Java - la plateforme

Java – la plateforme

- Java ?
- VM
- GC
- JIT

Java Aujourd'hui

3 environnements d'exécutions différents

- ~~Java ME (Micro Edition)~~ pour PDA, téléphone
  - Android (Java SE moins certain paquetages)
- Java SE (Standard Edition) pour desktop
- Java EE (Entreprise Edition) pour serveur
  - Servlet/JSP/JSTL/JSF
  - JTA/JTS, EJB,
  - JMS,
  - JavaMail, etc.

Java SE Aujourd'hui

API du JDK 1.8 (~19 000 classes) :

- java.lang, java.lang.reflect, java.lang.invoke
- java.util, java.util.stream, java.util.concurrent
- java.awt, java.applet, javax.swing
- java.io, java.nio, java.net
- java.beans, java.sql, javax.sql

etc...

Java Standard Edition

- JDK 1.0 (1995)
- JDK 1.1 (1997)
- JDK 1.2 - Java 2 (1999)
- JDK 1.3 (2001)
- JDK 1.4 (2002)
- JDK 1.5 - Java 5 (2004)
- JDK 1.6 - Java 6 (2006)
- JDK 1.7 - Java 7 (2011)
- JDK 1.8 - Java 8 (2014)

Compatibilitée ascendante

Java Community Process

Java est un ensemble de spécifications

Le JCP (jcp.org) fait évoluer la plateforme

Le JDK est l'implantation de Java par Oracle, IBM, RedHat, SAP, Apple et des individuels

Deux specs importantes:

- JLS (Java Language Specification)
- JVMS (Java Virtual Machine Specification)

JDK/OpenSource

Le JDK est OpenSource depuis novembre 2006 (complètement en 2008)

Toutes les sources sont disponibles : http://www.openjdk.org

N'importe qui peut contribuer, fixer des bugs, etc.

Créer votre JDK :)

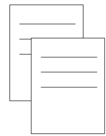
- hg fclone http://hg.openjdk.java.net/jdk8/jdk8
- cd make; make

VM

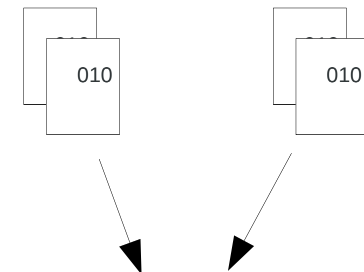
- Java fonctionne sur une machine virtuelle
- bcp de langages utilisent le même principe
  - SmallTalk, Self, Perl, Python, Java, Ruby, JavaScript, C#
- VM
  - WORA (write once run anywhere)
  - Services
    - (classloader, JIT, GC, concurency builtin, etc)

Architecture en C

Codes en Ascii



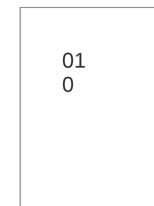
Compilateur



Le code est compilé sous forme objet relogeable

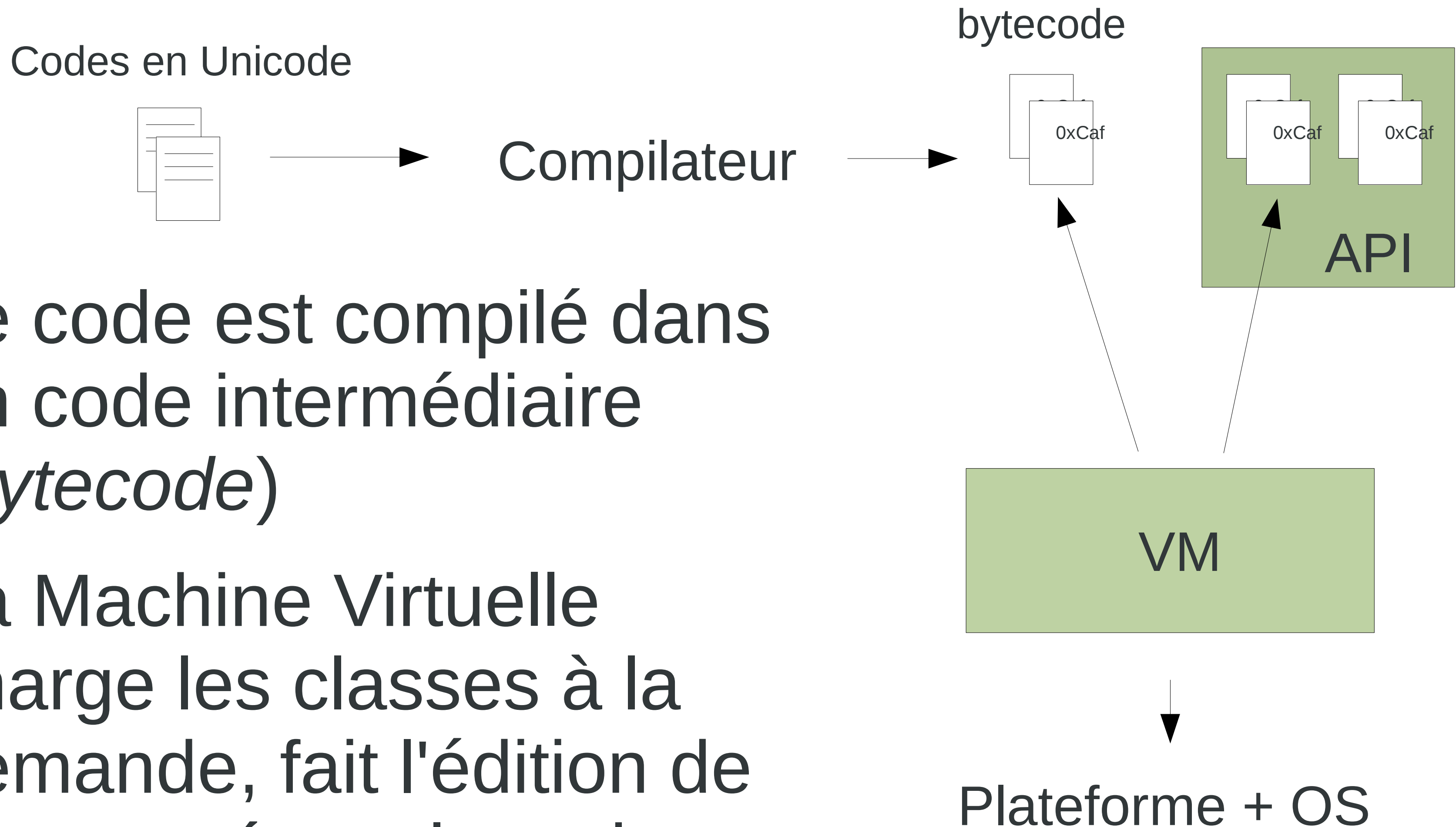
L'éditeur de liens lie les différentes bibliothèques entre elles pour créer l'exécutable

