INSA
CENTRE VAL DE LOIRE

POO Avancée : Java

JVM : Java Virtual Machine

Salwa SOUAF

Outline

JVM Introduction

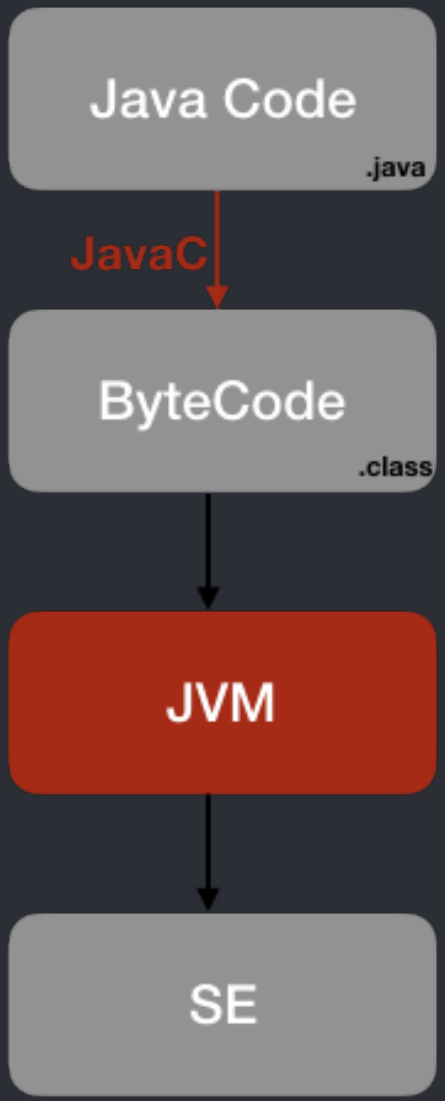
A la différence des langages classiques write once, compile anywhere, le langage Java est du type compile once, run anywhere. Le code compilé, le *bytecode* peut être exécuté indifférement sur une machine virtuelle implémentée pour fonctionner sur Windows, Linux, Android, etc...

Definition

Java Virtual Machine (JVM) est l'environnement d'exécution pour applications Java proposant une couche d'abstraction entre l'application Java et le SE.

Il existe plusieurs JVM: JVM SUN, JVM IBM, JAVA Apple...

JVM Introduction



Roles

- Interprétation du bytecode
- Interaction avec le système d'exploitation

Outline

Bytecode

Le *bytecode* est une séquence d'instruction pour la machine virtuelle. C'est le résultat de la compilation du code source Java. La JVM stocke pour chaque classe chargée le flot de *bytecode* associé à chaque méthode. Une méthode peut être par exemple constituée du flot ci-dessous :

```
// Bytecode stream: 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

opcodes

Le nombre d'opcodes est petit ce qui permet de faire tenir tous les opcodes sur un octet (i.e Byte).
Exemple des opcodes:

- iconst_X: empiler la constante X sur la pile
- iload_X: empiler la variable locale n°X
- istore_X: dépiler un entier et le stocker dans la variable locale n°X
- i2f: convertir un int en float
- iadd, imul, iinc...: opérations arithmétiques
- ireturn: retourne le résultat

Bytecode

Example : From Source Code to ByteCode

Source Code Snippet :

```
byte a = 1;
byte b = 1;
byte c = (byte) (a + b);
return c;
```

Compiled :

```
iconst_1  // Push int constant 1.
istore_1  // Pop into local variable 1, which is a: byte a = 1;
iconst_1  // Push int constant 1 again.
istore_2  // Pop into local variable 2, which is b: byte b = 1;
iload_1   // Push a (a is already stored as an int in local variable 1).
iload_2   // Push b (b is already stored as an int in local variable 2).
iadd      // Perform addition. Top of stack is now (a + b), an int.
int2byte  // Convert int result to byte (result still occupies 32 bits).
istore_3  // Pop into local variable 3, which is byte c: byte c = (byte) (a + b);
iload_3   // Push the value of c so it can be returned.
ireturn   // Proudly return the result of the addition: return c;
```

Bytecode

Decompilation à l'aide de l'outil javap

Source Code :

```
public class Decompilation {
  int test() {
      byte a = 1;
      byte b = 1;
      byte c = (byte) (a + b);
      return c;
  }

  public static void main(String[] args) {
      Decompilation d = new Decompilation();
      int res = d.test();
      System.out.println("Out: " + res);
  }
}
```

La décompilation à l'aide de l'outil *javap* :

```
javap -c -public  Decompilation > Decompilation.txt
```

- -public: Shows only public classes and members.
- -protected: Shows only protected and public classes and members.
- -package: Shows only package, protected, and public classes and members.
- -private: Shows all classes and members.

Decompilation à l'aide de l'outil javap : Résultat

