IN329 Système d'exploitation
La machine virtuelle Java

Christophe Garion

ISAE/DMIA - SUPAERO/IN
10 avenue Édouard Belin
31055 Toulouse Cedex 4

Plan

Une machine virtuelle

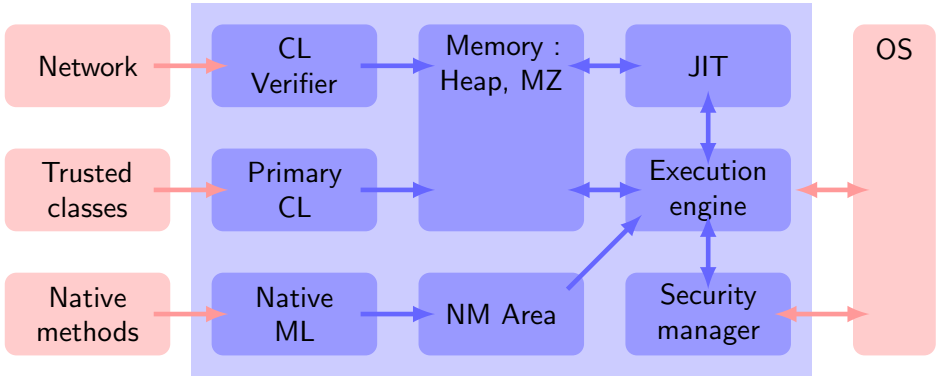
La JVM est une machine virtuelle :

- elle fournit une abstraction de la machine sur laquelle on veut faire tourner un programme ;
- c'est une **spécification** ;
- elle permet (en partie) la portabilité des applications Java ;
- elle interprète les programmes Java sous forme de **bytecode** (code intermédiaire).

Il existe deux versions de la JVM :

- version client (option `-client`), qui réduit le temps de démarrage et l'occupation mémoire
- version serveur (option `-server`) qui maximise la rapidité d'exécution du programme

Architecture de la JVM



Une machine à pile ?

La JVM est une **machine à pile** : elle fait abstraction du nombre de registres du matériel qui l'accueille. Elle a un jeu d'instructions très simple.

Considérons l'instruction suivante :

```
i = j + k;
```

Voici le code généré sur une machine à pile :

```
push j     // put j on the stack
push k     // put k on the stack
add        // add the two values, remove them
           // put the result on stack
store i    // take the top of stack and store it in i
```

Inconvénients : plus d'accès mémoire
Avantages : pas de décodage et de recherche des opérandes dans les registres (tout est sur la pile !)

Quelques options non standard de la JVM

- `-Xint` : mode interprété pur
- `-Xbatch` : la compilation passe en premier plan
- `-Xfuture` : verification stricte (pas le cas en 1.1)
- `-Xnoclassgc` : pas de ramasse-miettes
- `-Xincgc` : ramasse-miettes incrémental (moins de latence, plus de CPU utilisé)
- `-Xloggc` : log des événements du GC (voir aussi `-verbose :gc`)
- `-Xmsn` : taille initiale du tas (par défaut 2Mo), `n` étant la taille
- `-Xmxn` : taille maximale du tas (par défaut 64Mo), `n` étant la taille
- `-Xprof` : *profiling* du programme sur la sortie standard
- `-Xrunhprof` : *profiling* du CPU, de la pile ou des moniteurs (voir `java -Xrunhprof :help`)

Options avancées de la JVM

Java HotSpot VM options.
`http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp`.

Java tuning white paper.
`http://java.sun.com/performance/reference/whitepapers/tuning.html`.

- `-XX :-UseParallelGC` : utilisation du *garbage collector* distribué
- `-XX :-DisableExplicitGC` : plus d'appels à `System.gc()`
- `-XX :+AggressiveOpts` : optimisation agressive (par défaut à partir du SE 6.0)
- `-XX :+StringCache` : cache pour les chaînes de caractères souvent utilisées
- ...

Outils à disposition

Certaines options sont gérables directement ou via les JMX (*Java Management Extensions*).

