Modèle d'Exécution de Java
JVM / Bytecode

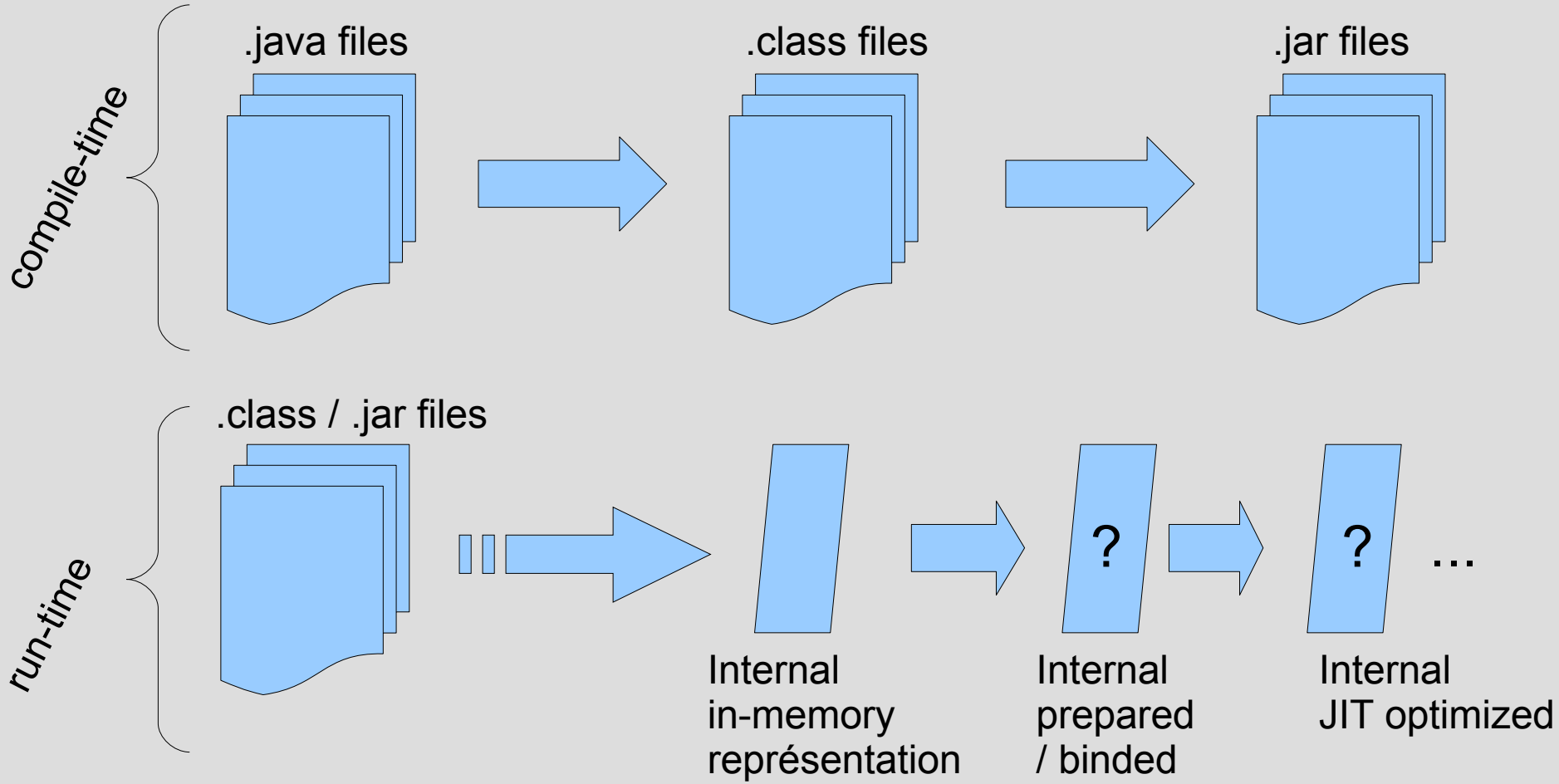
Arnaud Nauwynck

arnaud.nauwynck@gmail.com

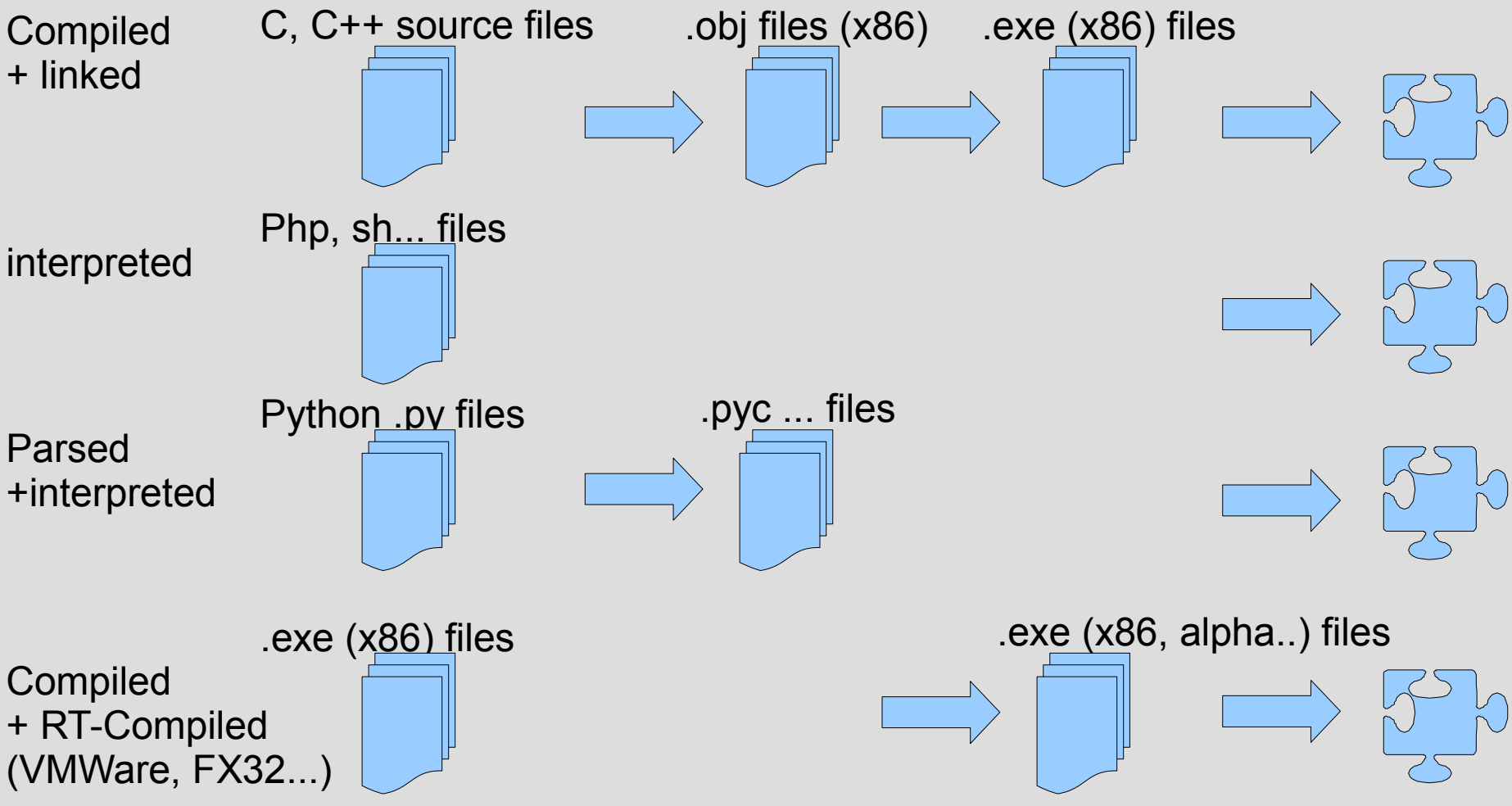
Plan

- Introduction : Compilation, Runtime, VM
- Le Bytecode de Java
- « Hello World » .class file détaillé
- « Hello World » exécution pas-à-pas
- Hotspot optimizer, Myth&Reality

