Java Virtual Machine

Karima Ennaoui

karima.ennaoui@insa-cvl.fr

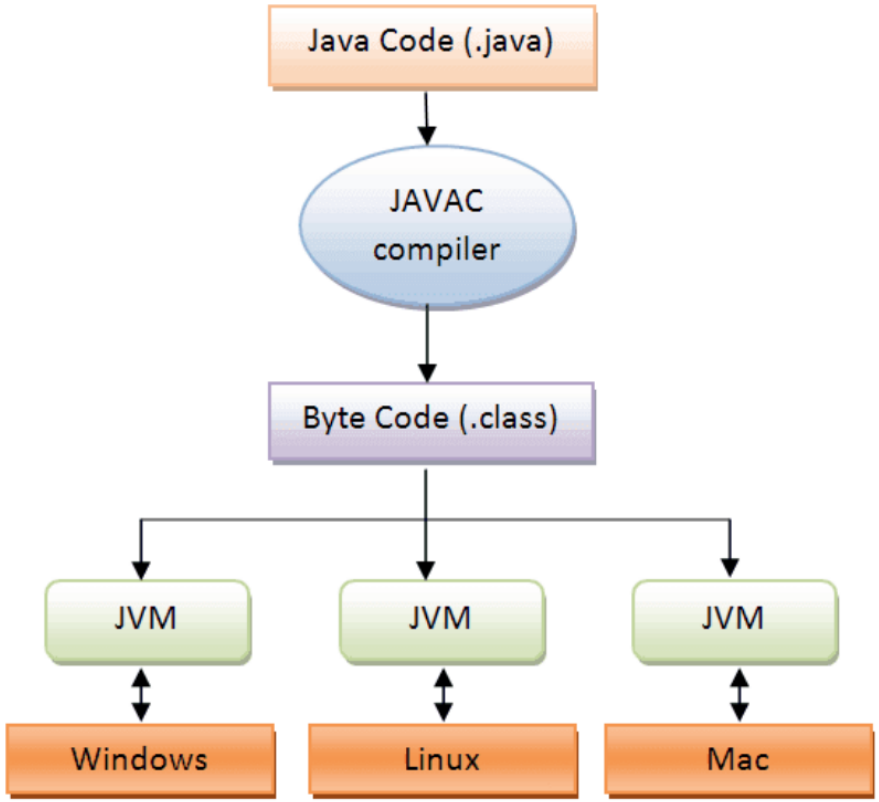
INSA CVL, 4A STI

Bibliographie: Bytecode basics, A first look at the bytecodes of the Java virtual machine, Bill Venners, JavaWorld.com, 09/01/96
https://www.cubrid.org/blog/understanding-jvm-internals/
Inspiré des slides de Jean-François Lalande

December 6, 2018

Introduction de la JVM

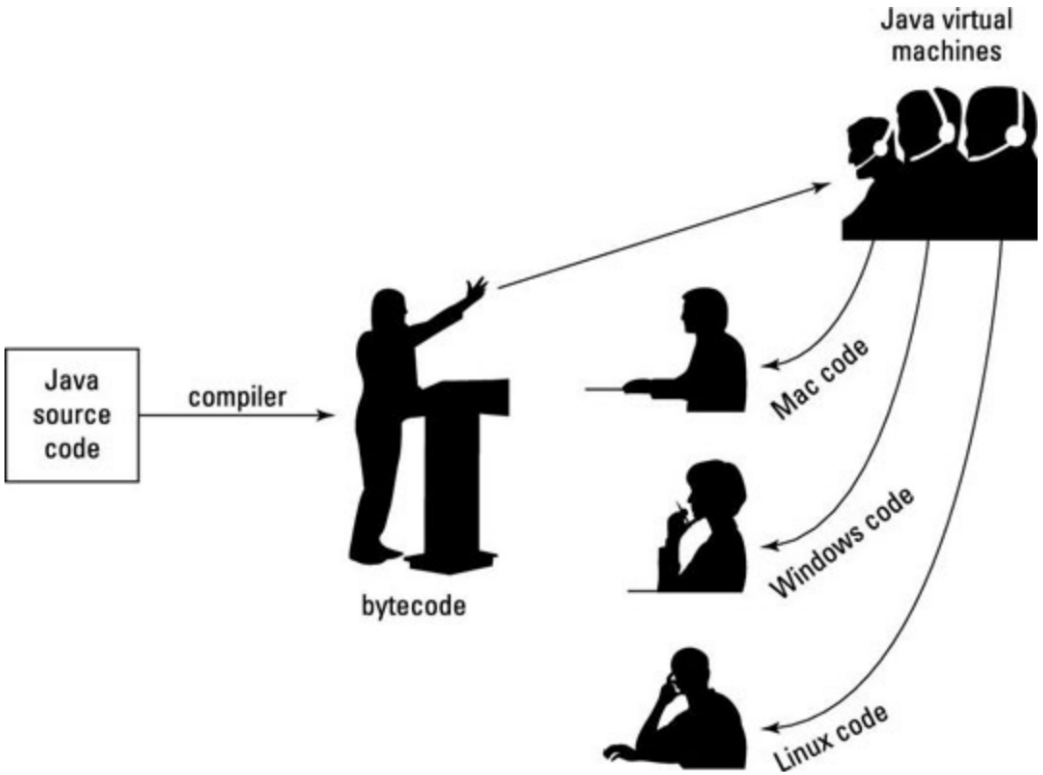
Java Virtual Machine (JVM) est l'environnement d'exécution pour applications Java proposant une couche d'abstraction entre l'application Java et le SE.