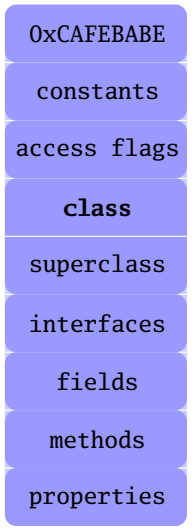
Chung, M.
Monitoring and managing Java SE 6 platform applications.
`http://java.sun.com/developer/technicalArticles/J2SE/monitoring/`.

Quelques outils (en ligne de commande) :

- `jconsole` : outil générique de surveillance d'une application
- `jhat` : analyse du tas
- `jstat` : statistiques diverses
- `jstack` : trace de la pile

Qu'y a-t-il dans un fichier .class ?

Les sources Java sont compilés en *bytecode* contenu dans des fichiers **.class** clairement définis.
Dans ces fichiers, on trouve un certain nombre d'informations :



4 premiers octets : magic number
4 octets suivants : numéros de version
Zone des « constantes » : tableau qui contient toutes les informations symboliques typées.

- *CONSTANT_Class*
- *CONSTANT_Fieldref*
- *CONSTANT_Methodref*
- *CONSTANT_...* pour les types primitifs
- *CONSTANT_NameAndType*
- *CONSTANT_Utf8*

Dans la suite du fichier, tous les symboles sont référencés par des index sur ce tableau.
access flags : classe, interface, publique, abstraite

Fichier .class : un exemple

```
public class Personne {

    private String nom;
    private int age;

    public Personne(String nom, int age) {
        this.nom = nom.trim();
        this.age=age;
    }

    public String getNom() {
        return this.nom;
    }

    public int getAge() {
        return this.age;
    }
}
```


Fichier .class : format hexadécimal

```
cafe babe 0000 0031 001f 0a00 0600 150a  .......1........
0016 0017 0900 0500 1809 0005 0019 0700  ................
1a07 001b 0100 036e 6f6d 0100 124c 6a61  .......nom...Lja
7661 2f6c 616e 672f 5374 7269 6e67 3b01  va/lang/String;.
0003 6167 6501 0001 4901 0006 3c69 6e69  ..age...I...<ini
743e 0100 1628 4c6a 6176 612f 6c61 6e67  t>...(Ljava/lang
2f53 7472 696e 673b 4929 5601 0004 436f  /String;I)V...Co
6465 0100 0f4c 696e 654e 756d 6265 7254  de...LineNumberT
6162 6c65 0100 0667 6574 4e6f 6d01 0014  able...getNom...
2829 4c6a 6176 612f 6c61 6e67 2f53 7472  ()Ljava/lang/Str
696e 673b 0100 0667 6574 4167 6501 0003  ing;...getAge...
2829 4901 000a 536f 7572 6365 4669 6c65  ()I...SourceFile
0100 0d50 6572 736f 6e6e 652e 6a61 7661  ...Personne.java
0c00 0b00 1c07 001d 0c00 1e00 100c 0007  ................
0008 0c00 0900 0a01 0008 5065 7273 6f6e  ..........Person
6e65 0100 106a 6176 612f 6c61 6e67 2f4f  ne...java/lang/O
626a 6563 7401 0003 2829 5601 0010 6a61  bject...()V...ja
7661 2f6c 616e 672f 5374 7269 6e67 0100  va/lang/String..
0474 7269 6d00 2100 0500 0600 0000 0200  .trim.!.........
0200 0700 0800 0000 0200 0900 0a00 0000  ................
0300 0100 0b00 0c00 0100 0d00 0000 3600  ..............6.
0200 0300 0000 122a b700 012a 2bb6 0002  .......*...*+...
b500 032a 1cb5 0004 b100 0000 0100 0e00  ...*............
0000 1200 0400 0000 0600 0400 0700 0c00  ................
0800 1100 0900 0100 0f00 1000 0100 0d00  ................
0000 1d00 0100 0100 0000 052a b400 03b0  ..........*.....
0000 0001 000e 0000 0006 0001 0000 000c  ................
0001 0011 0012 0001 000d 0000 001d 0001  ................
0001 0000 0005 2ab4 0004 ac00 0000 0100  ......*.........
0e00 0000 0600 0100 0000 1000 0100 1300  ................
0000 0200 14                             .....
```