```
Compiled from "Decompilation.java"
class Decompilation extends java.lang.Object{
public static void main();
Code:
 0:  new #2; //class Decompilation
 3:  dup
 4:  invokespecial #3; //Method "<init>":()V
 7:  astore_0
 8:  aload_0
 9:  invokevirtual #4; //Method test:()I
 12: istore_1
 13: getstatic #5; //Field java/lang/System.out:Ljava/io/PrintStream;
 16: new #6; //class java/lang/StringBuilder
 19: dup
 20: invokespecial #7; //Method java/lang/StringBuilder."<init>":()V
 23: ldc #8; //String Out:
 25: invokevirtual #9; //Method java/lang/StringBuilder.append:...
 28: iload_1
 29: invokevirtual #10; //Method java/lang/StringBuilder.append:...
 32: invokevirtual #11; //Method java/lang/StringBuilder.toString:...
 35: invokevirtual #12; //Method java/io/PrintStream.println:...
 38: return
}
```

Il existe des outils permettant de décompiler le bytecode et de revenir jusqu'au code source, par exemple JD (http://jd.benow.ca/)

Outline

Chargement dynamique de code

L'utilisation de *bytecode* intermédiaire impose de résoudre les dépendances entre classes lors de l'exécution. Cela n'empêche pas le compilateur de réaliser des vérifications entre classes, par exemple la présence ou non d'une fonction appelée sur un objet de type B depuis un objet de type A. La machine virtuelle cherche, dans le **CLASSPATH**, les classes mentionnées après les directives **import**.

```
import p.Decompilation;
  public class Chargement {
      public static void main() {
          Decompilation d = new Decompilation(); }}
```

A la compilation si on obtient :

```
javac Chargement.java
Chargement.java:1: package p does not exist
import p.Decompilation;
       ^
1 error
```

le compilateur ne trouve pas la classe **Decompilation** dans le sous répertoire **p** du **CLASSPATH**. Si celui-ci est situé dans **unautreendroit**, il faut mettre à jour le **CLASSPATH** comme suit:

```
export CLASSPATH=./unautreendroit:$CLASSPATH
```

Chargement dynamique de code

CLASSPATH

Le **CLASSPATH** donne la liste des emplacements où la machine virtuelle est autorisée à charger des classes. S'il s'agit d'un nom de répertoire, il désigne la racine de l'arborescence correspondante aux packages. Si le **CLASSPATH** contient des fichiers *jar*, les classes sont cherchées et chargées directement depuis l'intérieur de l'archive (i.e la racine de l'arborescence correspondant à la racine de l'archive).

Chargement dynamique de code

JARs

Un fichier JAR est un type de fichier compressé (similaire aux fichiersZIP). La spécification des fichiers JAR décrit l'utilisation du **Manifest** qui permet d'ajouter des informations pour l'utilisation du JAR. Ce **Manifest** contient:

- Des informations générales (version, date et auteur, **CLASSPATH** des ressources requises).
- La classe contenant le main si ce jar exécutable (i.e. contient une application qui est lancée via l'exécution de *java -jar x.jar*).
- Des informations pour les applets embarquées dans le jar.
- Des informations de signature.

Operation	Command
To create a JAR file	`jar cf jar-file input-file(s)`
To view the contents of a JAR file	`jar tf jar-file`
To extract the contents of a JAR file	`jar xf jar-file`
To extract specific files from a JAR file	`jar xf jar-file archived-file(s)`
To run an application packaged as a JAR file (requires the `Main-class` manifest header)	`java -jar app.jar`

Chargement dynamique de code

ClassLoader

Le chargement dynamique de classe repose sur l'utilisation d'un chargeur de classe (classloader). A partir du nom de la classe, il localise (notamment en parcourant le **CLASSPATH**) ou génère les données qui définissent la classe. A partir des données récupérées, il permet de créer un objet de type **Class** (type que l'on retrouve quand on fait de l'introspection). La méthode importante du chargeur de classe, qui est invoqué lorsqu'on réalise une instanciation est:

```
// L'appel permettant de créer l'objet Class à partir de son nom
Class r = loadClass(String className, boolean resolveIt);
```

Le booléen permet de spécifier si un lien permanent est créé entre la classe chargée et son nom. Les étapes de l'implémentation de **loadClass** sont:

- vérifications (nom, classe déjà chargée, classe system)
- chargement des données
- définition de la classe (conversion de bytes en Class)
- résolution de la classe (lier la classe au nom)
- retourne l'objet classe

Chargement dynamique de code

ClassLoader : Hiérarchie de délégation

Les différents chargeurs de classe sont hiérarchisés selon un modèle de délégation.

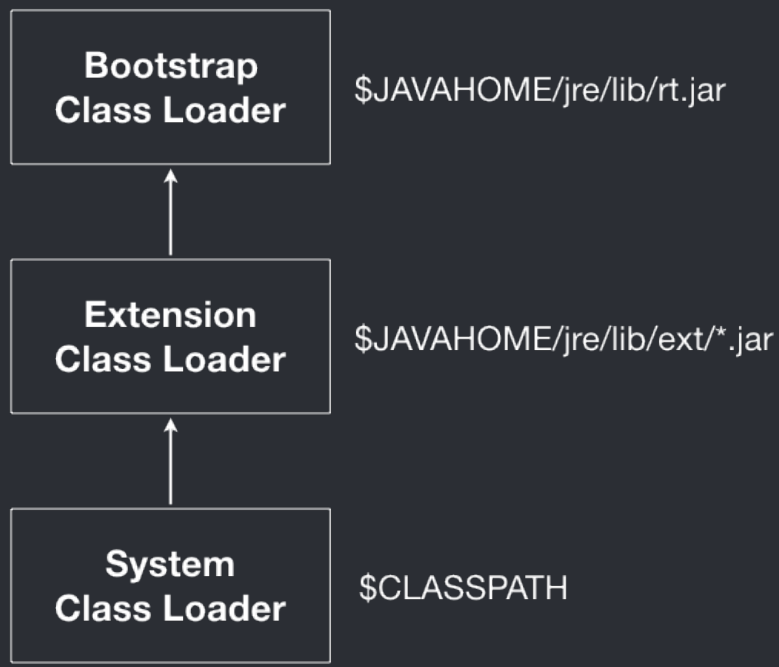
La méthode *loadClass* opère de la sorte:

- Si la classe est déjà chargée, elle est retournée
- Sinon, le chargeur délègue le chargement à son parent
- Si le parent ne trouve pas la classe, le chargeur:
  - appelle **findClass** pour la trouver
  - charge la classe le cas échéant

Chargement dynamique de code

ClassLoader : Hiérarchie de délégation

- Bootstrap Class Loader: ce CL est créé au démarrage de la JVM. Il s'occupe de charger les classes des APIs standard (dans le JRE), y inclues les classes objets. Contrairement aux autres CLs, celui ci n'est pas implémenté en JAVA.
- Extension Class Loader: ce CL s'occupe des classes d'extensions installées et des APIs non standards.
- System Class Loader: ce CL s'occupe des classes dans le CLASSPATH.



Chargement dynamique de code

ClassLoader : Personnalisation

Un chargeur de classe personnalisé peut être créer. Il est possible de l'utiliser explicitement ou bien en le précisant au lancement de la machine virtuelle:

```
package cl;
public class CustomCL extends ClassLoader {
      public Class<?> loadClass(String name) throws ClassNotFoundException {
              System.out.println("Custom class loader. Chargement de: " + name);
              System.out.println("Chargement via mon père...");
              return super.loadClass(name);
      }}
```

```
java -Djava.system.class.loader=cl.CustomCL cl.Main
```

Ce qui donne pour un programme Main:

```
System.out.println("Démarrage...");
System.out.println("Mon classLoader est: " + Main.class.getClassLoader());
Integer i = new Integer(4);
A a = new A();
System.out.println("Fin.");
```

```
Custom class loader. Chargement de: cl.Main
Chargement via mon père...
Démarrage...
Mon classLoader est: sun.misc.Launcher$AppClassLoader@3432a325
Fin.
```

Chargement dynamique de code

Personnalisation ClassLoader : Surcharge

```
    public Class<?> loadClass(String name) throws ClassNotFoundException {
            System.out.println("Custom class loader. Chargement de: " + name);
             if (name.startsWith("cl2.")) {
                return getClass(name); // Chargement spécialisé
            }
            System.out.println("Chargement via mon père...");
            return super.loadClass(name);
    }

    private Class<?> getClass(String name)
            throws ClassNotFoundException {
            String file = name.replace('.', File.separatorChar) + ".class";
            byte[] b = null;
            try {
                b = loadClassData(file);
                Class<?> c = defineClass(name, b, 0, b.length);
                resolveClass(c);
                return c;
            } catch (IOException e) { e.printStackTrace(); return null; }
        }

    private byte[] loadClassData(String name) throws IOException {
    InputStream stream = getClass().getClassLoader() .getResourceAsStream(name);
    int size = stream.available();
    byte buff[] = new byte[size];
    DataInputStream in = new DataInputStream(stream);
    in.readFully(buff); in.close();
    return buff;
}
```

Chargement dynamique de code

Personnalisation ClassLoader : Résultat

Le branchement dans la méthode *loadClass* permet d'appeler une implémentation spécifique de *getClass*. L'appel à la méthode *defineClass* charge, depuis le fichier, le tableau contenant le code de cette classe particulière. Comme *defineClass* est une méthode final, on passe la main à l'implémentation de la JVM pour réaliser cela et l'objet *Class* résultant possède alors un pointeur vers le chargeur de classe qui l'a chargé. Ainsi, tous les objets internes à cette classe pourront être eux aussi chargés par ce chargeur de classe personnalisé. Avec ce chargeur, le **Main** précédent affiche:

```
Custom class loader. Chargement de: cl2.Main
Custom class loader. Chargement de: java.lang.Object
Chargement via mon père...
Custom class loader. Chargement de: java.lang.String
Chargement via mon père...
Custom class loader. Chargement de: java.lang.System
Chargement via mon père...
Custom class loader. Chargement de: java.io.PrintStream
Chargement via mon père...
Démarrage...
Custom class loader. Chargement de: java.lang.StringBuilder
Chargement via mon père...
Custom class loader. Chargement de: java.lang.Class
Chargement via mon père...
Mon classLoader est: cl2.CustomCL@565f0e7d
Custom class loader. Chargement de: java.lang.Integer
Chargement via mon père...
Custom class loader. Chargement de: cl2.A
Fin.
```

Chargement dynamique de code

ClassLoader : Limites

- Charge les classes sur demande (i.e lors d'un **new** ou d'un appel à une méthode statique).
- Charge les classes linéairement et s'arrête à la première classe correspondant au nom complet demandé.
- **CLASSPATH** est plat sans aucune notion de dépendances entre jar.
- L'ordre des jars dans le **CLASSPATH** est important.
- Aucune vérification n'est faite à l'exécution sur l'existence de plusieurs occurrences d'une même classe.
- Aucune vérification n'est faite au démarrage de l'application sur la présence de tous les jars / classes nécessaires au bon fonctionnement de l'application.

Outline

Modularisation

Java 9 était livré avec un ensemble de fonctionnalités riche. Bien qu'il n'y avait pas de nouveaux concepts de langage, les nouvelles API et les commandes de diagnostic sont certainement intéressantes pour les développeurs.
Avec cette version de Java, l'écosystème du langage a radicalement changé. Avant, le langage était contenu dans quelques bibliothèques assez grosses : **rt.jar** qui faisait près de 53 Mo. Maintenant le langage est découpé en modules, avec un module de base **java.base** qui définit les API de base de la plate-forme Java SE qui lui fait près de 16 Mo.
Cette version a introduit la notion de modularisation :

- CLASSPATH sous forme d'arbre de dépendances
- Vérification de la présence de tous les modules nécessaires à notre application au démarrage.
- Renforcement de la sécurité, seuls les packages exportés explicitement par un module sont visibles par un autre.
- JVM modulaire

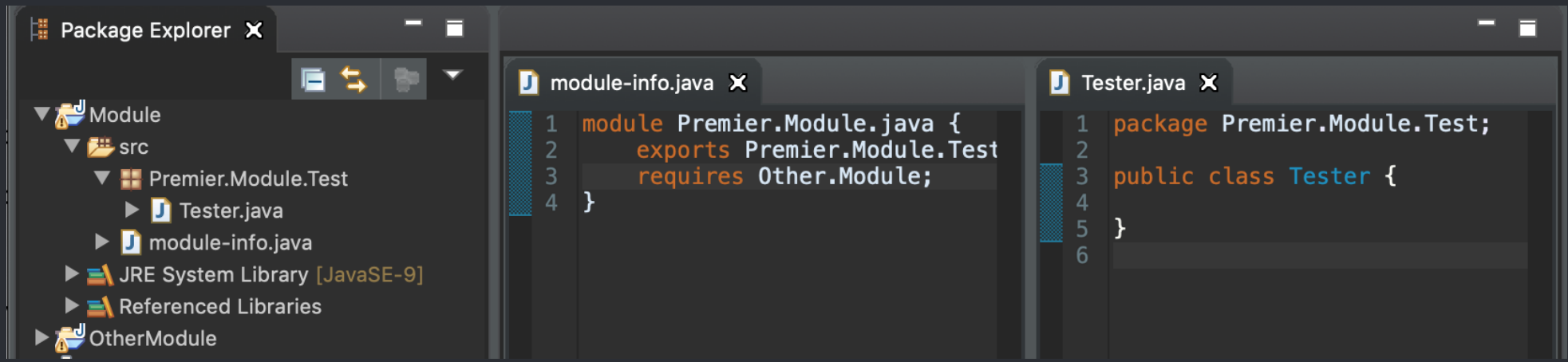
JVM Modulaire

Le **rt.jar** a été découpé en modules afin de :

- Réduire la taille physique du JRE sur les appareils embarqués.
- Les classes internes de la JVM sont maintenant réellement privées.