Introduction de la JVM

- Il existe plusieurs JVM: JVM SUN, JVM IBM, JAVA Apple...
- Pour tester si une VM est une JVM, elle doit répondre aux spécifications imposées dans le JSR (Java Specification Request).
- Le TCK (Technology Compatibility Kit) est un ensemble de tests utilisé (par la JCP, Java Comunity Process) afin de tester une nouvelle JVM.
- Dalvik VM pour android ne suit pas les spécifications JVM.

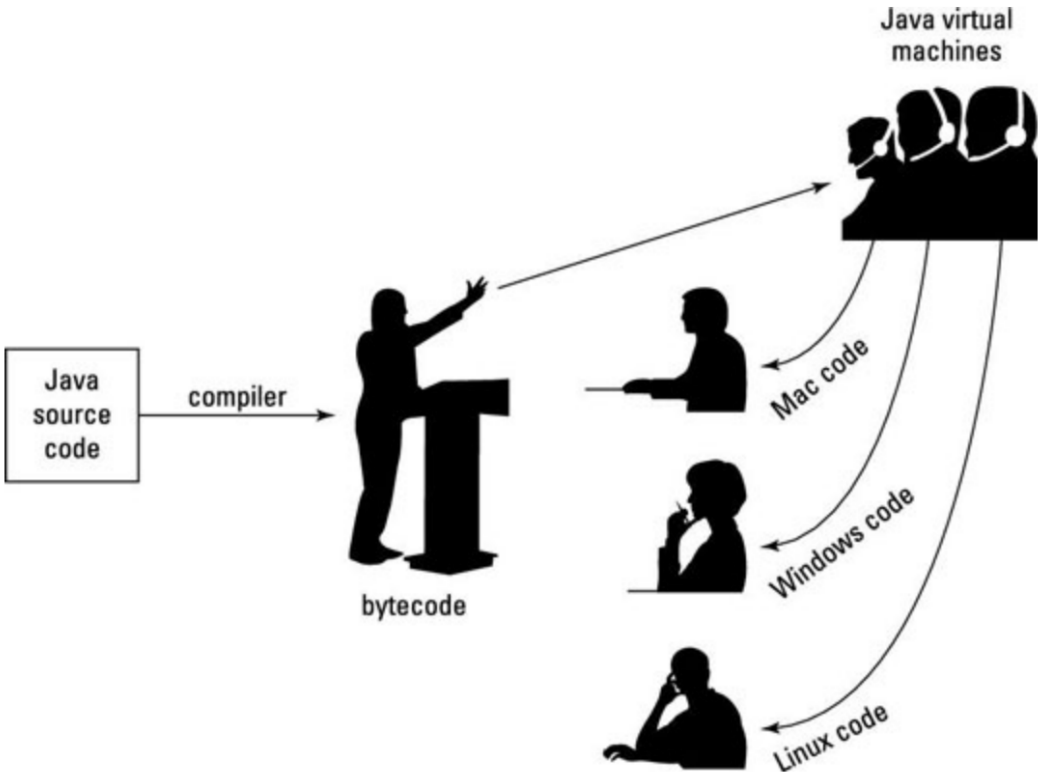
Rôle de JVM



Source: Wiley brand

Introduction de la JVM

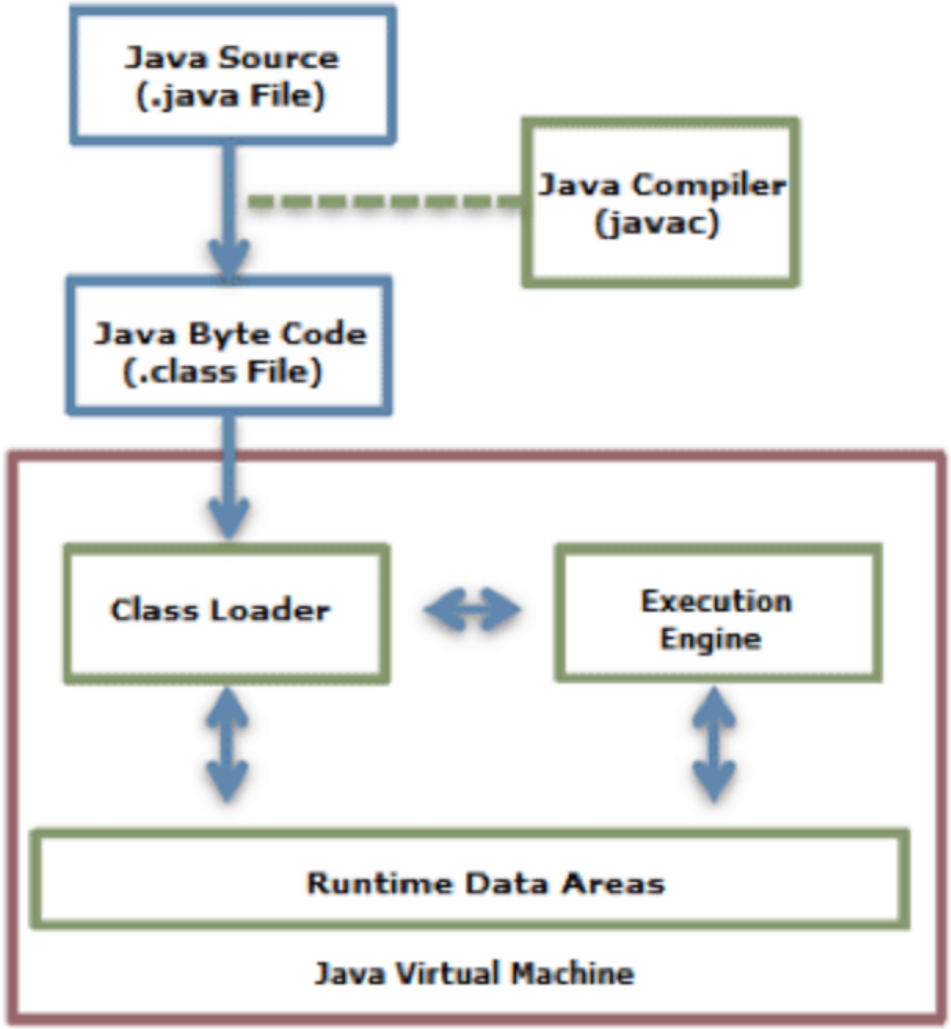
Rôle de JVM



Source: Wiley brand

- l'interprétation du bytecode.
- Interaction avec le système d'exploitation.

Structure de JVM



Bytecode

Le Bytecode est le résultat de la compilation du code sources Java. Il est généré dans des fichiers .class:

- binaire, Non lisible par un humain
- la taille est presque identique à celle du fichier .java, ce qui facilite le transfert et l'exécution de classe depuis le réseau.
- Géré par l'outil Javap de la JVM, aussi appelé le "Décompilateur".
- Executable là où une JVM est installée, indépendamment de la plateforme. D'où la portabilité de Java.