0000 0031
➥ major version 49 **001f**
➥ 31 constantes utilisées
09 0005 0018
➥ field
➥ class #5
➥ name and type #24
07 001a
➥ class
➥ name #26
01 0008 5065...
➥ UTF-8 : Personne
09 0005 0018
➥ field
➥ class #5
➥ name and type #24
0c 0007 0008
➥ name and type
➥ name #7

Fichier .class : désassemblage

Désassemblage avec `javap -verbose -c Personne` :

```
Compiled from "Personne.java"
public class Personne extends java.lang.Object
  SourceFile : "Personne.java"
  minor version : 0
  major version : 49
  Constant pool :
const #1 = Method       #6.#21 ; //  java/lang/Object."<init>" :()V
const #2 = Method       #22.#23 ;       //  java/lang/String.trim :()Ljava/lang/String;
const #3 = Field        #5.#24 ; //  Personne.nom :Ljava/lang/String;
const #4 = Field        #5.#25 ; //  Personne.age :I
const #5 = class        #26 ;    //  Personne
const #6 = class        #27 ;    //  java/lang/Object
const #7 = Asciz        nom;
const #8 = Asciz        Ljava/lang/String;;
const #9 = Asciz        age;
const #10 = Asciz       I;
const #11 = Asciz       <init>;
const #12 = Asciz       (Ljava/lang/String;I)V;
const #13 = Asciz       Code;
const #14 = Asciz       LineNumberTable;
const #15 = Asciz       getNom;
const #16 = Asciz       ()Ljava/lang/String;;
const #17 = Asciz       getAge;
const #18 = Asciz       ()I;
const #19 = Asciz       SourceFile;
const #20 = Asciz       Personne.java;
const #21 = NameAndType #11 :#28 ;//  "<init>" :()V
...
```

Fichier .class : désassemblage

```
{
public Personne(java.lang.String, int);
  Code:
   Stack=2, Locals=3, Args_size=3
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   aload_0
   5:   aload_1
   6:   invokevirtual   #2; //Method java/lang/String.trim:()Ljava/lang/String;
   9:   putfield        #3; //Field nom:Ljava/lang/String;
   12:  aload_0
   13:  iload_2
   14:  putfield        #4; //Field age:I
   17:  return
  LineNumberTable:
   line 6: 0
   line 7: 4
   line 8: 12
   line 9: 17
```


Fichier .class : désassemblage

```
public java.lang.String getNom();
  Code:
   Stack=1, Locals=1, Args_size=1
   0:   aload_0
   1:   getfield        #3; //Field nom:Ljava/lang/String;
   4:   areturn
  LineNumberTable:
   line 12: 0

public int getAge();
  Code:
   Stack=1, Locals=1, Args_size=1
   0:   aload_0
   1:   getfield        #4; //Field age:I
   4:   ireturn
  LineNumberTable:
   line 16: 0

}
```


Méthode getNom : que se passe-t-il ?

Code mnémoniques :

```
0:   aload_0
1:   getfield        #3; //Field nom:Ljava/lang/String;
4:   areturn
```

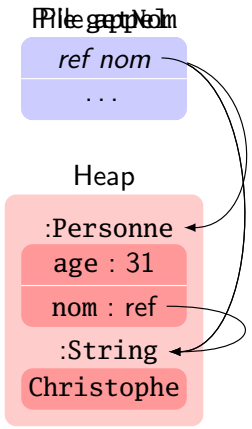
aload_0 [... ⇒ ..., objectref]

aload_0 place la référence vers l'objet index #0 sur la pile

➥ il s'agit de **this**

getfield [..., objectref ⇒ ..., value]

getfield #3 prend la valeur sur la pile (c'est une référence) et utilise l'index #3 de la table des constantes pour obtenir un attribut.
On résoud ensuite « l'attribut », on prend sa valeur dans la référence obtenue sur la pile et on place cette valeur sur la pile.