Editeur de lien



Plateforme + OS

Architecture en Java



Interpreteur ?

Certain langages ne possèdent qu'un interpreteur

Perl, CPython, CRuby

Les autres générent du code assembleur à la volée pour augmenter la vitesse exécution (JIT)

- Mixed-mode: interpreter + JIT(1 ou plusieurs)
  - Java (Oracle Hotspot)
- JIT-only
  - Java (IBM J9, Oracle JRockit) / Javascript (Google V8 / Mozilla *Monkey) / C#

Garbage Collector

Algorithme utilisé pour réclamer la mémoire de plusieurs objets lorsque ceux-ci ne sont plus accessibles

Il se déclenche

- Lors d'un new si il n'y a plus assez de mémoire
- Périodiquement
  - en minutes sur un desktop, en milisecondes sur un server

Limitation

Le GC doit connaitre la mémoire totale qui lui sera allouée.

Algos de Gcs
(simplifié)

Dans Hotspot, principalement 3 algos de GCs

- Serial GC (-XX:+UseSerialGC)
  - desktop (1 thread)
- Parallel GC (-XX:+UseParallelGC)
  - desktop/server (n threads mais temps de pause pas garanties)
- CMS (-XX:+UseConcMarkSweepGC)
  - server (n threads, temps de pause best effort)

Et un petit nouveau

- G1 (-XX:+UseG1GC)
  - server (n threads, soft real time)

Mark & Sweep

Phase 1: mark

À partir des piles de chaque threads et des variable statiques, le GC marque tous les objets atteignablent (récursivement)

Phase 2: sweep

Le GC libère la mémoire (comme free) des objets non-marqué

Problèmes:

- plein d'objets ont une durée de vie faible
- la mémoire n'est pas “défragmentée”

Mark & Compact

Phase 1: mark

À partir des piles de chaque threads et des variable statiques, le GC marque tous les objets atteignablent (récursivement)

Phase 2: compact

Le GC recopie tout les objets marqués dans une autres zones

Problèmes:

- “Stop the world”
- Au pire, on a besoin de 2x la place nécessaire

GC générationel

Hypothèses:

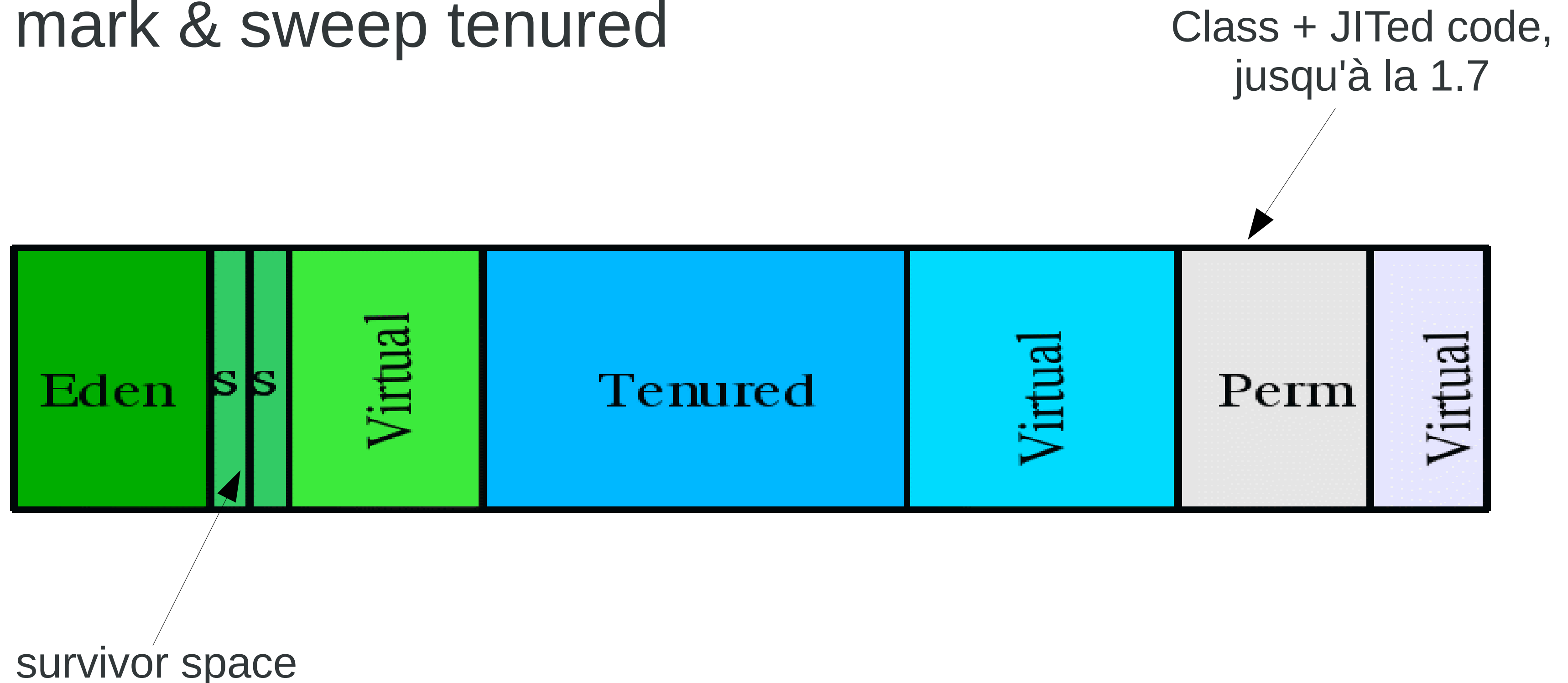
- les objets allouées en même temps ont la même durée de vie
- beaucoup d'objets meurent rapidement

Il n'est nécessaire de collecter l'ensemble de la mémoire, les zones recevant des objets vieux peuvent être collecter moins fréquemment (major collection) que les zones recevant des objets jeunes (minor collection)

Generationnel mark & compact/sweep

Utilise les 2 techniques et les hypothèses générationnels

- mark & compact eden -> survivor, survivor -> survivor, survivor -> tenured
- mark & sweep tenured



Java ne s'optimise pas comme du C

Quelle est la complexité du code ci-dessous ?

Avec un processeur à 1Ghz, combien de temps le calcul prendra t'il ?

```java
public static void main(String[] args) {
  int sum = 0;
  for(int i=0; i<Integer.MAX_VALUE; i++) {
    for(int j=0; j<Integer.MAX_VALUE; j++) {
      sum++;
    }
  }
  System.out.println(sum);
}
```

Java ne s'optimise pas comme du C

Et si le code est le suivant ?

```
public static void main(String[] args) {
  int sum = 0;
  for(int i=0; i<Integer.MAX_VALUE; i++) {
    sum = m(sum);
  }
  System.out.println(sum);
}

private static int m(int sum) {
  for(int j=0; j<Integer.MAX_VALUE; j++) {
      sum++;
  }
  return sum;
}
```

Java ne s'optimise pas comme du C

```
public static void main(String[] args) {
  int sum = 0;
  for(int i=0; i<Integer.MAX_VALUE; i++) {
    sum = m(sum);
  }
  System.out.println(sum);
}

private static int m(int sum) {
  for(int j=0; j<Integer.MAX_VALUE; j++) {
      sum++;
  }
  return sum;
}
```

Le code est "recompilé" en code machine à l'exécution !

```
java -server -XX:+PrintCompilation ...
     72    1 %           Test1::m @ 5 (19 bytes)
   1470    1             Test1::m (19 bytes)
   2836    2 %           Test1::main @ 7 (22 bytes)
```

Optimisations fréquentes

Inlining

Déroulement de boucle

Hoisting des invariants hors des boucles

De-virtualisation (CHA)

Inlining cache, bi-morphic cache

Null check removal

Range check removal

...

Optimisation du code

- On optimise pas car cela coute trop cher !
  - Il faut appeler 10 000 fois une méthode avant que le JIT se déclenche donc pas d'optim avant 10 000 appels
- par contre
  - on choisie le bon algo (sous peine de mort)
  - on ne fait pas deux fois le même calcul
  - on code en suivant le standard
    - les JITs reconnaissent des patterns qu'ils optimisent