Introduction : Compilation, Runtime, JVM

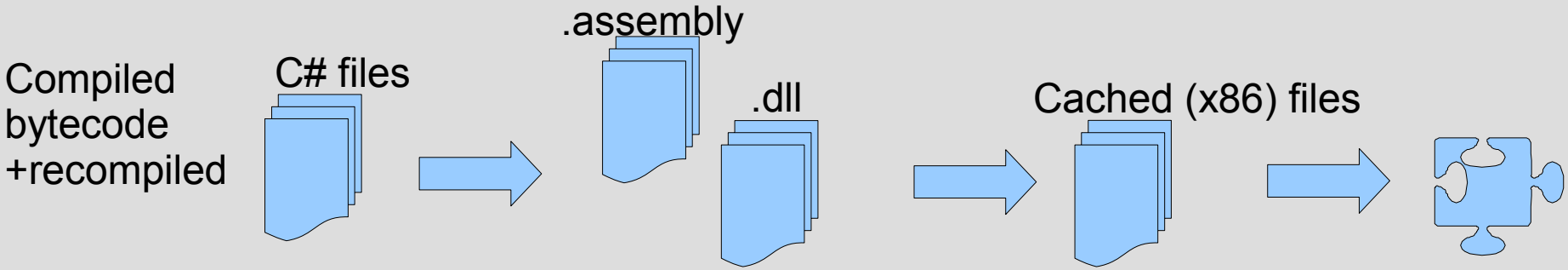


Comparison de Modèles d'Exécutions



Modèle d'Exécution de DotNet

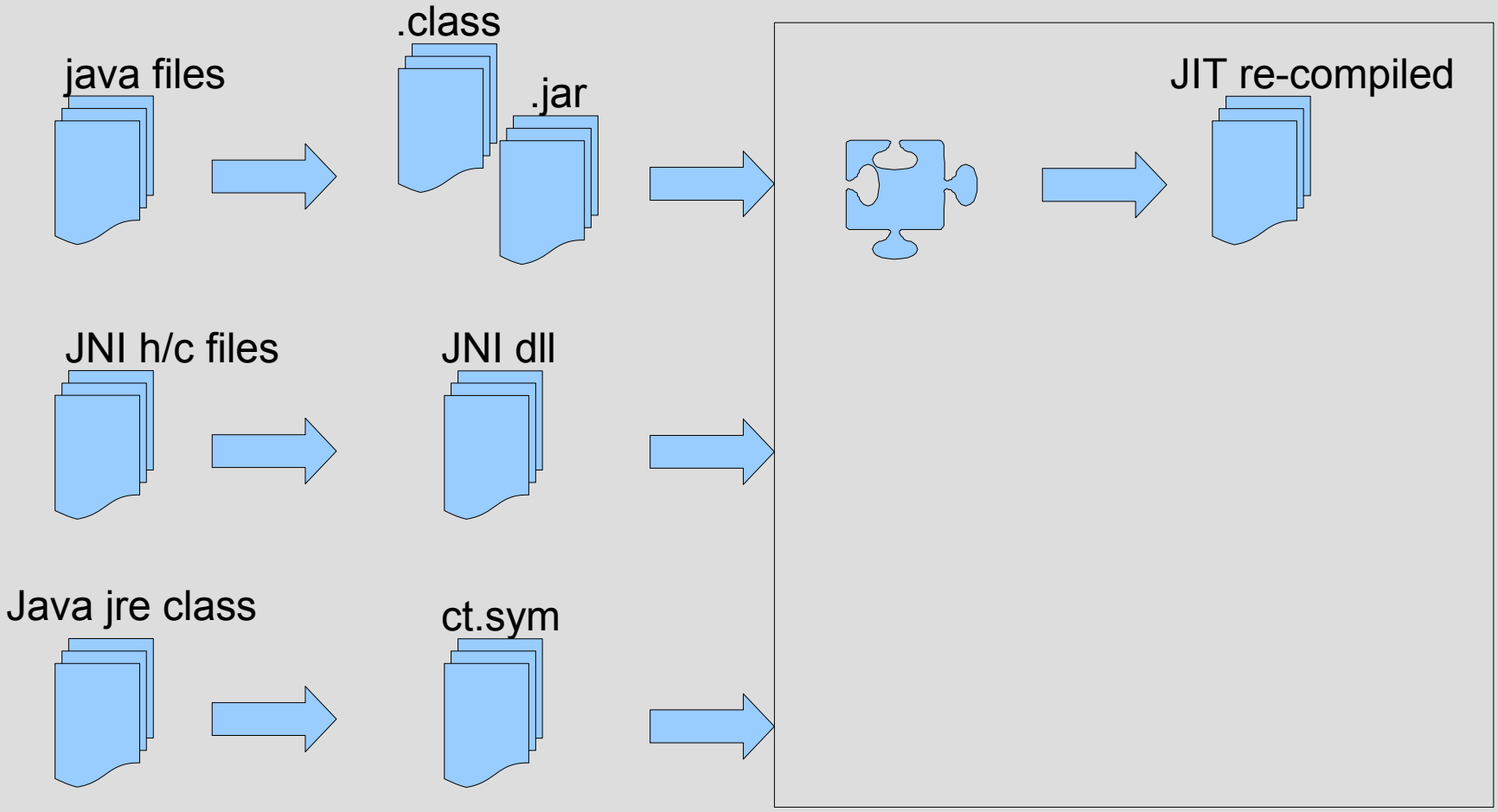
- Compatibilité binaire au runtime... idem DLLs + typage objet des APIs
  - (... le grall Microsoft: OLE, OLE2, ActiveX, COM, DCOM, Com+...)
- Pas de JIT ... compilation systématique à la 1ere exécution du bytecode + cache disk