Organisation mémoire : grands principes

4 grandes zones :

- la **zone des méthodes** qui contient le code à exécuter
- le **tas** qui contient les objets alloués par **new**
- la **pile** qui contient les contextes (**frames**) d'invocation des méthodes. Elle est gérée par les registres optop, frame et vars
- les **registres** : les trois registres de piles et le compteur ordinal

Attention :

	JVM	thread
Method area	×	
Heap	×	
Stack		×
Registries		×

S'il n'y a plus de place, une erreur OutOfMemoryError est levée.

Le tas (heap) et la zone des méthodes

La zone du tas (*heap*) est l'endroit où vivent les objets Java.

Elle est automatiquement gérée par un processus, le ramasse-miettes (*garbage-collector*).

La zone des méthodes est analogue avec la zone mémoire contenant le code à exécuter dans le cas d'un code compilé.

Elle contient le *bytecode* des méthodes et des constructeurs (`<init>`) des classes.

Le compteur de programme pointe dessus.

Elle contient également la table des constantes permettant l'identification des constantes des classes.

Elle devrait faire partie du tas, mais on peut choisir de ne pas utiliser de GC dessus par exemple.

Pile, contexte et registres

La pile Java stocke les paramètres, les résultats intermédiaires, les retours de méthode etc.

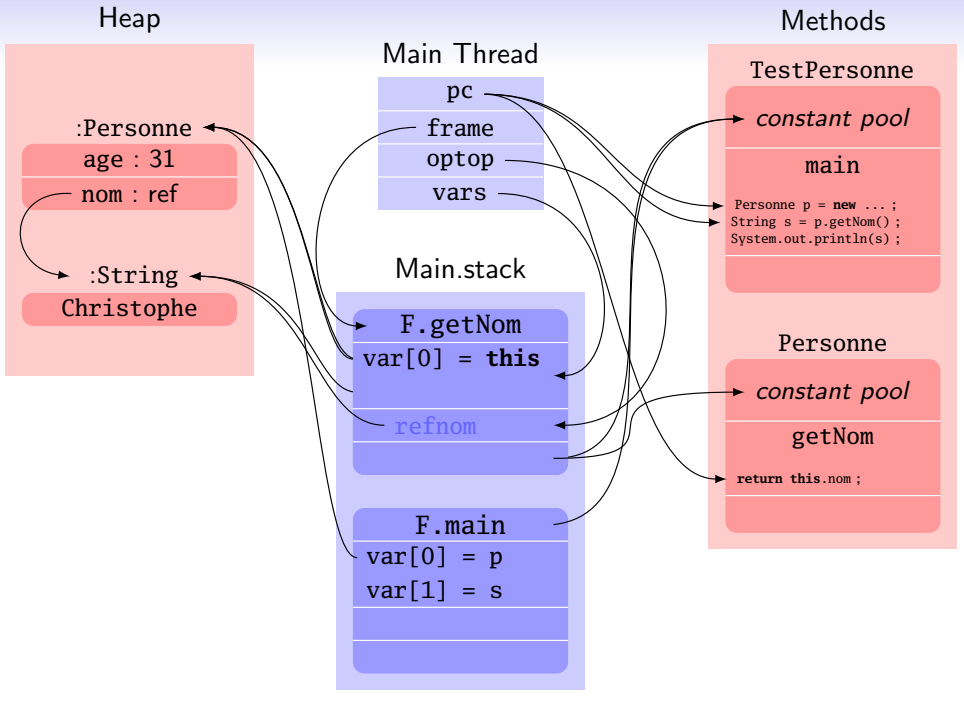
Pour chaque méthode exécutée, il existe un contexte d'exécution (*stack frame*) qui est une certaine zone de la pile :

- zone des variables locales (pointée par `vars`) représentée par un tableau. Cette zone contient également **`this`** (index 0) et les paramètres de la méthode.
- zone opérandes (pointée par `optop`) : paramètres des instructions etc. C'est une pile de longueur maximale déterminée à la compilation. Elle contient des valeurs des types défini dans Java.
- zone d'environnement d'exécution (pointée par `frame`) pour conserver des informations sur les manipulations de la pile.
- la pile contient également une référence vers la zone *constant pool* de la classe de la méthode.

