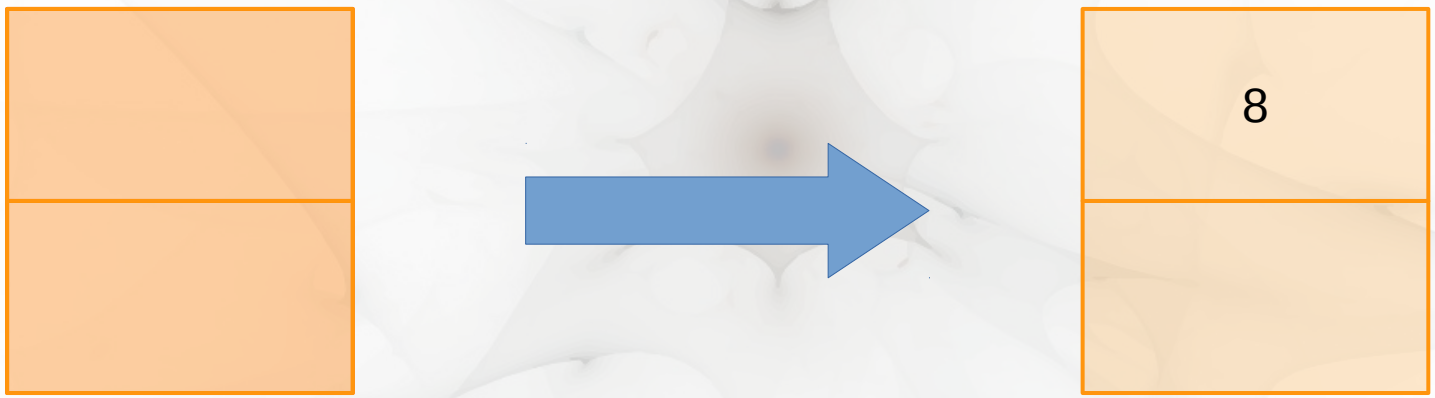
- istore_1

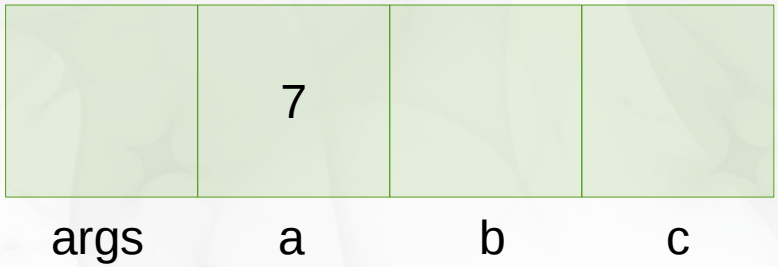




Bytecode

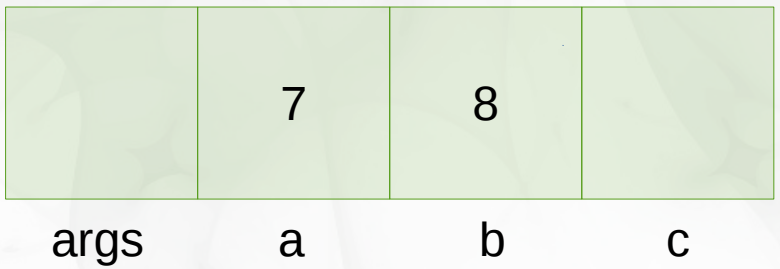
- bipush 8





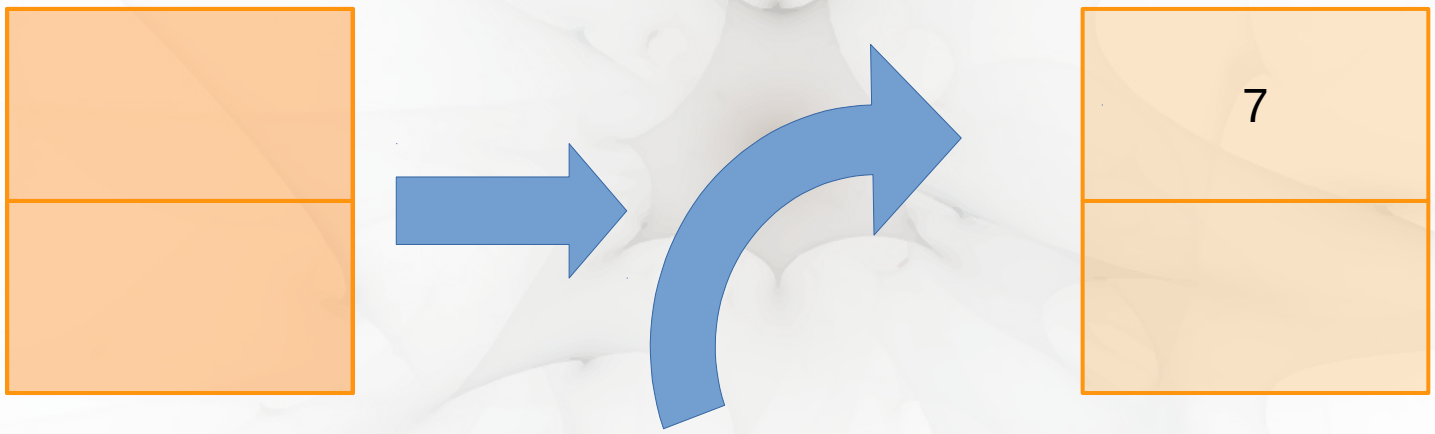
Bytecode

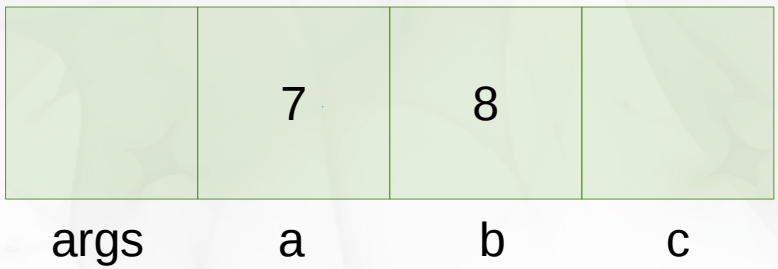
- istore_2



Bytecode

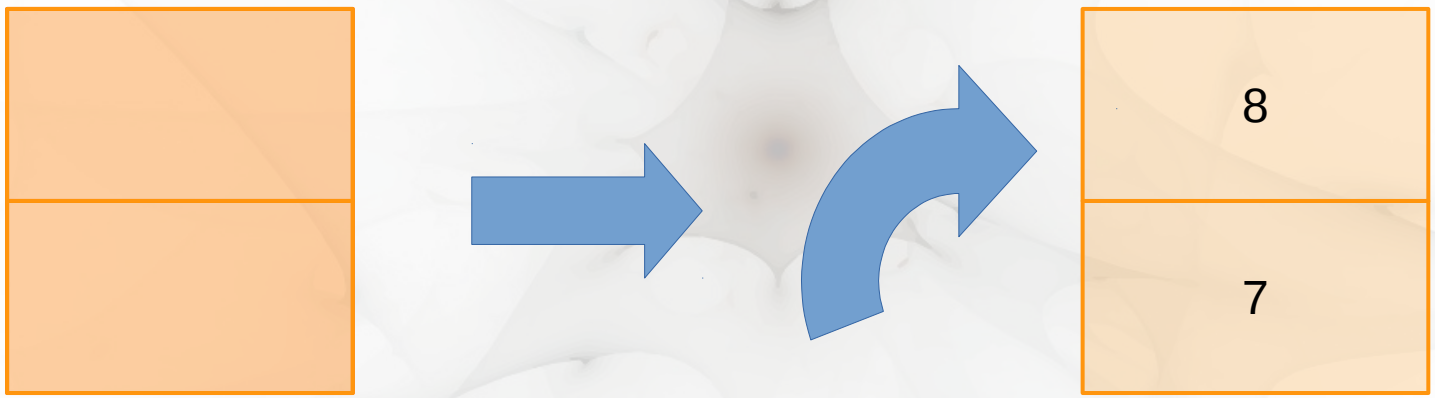
- iload_1

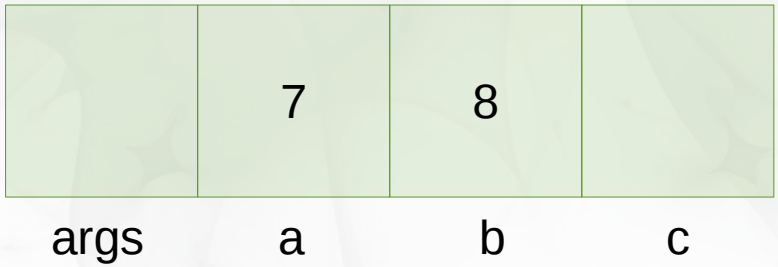




Bytecode

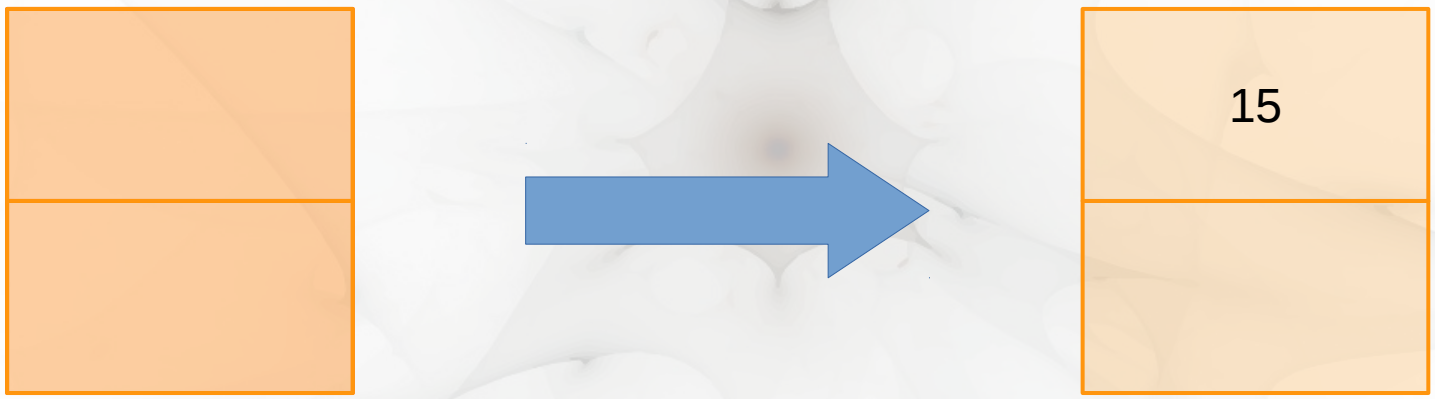
- iload_2

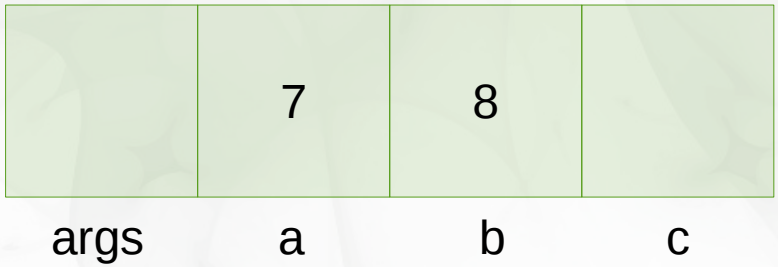