Modèle d'Exécution de DotNet

- Très bon compromis !
- Plus simple et plus efficace(?) qu'en java
- Problématique pour les applets ?

Modèle d'Exécution de Java - JVM Sun 1.6



Modèle d'Exécution de Java

- Très souple et finallement très bon
- Polémique, critique ... souvent à tord
- Implémentation de Sun
  - Hotspot en continuelle évolution
  - Class sharing, Meilleur ClassLoading java1.6
- Autres VM. Ex: Jrockit, ...

Plan

- Introduction : Compilation, Runtime, VM
- Le Bytecode de Java
- « Hello World » .class file détaillé
- « Hello World » exécution pas-à-pas
- Hotspot optimizer, Myth&Reality

Bytecode(VM) vs Assembleur

- Le Bytecode est simple
  - (~220 instructions < 256= 2^8 = byte)
- Strictement typé
  - => sécurisé
- Orienté-Objet
  - => très compact
- Orienté pile (et non pas registres)
  - => indépendant archi / facilement optimizable

Bytecode(VM) vs Assembleur

- Le bytecode n'a presque que des avantages
  - Compil Once, Run Anywhere
- Initialement conçut pour
  - l'électronique embarqué (car compact)
  - les applets (car sécurisé)
- Finalement, on le retrouve aussi (surtout) côté serveur