Organisation mémoire de la JVM : résumé

```
public class TestPersonne {

    public static final void main(final String[] args) {
        Personne p = new Personne("Christophe", 31) ;
        String s = p.getNom() ;
        System.out.println(s) ;
    }
}
```


Organisation mémoire de la JVM : résumé



Manque de mémoire sur le tas : exemple

```
public class MemErr {
    public static void main(String[] args) throws Exception {
        Thread.sleep(5000) ;

        Cellule aux = null;

        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            aux = new Cellule(aux) ;
        }
    }
}

class Cellule {
    Cellule suivante;

    Cellule (Cellule suivante_) {
        this.suivante  = suivante_;
    }
}
```


Manque de mémoire sur le tas : exemple

```
% java MemErr

Exception in thread "main" java.lang.OutOfMemoryError : Java heap space
        at MemErr.main(MemErr.java :8)
```

Quelles erreurs ?

- heap space (ici) : pas de place pour allouer de la mémoire pour un objet
- PermGen space : pas de place pour charger une classe ou stocker une chaîne de caractères interne
- *native memory error* : problème d'allocation sur la pile native

Le compilateur JIT

L'interprétation est très pénalisante en terme de performances (par exemple pour les boucles) : utilisation d'un **compilateur Just in Time**.

Est-ce que le compilateur JIT compile chaque méthode ?

- temps de compilation préjudiciable pour l'utilisateur
- même s'il pouvait optimiser complétement le code, ça ne marcherait pas pour Java du fait de son caractère dynamique :
  - vérification dynamique de types fréquentes (*cast*, tableaux)
  - objets alloués sur le tas (pas toujours le cas pour C++)
  - méthodes principalement virtuelles, donc *inlining* difficile
  - chargement dynamique des classes, donc optimisation globale difficile pour le compilateur

La JVM HotSpot

La JVM HotSpot :

- démarre un interpréteur et lance le programme
- analyse le code pendant qu'il tourne et détecte les points chauds qui sont exécutés souvent (ils sont quand même interprétés !)
- données collectées pendant l'exécution (temps, appel de méthodes virtuelles etc.)
- *inline* le code : moins d'appel de méthodes et optimisation plus efficace (code plus gros)
- déoptimisation/réoptimisation dynamique (à cause du chargement dynamique)

Quelques optimisations du compilateur :

- *inlining* comme vu précédemment
- détection de type efficace
- élimination de vérification de dépassement de tableau
- boucles « déroulées »
- *profiling* avant compilation en natif

Un exemple d'inlining sur la JVM

```
import java.util.Vector;

public class InlineMe {

    int counter=0;

    public void method1() {
        for(int i=0;i<1000;i++)
        addCount();
        System.out.println("counter="+counter);
    }

    public int addCount() {
        counter=counter+1;
        return counter;
    }

    public static void main(String args[]) {
        InlineMe im=new InlineMe();
        im.method1();
    }
}
```


Un exemple de profiling sur la JVM

```
CPU TIME (ms) BEGIN (total = 277) Thu Nov 16 22 :16 :44 2006
rank   self  accum   count trace method
   1 10.11% 10.11%       2 300706 java.net.URLClassLoader.findClass
   2  7.22% 17.33%       4 300532 sun.misc.URLClassPath$JarLoader.parseExtensions
   3  6.14% 23.47%      19 300279 sun.nio.cs.SingleByteEncoder.encodeArrayLoop
   4  5.42% 28.88%       1 300964 sun.net.www.protocol.file.Handler.createFileURI
   5  3.97% 32.85%       1 301313 InlineMe.method1
   6  3.61% 36.46%     931 300277 java.lang.String.charAt
   7  3.61% 40.07%    1037 300695 java.lang.Math.max
   8  2.89% 42.96%     931 300276 sun.nio.cs.Surrogate.is
   9  2.89% 45.85%      19 300280 sun.nio.cs.SingleByteEncoder.encodeLoop
  10  2.89% 48.74%       4 300545 java.util.jar.JarFile.hasClassPathAttribute
  11  2.17% 50.90%    1000 301223 InlineMe.addCount
  12  1.81% 52.71%       4 300238 sun.net.www.ParseUtil.decode
  13  1.81% 54.51%       2 301278 java.io.FileOutputStream.write
  14  1.44% 55.96%       4 300242 java.io.UnixFileSystem.normalize
  15  1.44% 57.40%       4 300517 sun.misc.URLClassPath$JarLoader.getJarFile
  16  1.08% 58.48%     195 300231 java.lang.String.length
```