- Augmenter la sécurité des applications : moins de classes chargées par la JVM => réduction de la surface d'attaque.

Module

- Le nom d'un module doit être unique, tout comme pour un nom de package.
- Chaque module ne doit contenir qu'un et un seul fichier **module-info.java** et il doit être placé à la racine du module.
- Un module peut contenir plusieurs packages mais un package ne peut être que dans un et un seul module.
- Le fichier module-info.java doit être à la racine de votre code source.
- **java.base** est le module de base requis par tous et résolu automatiquement par tous les modules.



Module

Options

- requires <module> : accéder à un autre module.
- requires transitive <module> : pour pouvoir accéder aux classes de ce deuxième module lorsqu'on fait appel au premier.
- requires static <module> : représente le concept de dépendance optionnelle :
  - obligatoire à la compilation : une erreur de compilation sera levée si le module n'est pas présent dans le module path à la compilation.
  - optionnel à l'exécution : le module ne sera pas pris en compte dans la phase de sanity check au démarrage de l'application. L'application pourra démarrer même si le module n'est pas présent.
- export <package> : rendre les types public du package donné visibles depuis les autres modules.
- export <package> to <module> : rendre les types public du package donné visibles depuis un module précis.
- open module <module> et opens <package> : Pour accéder par réflexion à des champs privés. (i.e permet l'introspection des classes [WE WILL SEE THAT LATER])

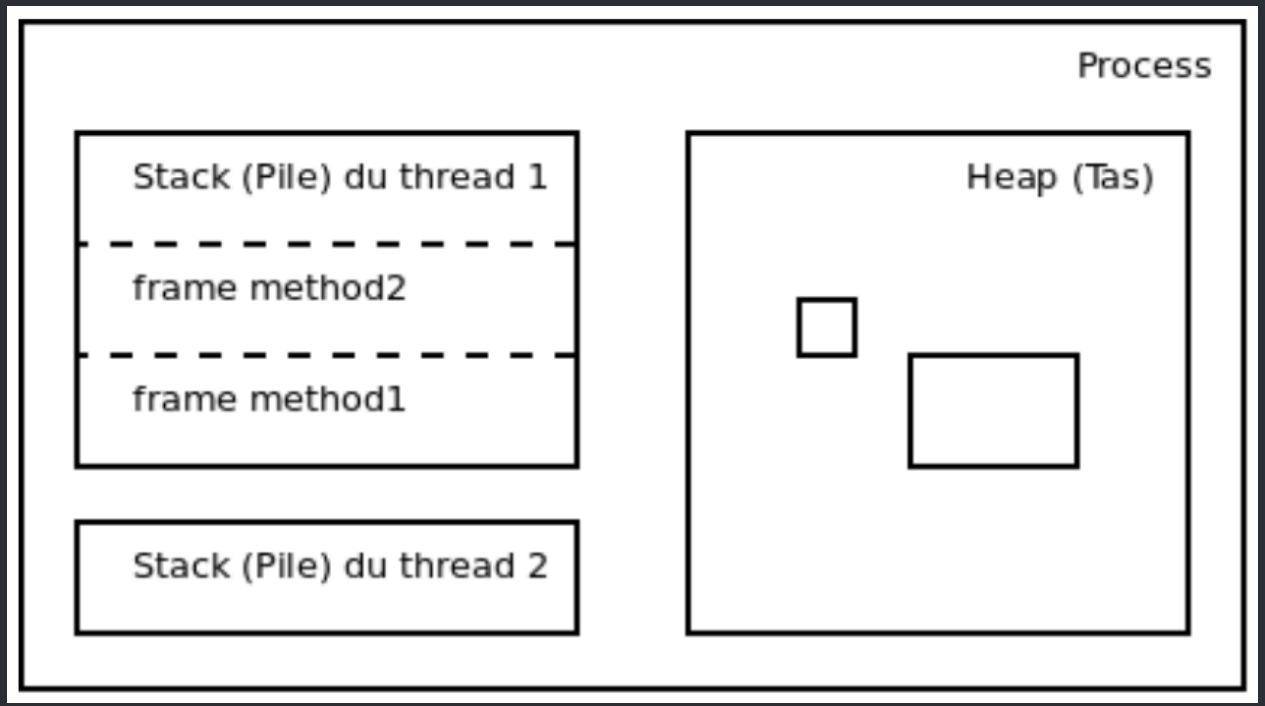
Outline

Données Runtime

Un programme Java peut être multithreadé. Chaque thread géré par la JVM possède un *program counter register* contenant l'adresse de l'instruction actuellement exécutée.

Chaque **thread** Java possède une pile (stack). Une pile stocke des frames. Cela permet de gérer les variables locales, résultats partiels, appels de fonction. La mémoire de cette pile n'a pas besoin d'être contigüe.

Au démarrage de la JVM, un tas (heap) est créé. Ce tas contient les objets et tableaux instanciés. Ces objets ne sont jamais désalloués (cf garbage collector). Ce tas est lui aussi de taille fixe ou dynamique, et non nécessairement contigu.



Données Runtime

Personalized sizes

La spécification permet à une JVM d'avoir une pile de taille fixe (StackOverflowError) ou dynamique (OutOfMemoryError). On peut spécifier la taille de la pile au lancement de la JVM:

```
java  -Xss1024k maClasse
// peut lever une exeception java.lang.StackOverflowError .
```

On peut aussi préciser la taille du tas:

```
java  -Xmx512M maClasse
```

La JVM contient aussi une zone de méthode (method area) qui stocke les structures de classe, attributs, codes de méthodes et constructeurs. Cette zone peut faire partie du tas. La JVM contient un pool de données constantes (runtime constant pool) qui est une table par classe contenant des constantes connues à la compilation ou des attributs déterminés au runtime. Enfin, la JVM peut contenir des méthodes natives, par exemple écrites en C.

Données Runtime

Frames

Une frame est utilisée pour stocker des données locales, des résultats partiels, des valeurs de retour pour les méthodes, et les pointeurs vers les objets du tas. Une frame n'existe que le temps d'existence d'une méthode et est empilée sur la pile, ainsi que le thread ayant créé cette frame. Chaque frame possède:

- son propre tableau de variables locales
- sa propre pile d'opérandes
- une référence sur le *runtime* constant pool de la classe de la méthode courante

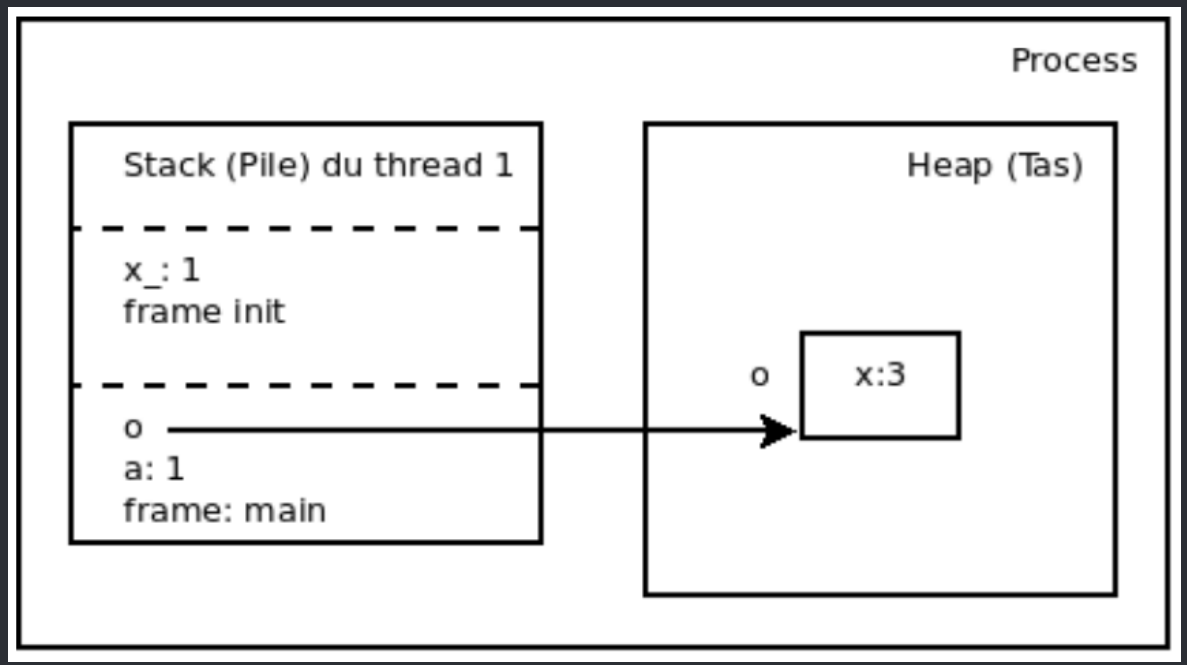
Par conséquent, pour chaque thread, une seule frame est active à la fois à un moment donné. La frame courante cesse de l'être si une méthode est appelée ou si la méthode termine.

Données Runtime

Exemple

```
public class MyObj {
        public int x;
        void init(int x_) {
                x = x_;
        }

        public static void main(String[] args) {
                int a = 1;
                MyObj o = new MyObj();
                o.init(a);
        }
}
```



Données Runtime

Operand stack

La pile d'opérandes d'une fonction contient les variables manipulées par le *bytecode*. Dans l'exemple suivant, l'opcode **aload_0** pousse l'objet de la variable 0, ici **this**, sur la pile d'opérandes. Puis **iload_1** charge la valeur de la variable locale 1. Puis, **putfield** consomme ces deux valeurs empilées pour modifier le champ numéro 18 de l'objet considéré.

```
    void init(int arg0) {
        /* L6 */
        0 aload_0;                  /* this */
        1 iload_1;                  /* x_ */
        2 putfield 18;              /* .x */
        /* L7 */
        5 return;
    }