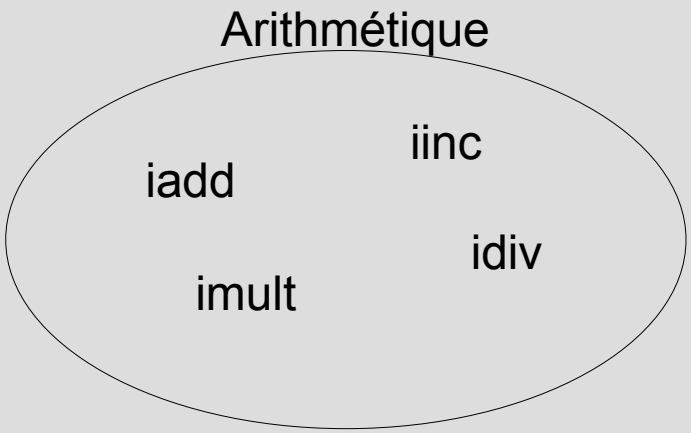
Faiblesse (des Optimizeurs) de Bytecode

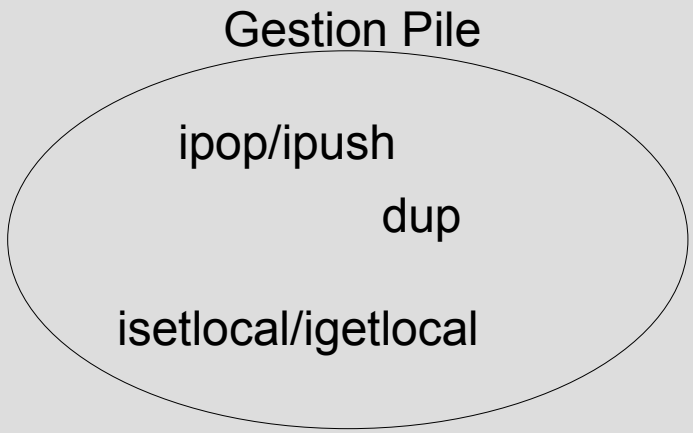
- ne contient pas d'instructions complexes optimizées (CISC) : MMX, DSP...
- Ne contient pas d'instructions « unsafe »
  - Une JVM doit implémenter TOUS les contrôles NullPointerException / ArrayIndexOutOfBound / ClassCast...
- Les calculs numériques sont stricts et unitaires
  - Pas d'arrondi comme dans « a := a*b+c »...

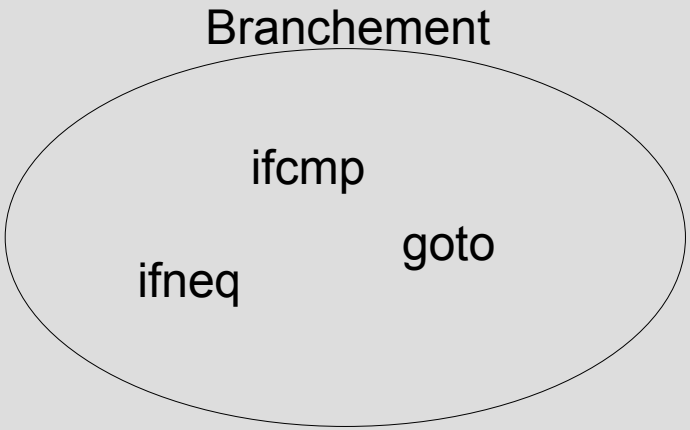
Typage/Nommage pour les OpCodes

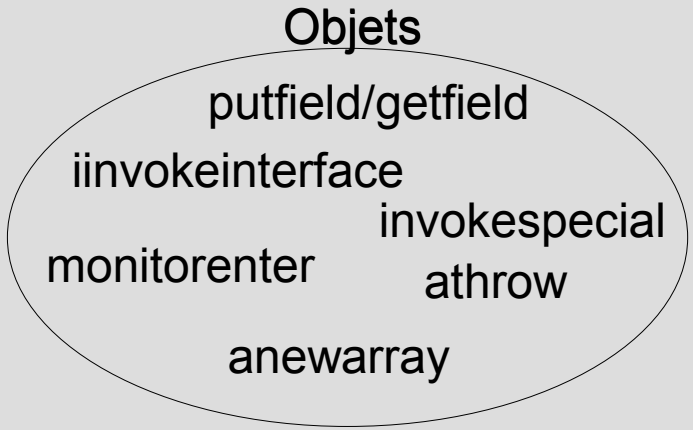
- Toutes les instructions sont typées (int / long / short / byte / float / double / ...)
  - toute opération (même « +1 »...) existe en N variantes
  - Ex: iadd / ipop / iinvokeinterface / ...
- Le nom de l'op contient le préfix du type
  - I = int / L = long / F = Float / D = Double
  - B = Byte / Z = Boolean (!!)
  - A = reference