Un exemple de profiling sur la JVM : inline

```
CPU TIME (ms) BEGIN (total = 218) Thu Nov 16 23 :59 :33 2006
rank   self  accum   count trace method
   1  9.63%  9.63%      19 300279 sun.nio.cs.SingleByteEncoder.encodeArrayLoop
   2  5.50% 15.14%       4 300545 java.util.jar.JarFile.hasClassPathAttribute
   3  3.21% 18.35%      19 300280 sun.nio.cs.SingleByteEncoder.encodeLoop
   4  3.21% 21.56%       1 301313 InlineMeNow.method1
   5  2.75% 24.31%    1000 301223 InlineMeNow.addCount
   6  2.29% 26.61%     940 300276 sun.nio.cs.Surrogate.is
   7  2.29% 28.90%     940 300277 java.lang.String.charAt
   8  2.29% 31.19%    1037 300695 java.lang.Math.max
   9  1.83% 33.03%       4 300011 java.security.AccessControlContext.optimize
  10  1.83% 34.86%       4 300238 sun.net.www.ParseUtil.decode
  11  1.83% 36.70%       4 300517 sun.misc.URLClassPath$JarLoader.getJarFile
  12  1.83% 38.53%       1 300964 sun.net.www.protocol.file.Handler.createFileURI
  13  1.38% 39.91%      15 300178 java.lang.AbstractStringBuilder.append
  14  1.38% 41.28%     191 300234 java.lang.StringBuilder.append
  15  1.38% 42.66%       4 300242 java.io.UnixFileSystem.normalize
  16  1.38% 44.04%       1 300442 java.lang.Class.forName
```


Alors, est-ce que ça marche ?

Java plus rapide que du C ?

P.Lewis, J. and Neumann, U. (2004).
Performance of Java versus C++.
http://www.idiom.com/~zilla/Computer/javaCbenchmark.html.

Problématique

Le ramasse-miette est un processus chargé de libérer la mémoire allouée dans le tas par l'opérateur **new**.

Les objets qui ne sont plus référencés dans le programme sont des candidats potentiels au recyclage.

- permet d'éviter la fragmentation et l'augmentation de la taille du tas ;
- libère le programmeur de la tâche de libérer la mémoire (en particulier quand la libérer) ;
- sécurité ;
- charge de calcul supplémentaire ;

Ça fonctionne : création de 100000 d'instances d'`Object` et trace avec `-verbose :gc`

```
[GC 512K->122K(1984K), 0.0028650 secs]
```


Comment recycler la mémoire ?

Deux objectifs pour le ramasse-miettes :

- trouver les objets récupérables
- réclamer l'espace du tas utilisé par ces objets

C'est un champ d'étude à part entière !

Quelques techniques :

- comptage de références :
  - quand un objet est référencé, on incrémente son compteur ;
  - quand un objet est déréférencé, on décrémente son compteur ;
  - un objet dont le compteur est à 0 est récupérable ;
  - mais références cycliques et temps CPU...
- traçage des objets :
  - graphe des objets : on marque les objets accessibles à partir d'objets basiquement accessibles
  - *mark and sweep*
  - point d'ancrages : variables locales, zone opérandes, variables de classe
  - algorithmes conservatifs

Lutter contre la fragmentation

Compactage :

- on place tous les objets dans une extrémité libre du tas
- l'autre extrémité est libre
- modification des références via un degré d'indirection ou pas

Copie :

- on déplace les objets vers une nouvelle zone
- on les colle les uns aux autres
- on peut les déplacer au fur et à mesure de leur découverte

Peut-on interagir avec le GC depuis Java ?