    public static void main(java.lang.String[] args) {
        /* L10 */
        0 iconst_1;
```

Bytecode | Source Code

Console | Tasks | Problems | Executables | Variables | Operand Stack

Off:	Bytecode Instruction	Operand Stack before	Operand Stack after	Operand Stack depth
0	aload_0	<empty>	L this	1
1	iload_1	L this	L this, I x_	2
2	putfield	L this, I x_	<empty>	0
5	return	<empty>	<empty>	0

Outline

Garbage Collector

Dans Java, la gestion du nettoyage de la mémoire est délégué à la JVM au travers du ramasse-miette (garbage collector). Le but principal est d'éviter au développeur la gestion de la désallocation des ressources inutilisées, d'éviter des bugs de double désallocation, voire même de perdre du temps à désallouer.
Tous les objets sont stockés dans le tas (heap) après des appels à l'opérateur new.
Les objets sont collectés par le ramasse-miette lorsque:

- le programme ne référence plus l'objet du tas
- aucun objet référencé ne contient de référence vers l'objet du tas Le ramasse-miette cherche aussi à combattre la fragmentation du tas. La taille du tas, même dynamique, coûte en espace et en temps.

L'implémentation du ramasse-miette est libre. La spécification de la JVM dit juste que le tas doit être garbage collected. La difficulté d'implémentation du ramasse-miette réside dans le fait qu'il faut garder une trace des objets utilisés ou non, puis les détruire. Cela coûte d'avantage en temps CPU qu'une désallocation manuelle.
A la destruction, la JVM appelle la méthode finalize, dernière méthode étant appelée:

```
protected finalize() throws Throwable { ... }
```

Garbage Collector

Atteignabilité - 1

La manière la plus simple de déterminer les objets à collecter est de calculer l'atteignabilité d'un objet. Un objet est dit atteignable *ssi* il existe un chemin de références depuis les racines (les références du programme) jusqu'à cet objet. Tous les objets sont situés dans le tas. Les variables locales sont, quant à elles, sur la pile. Chaque variable locale est soit un type primitif, soit une référence d'objet. Ainsi, les racines sont donc l'union de de toutes les références des piles. Les racines contiennent aussi les objets constants comme les *String* dans le **runtime** constant pool. Par exemple, le **runtime** constant pool peut pointer sur des String de la pile contenant le nom de la classe ou des méthodes.

Garbage Collector

Atteignabilité - 2

Deux grandes familles de ramasse-miettes peuvent être distingués:

- ramasse-miettes par décompte de références (reference counting)
- ramasse-miettes par traces (tracing)

Le ramasse-miette par décompte de référence collecte les objets lorsque le compteur tombe à zéro. Le problème d'un tel ramasse-miette est l'overhead d'incrémentation/décrémentation mais surtout la non détection de cycles d'objets.
Le ramasse miette par traces parcourt le graphe des objets et pose une marque sur chacun d'eux. Puis les objets non marqués sont supprimés du tas. On appelle ce processus *mark and sweep*.
Quel que soit le type de ramasse-miette, la présence de méthodes de finalisation impacte la phase sweep. Les objets non marqués sans méthode *finalize* peuvent être collectés à moins qu'ils ne soient référencés par des objets non marqués ayant une méthode *finalize*.

Garbage Collector

Fragmentation - 1

Pour combattre la fragmentation, deux stratégies similaires permettent de réduire la fragmentation de la pile après la dés-allocation d'objets:

- Défragmentation par compactage (compacting)
- Défragmentation par copie (copying)

La défragmentation par compactage déplace les objets marqués d'un côté de la pile, afin d'obtenir une large plage contigüe de l'autre: on compacte les objets d'un côté. En insérant un niveau d'indirection dans les adresses du tas d'objets, on évite le rafraichissement de toutes les références de la pile.

Garbage Collector

Fragmentation - 2

La défragmentation par copie déplace tous les objets dans une nouvelle zone de la pile. L'intégralité des objets étant déplacés, ils sont assurément contigus. Pour réaliser cette opération, une traversée totale des objets depuis les racines est obligatoire. Dans le cas d'un ramasse-miette de ce type, on peut combiner la recherche d'objets à collecter et la défragmentation: l'ancienne zone mémoire est donc par conséquent libre ou contenant des objets à désallouer. Ce processus élimine la phase de marquage et de collecte (*mark and sweep*).
Une implémentation classique de ce procédé est appelé le ramasse-miette *stop and copy*. La pile est divisée en deux régions et les allocations se font dans l'une d'elle jusqu'à épuisement. Le programme est alors arrêté et les objets copiés dans l'autre partie à partir de la traversée du graphe d'objets. Le programme est relancé et les allocations se font dans la nouvelle région.

Garbage Collector

Ramasse-miette générationnel

La durée de vie des objets impacte l'efficacité du ramasse-miette. De nombreux objets (plus de 98% dans des mesures expérimentales) ont une durée de vie très courte: *most objects die young*. Le ramasse-miette par copie est dans ce cas très efficace car les objets meurent car ils ne sont pas visités lors de la copie. Si l'objet survit après le premier passage du ramasse-miette, il y a de grandes chances qu'il devienne permanent. Le ramasse-miette par copie est peu performant sur ce type d'objets. Inversement, le ramasse-miette par marquage-compactage est performant sur les objets à durée de vie longue puisqu'ils sont compactés d'un côté de la pile et ne sont ensuite plus recopiés. Les ramasse-miettes par marquage sont pour leur part plus longs à exécuter puisqu'ils doivent examiner de nombreux objets dans la pile.

Un ramasse-miette générationnel se base sur le principe de ségrégation des générations: certains objets sont dits jeunes et d'autres sont promues à un niveau de génération supérieur. Pour simplifier on peut considérer deux génération: *young generation et old generation*.

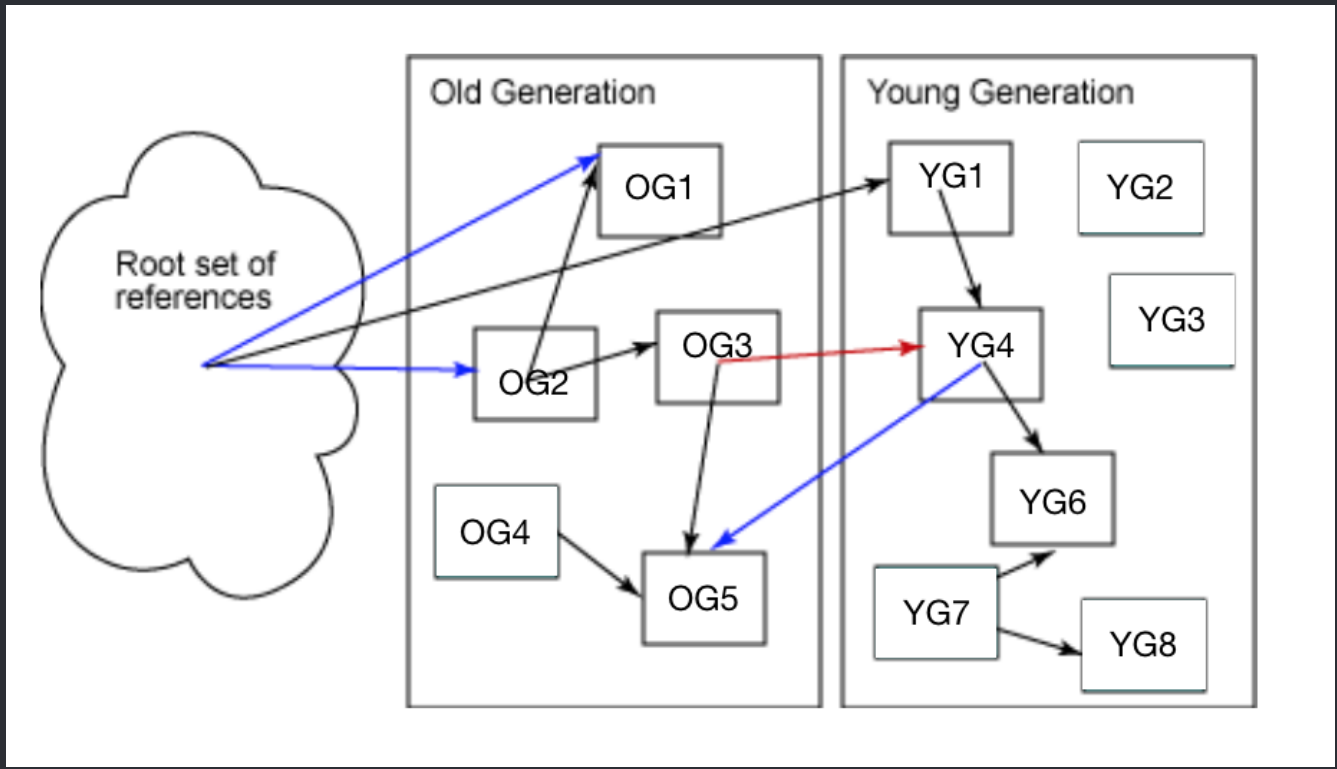
Garbage Collector

Références inter-générationnelles

Lorsqu'une allocation échoue, un trigger minor collection est déclenché, qui engendre la collection de la génération la plus jeune. Cette collection peut être très rapide et récupérer un espace mémoire conséquent. Si la récupération est suffisante, le programme peut continuer, sinon une autre génération est attaquée. La collection par générations successives peut engendrer la collection d'un objet jeune alors qu'un objet vieux pointe encore sur cet objet. En effet, le tas est divisé en plusieurs générations et tous les types de ramasse-miette par copie ou marquage doivent parcourir les objets depuis les racines. L'algorithme de parcourt se limite donc à une génération du tas, évitant de parcourir l'intégralité du tas. Il peut alors considérer un objet comme à collecter car non marqué, alors qu'un objet de la vieille génération pointe encore dessus.
Le ramasse-miette doit donc construire la liste des références inter-générationnelles, par exemple lors de la promotion d'un objet jeune dans la vieille génération (méthode la plus efficace). Lors de l'analyse de la *minor collection*, ces références inter-générationnelles sont alors considérées comme des références racines, résolvant le problème évoqué ci-avant.

Garbage Collector

Références inter-générationnelles



Garbage Collector

Références inter-générationnelles