OpCodes par Groupes









Plan

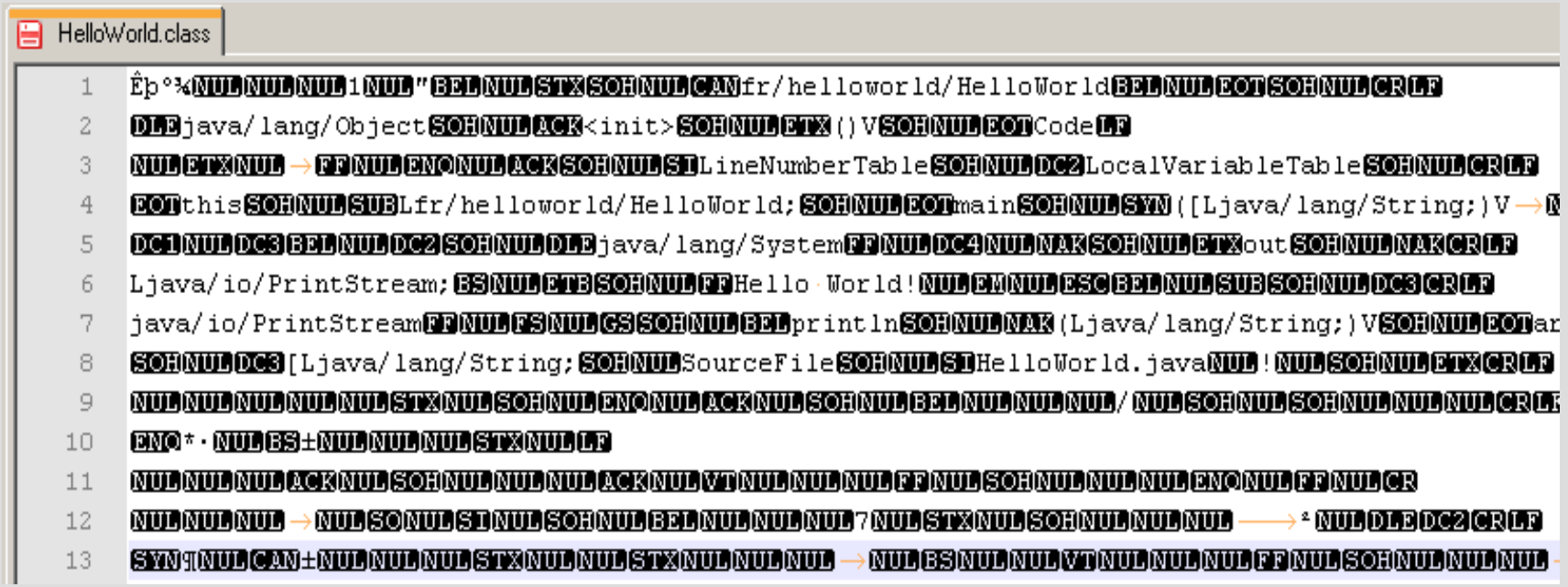
- Introduction : Compilation, Runtime, JVM
- Le Bytecode de Java
- « Hello World » .class file détaillé
  - .class structure, javap -c
- « Hello World » exécution pas-à-pas
- Hotspot optimizer, Myth&Reality

.Class Files Structure

- Spécification du format binaire
  - Class = Header + ConstantPool
    + List Field + List Method + List contructor
    + List Attribute
- Non humainement lisible
- ... mais simple à parser et à manipuler
  - nombreuses librairies disponibles (bcel, ...)
- Extensible via les « Attributs »