Comment interagir avec le ramasse-miettes depuis Java ?

La méthode `finalize` d'`Object` :

- **public void** `finalize()` **throws** `Throwable`
- appelée avant la réclamation de l'espace mémoire de l'objet
- utile pour les ressources ouvertes
- utiliser **super**`.finalize`
- résurrection unique d'un objet

Des méthodes « utiles » :

- `System.gc()`
- `System.runFinalization()`
- `Runtime.gc()`

Une modélisation des références en Java

Peut-on « ramasser » un objet même si l'on a une référence dessus (exemple d'images dans une application avec ascenseurs) ?

On peut utiliser des objets références dont le rôle est de maintenir une référence vers un objet référent. On n'utilise pas alors de poignée directe sur l'objet.

On peut pour cela utiliser la classe abstraite `java.lang.ref.Reference` :

- `public Object get()`
- `public void clear()`
- `public boolean enqueue()`
- `public boolean isEnqueued()`

Force des références et cycle de vie d'un objet

On dispose de trois classes de références : `SoftReference`, `WeakReference`, `PhantomReference`.

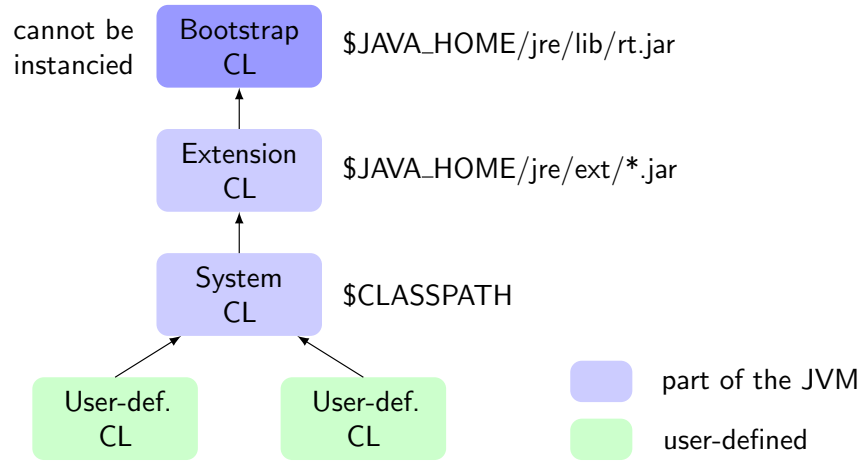
Elles correspondent aux différents états d'atteignabilité d'un objet :

- les références normales sont appelées **références fortes**. Un objet est dit **atteignable fortement** s'il existe une chaine de références fortes vers cet objet ;
- un objet **« légérement » atteignable** n'est pas atteignable fortement, mais atteignable par une chaine qui contient une réference légère. Il peut être ramassé ;
- un objet **faiblement atteignable** n'est pas atteignable « légérement », mais atteignable par une chaine qui contient une réference faible. Il sera ramassé ;
- un objet est « **finalize reachable** » s'il n'est plus faiblement atteignable, mais il n'a pas été finalisé ;
- un objet est « **phantom reachable** » s'il a été finalisé mais est atteignable par une chaine contenant une référence fantôme. Sa méthode `get` renvoie **`null`** ;
- un objet n'est plus atteignable s'il n'est plus atteint par aucune référence.