Bytecode

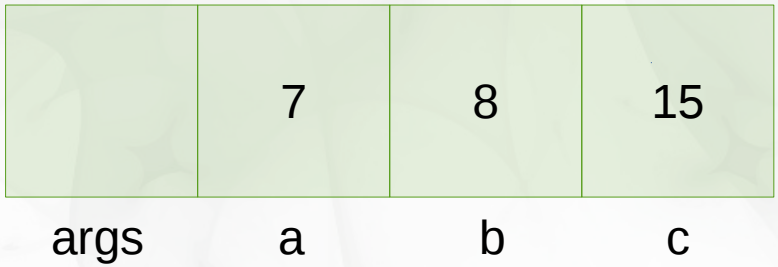
- iadd





Bytecode

- istore_3



Exercice 1

- Créer un fichier Calculator.java contenant

```
public class Calculator {

   public static void main(String[] args) {
      int a = 7;
      int b = 8;
      int c = a + b;

      System.out.println(c);
   }

}
```

Exercice 1

- le compiler avec **javac**
- l'exécuter avec **java**
- le décompiler avec **javap -v**
- le décompiler avec JBE

- modifiez l'implémentation de la méthode pour ne plus stocker les résultats intermédiaires dans des variables locales.

Exercice 2

- sur le même principe que l'exercice 1 observez les différentes déclarations :
  - if / switch
  - for / break
  - while
  - do while
  - lambda

Exercice 3

- Avec l'aide de jbe et de javap
- Avec une bibliothèque comme BCEL ou ASM
- A partir de l'AST obtenu avec le parser, générer le bytecode et l'exécuter

Merci de votre attention