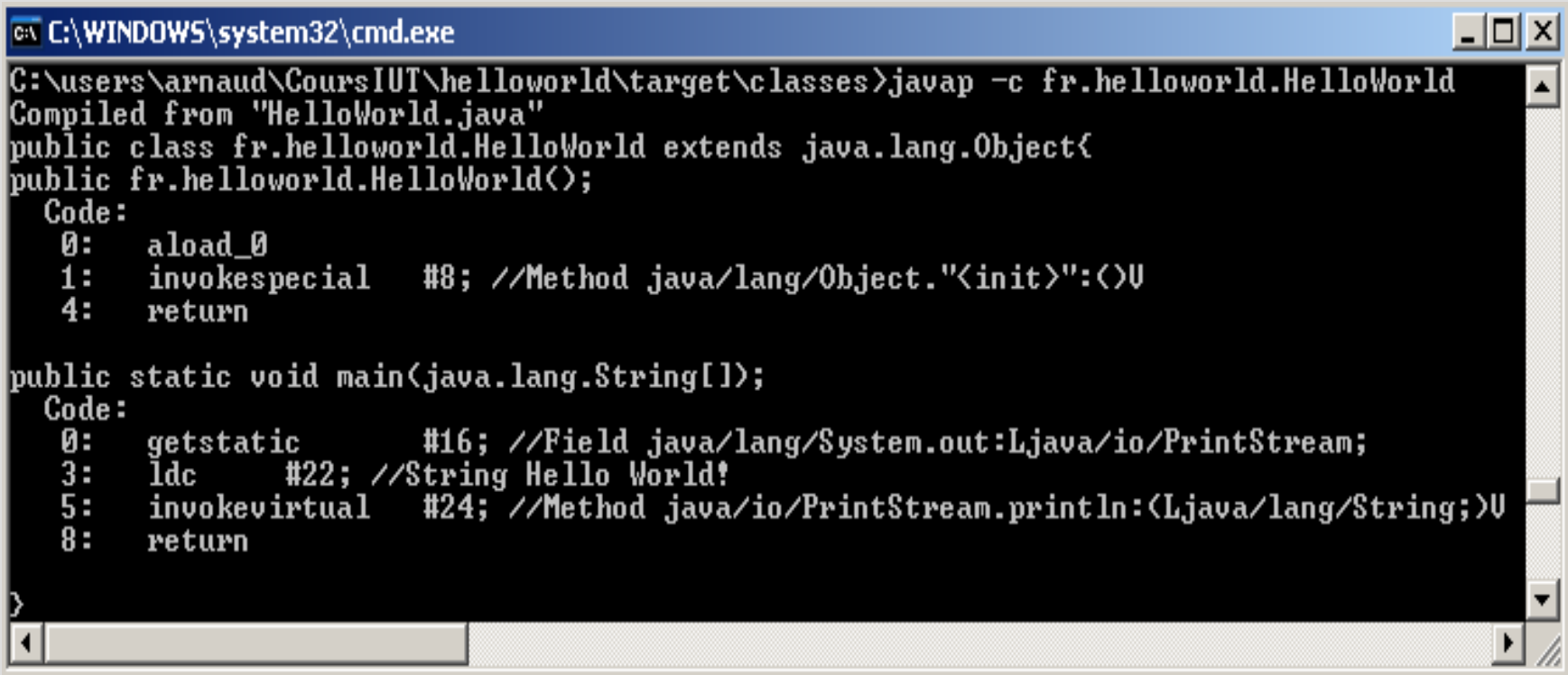
HelloWorld .java/.Class Files

- Package tp.helloworld;
  public class HelloWorld {
    public static void main(String[] args) {
      System.out.println(« Hello World »); } }



Reading .class file : javap -c

- Cf tool in jdk\bin
  javap.exe -c tp.helloworld.HelloWorld



Nommage (Mangling) Interne des Types / Signatures

- En interne les types et signatures sont codés en String (mangling)
- Toutes ces String sont rajoutés dans le ConstantPool de chaque classe
- Tous les objets sont identifiés par leur full classname + name + signature
  - =1/2/3 indexes dans constant pool
  - Signature nécessaire pour l'overloading de noms!!

Mangling : Grammaire (bis)

- (type1 type2 .... typeN) returnType
- Type =
  - « IJSBZFDA» (Int | Long | Short | Byte ...)
  - « L<fullname> ; » (type / classe)
  - « [type» tableau ([[type = table 2D ....)
- Exemple:
  méthode 'void main(String[] args)'
  => «(Ljava/lang/String;)V»

ConstantPool - Binding

- Toutes les références croisées dans les classes sont linkables... via le ConstantPool
- Exemple:
  - lien vers une classe
    = index dans ConstantPool de la String portant le fullname la classe
  - Lien vers un champ
    = index cp nom de classe + index cp nom du champ + index cp de signature du type

Typage/Contenu des .Class

- Les méthodes/fields/constructors sont typés
  - Seul le nom des variables / commentaires / code source n'est pas dans les .class
  - Rappel... le bytecode aussi est typé !
- Les sources/lineNumberTable peuvent être rajoutés (sous forme d'attributs optionnels)
  - Utiles pour débuggés / afficher les exceptions throwées

Plan

- Introduction : Compilation, Runtime, JVM
- Le Bytecode de Java
- « Hello World » .class file détaillé
- « Hello World » exécution pas-à-pas
  - ClassLoader
  - Late Binding, « _quick » bytecodes
- Hotspot optimizer, Myth&Reality

Plan

- Introduction : Compilation, Runtime, JVM
- Le Bytecode de Java
- « Hello World » .class file détaillé
- « Hello World » exécution pas-à-pas
- Hotspot optimizer, Myth&Reality