Références : un exemple

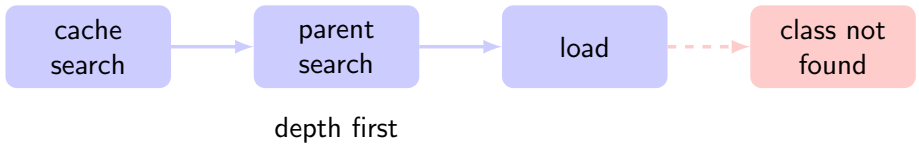
```
import java.lang.ref.* ;
import java.io.File ;

public class DataHandler {
    private File lastFile ;
    private WeakReference lastData ;

    byte[] readFile(File file) {
        byte[] data ;

        if (file.equals(lastFile)) {
            data = (byte[]) lastData.get() ;
            if (data != null) {
                return data ;
            }
        }

        data = readBytesFromFile(file) ;
        lastFile = file ;
        lastData = new WeakReference(data) ;
        return data ;
    }
}
```


Architecture du class loader



Architecture du Class loader

Fonctionnement lors de la demande de chargement d'une classe :



depth first

Peut lever une `ClassNotFoundException`.

Class loader et espace de nommage

Lorsqu'une classe est chargée, la JVM « retient » quel *class loader* a été utilisé.

On utilise le même *class loader* pour les classes qui ont une relation de dépendance :

Principe (utilisation de class loader)

Si un objet de type C est instancié dans A, alors C sera chargée via le class loader de A

Principe (espace de nommage)

Une classe ne peut par défaut qu'interagir avec les classes qui ont été chargées par le même class loader.

On peut donc avoir plusieurs classes différentes qui portent le même nom...

Le vérificateur de bytecode

Le vérificateur garantit que les classes chargées ont une structure interne correcte (en particulier les instructions de saut qui doivent « rester » dans la méthode).
4 passes :

1. au **chargement** du fichier, vérification de la structure du fichier *magic number*, bon nombre d'octets etc.)
2. au **link** de la classe, on vérifie dans un premier temps un certain nombre de propriétés ne dépendant pas du code à exécuter :
   - existence de la super-classe
   - attributs et méthodes avec des noms valides
   - respect de **`final`**
3. toujours au **link**, on analyse le code de chaque méthode :
   - les variables locales sont accédées si elles ont été initialisées
   - les méthodes sont invoquées avec les bons arguments
   - les attributs ont des valeurs du bon type
   - les *opcodes* ont les bons arguments
4. vérification des références symboliques à l'**exécution**, par exemple chargement des classes nécessaires.

Comprendre la classe `ClassLoader`

`ClassLoader` est une classe abstraite. Elle possède une méthode particulière à redéfinir :

```
protected Class findClass(String name) throws ClassNotFoundException
```

C'est cette méthode qui charge le *bytecode* de la classe.

Quelques méthodes utiles :

- `getClassLoader()` sur un objet de type `Class`
- `getSystemClassLoader()`
- **`public class`** `loadClass(String name)` qui fonctionne comme suit :
  - utilise `findLoadedClass` pour voir si la classe n'a pas été chargée précédemment
  - sinon on appelle `loadClass` sur le *class loader* père
  - si elle n'est toujours pas chargée, on utilise `findClass`
- `resolveClass(Class c)` qui *link* la classe

Comment utiliser un class loader ?

Chargement de la classe `Personne` :

```
Class personne = monCL.loadClass("Personne", true);
```

Création d'une instance de `Personne` :

```
Object o = personne.newInstance();
```

Utilisation de l'objet ?

```
((Personne) o).getName() // ne fonctionne pas par defaut !
```

Il faut une classe ou une interface en commun entre les deux *class loaders*.

Références

Java SE HotSpot at a glance.
http://java.sun.com/javase/technologies/hotspot/index.jsp.

K. Arnold, J. Gosling, and D. Holmes.
The Java Programming Language.
Java Series. Addison-Wesley, fourth edition, 2005.

J. Gosling, B. Joy, G. Steele, and G. Bracha.
The Java Language Specification.
Java Series. Prentice Hall, 3rd edition, 2005.
Available on http://java.sun.com/docs/books/jls/index.html.

T. Lindholm and F. Yellin.
The Java Virtual Machine.
Addison-Wesley Professional, 2nd edition, 1999.
Available on http://java.sun.com/docs/books/jvms/.

S. Wilson and J. Kesselman.
Java Platform Performance : Strategies and Tactics.
Prentice Hall, 2001.
Available on http://java.sun.com/docs/books/performance/.