Bytecode

Exemple

.java

```
public void add(String userName) {
    admin.addUser(userName);
}
```

.class

```
2a b4 00 0f 2b b6 00 17 57 b1
```

javap -c

```
public void add(java.lang.String);
  Code:
    0:   aload_0
    1:   getfield        #15; //Field admin:Lcom/nhn/user/UserAdmin;
    4:   aload_1
    5:   invokevirtual   #23; //Method com/nhn/user/UserAdmin.addUser:(Ljava/lang/String;)V
    8:   return
```

Le numéro du byte

opcode
(codé en 1 byte, donc
en total 256 opcodes)

opérande

Bytecode

Exemple

- **aload_0 :** ajoute la variable locale #0 à la pile des opérandes. la variable #0 fait toujours référence à l'instance de classe courante.
- **getfield #15 :** dans le constant pool de la classe courante, elle récupère la #15 et la rajoute dans la pile des opérandes. c'est l'attribut admin qui est récupéré (comme référence).
- **aload_1 :** ajoute la variable locale #1 à la pile des opérandes. C'est le paramétre de méthode; la chaine de caractère userName.
- **invokevirtual #23 :** appelle la méthode correspondant à l'indexe #23 dans le constant pool. La référence rajoutée par getfield et le parametre rajouté par aload_1 sont envoyé à la méthode qu'on appelle. Si la méthode renvoie une valeur non nul, alors elle est empilée dans la pile des opérandes.

L'outil javap

- La décompilation peut se faire à l'aide de l'outil **javap**:

```
javap -c -private Decompilation
```

- -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.

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

Exemple d'erreur

```java
// UserService.java
…
public void add(String userName) {
    admin.addUser(userName);
}
```

```java
// UserAdmin.java - Updated library source code
…
public User addUser(String userName) {
    User user = new User(userName);
    User prevUser = userMap.put(userName, user);
    return prevUser;
}
// UserAdmin.java - Original library source code
…
public void addUser(String userName) {
    User user = new User(userName);
    userMap.put(userName, user);
}
```

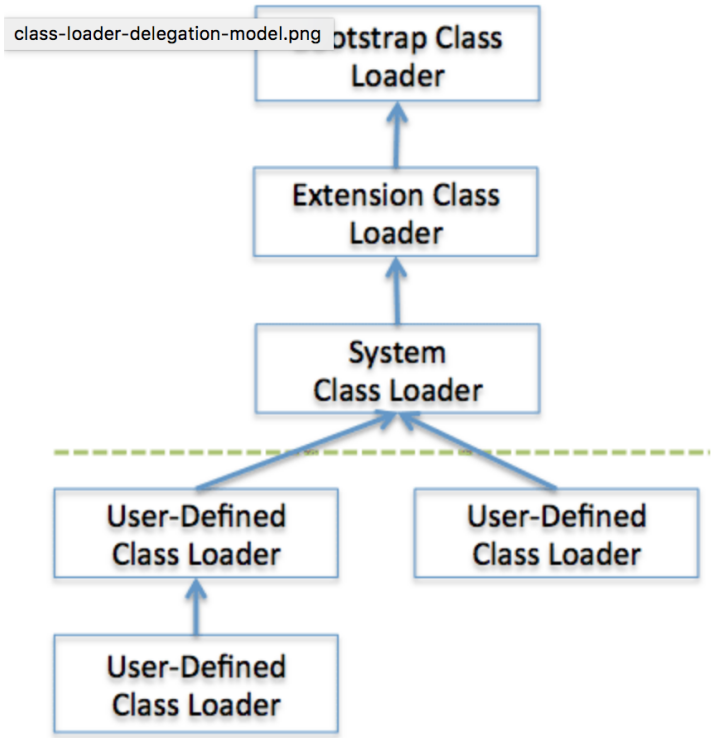
```
Exception in thread "main" java.lang.NoSuchMethodError: com.nhn.user.UserAdmin.addUser(Ljava/lang/String;)V
    at com.nhn.service.UserService.add(UserService.java:14)
    at com.nhn.service.UserService.main(UserService.java:19)
```

Class Loader

- Les Class Loaders permettent de charger des classes depuis le Système de fichiers, aussi depuis de multiples endroits (BD, réseau, etc... sous forme de tableaux d'octets).
- Rôle principale du Class Loader est de convertir un nom de classe en tableau d'octets représentant la classe.
- Le Class Loader est concrètement une hiérarchie de classes qui implémentent la méthode:

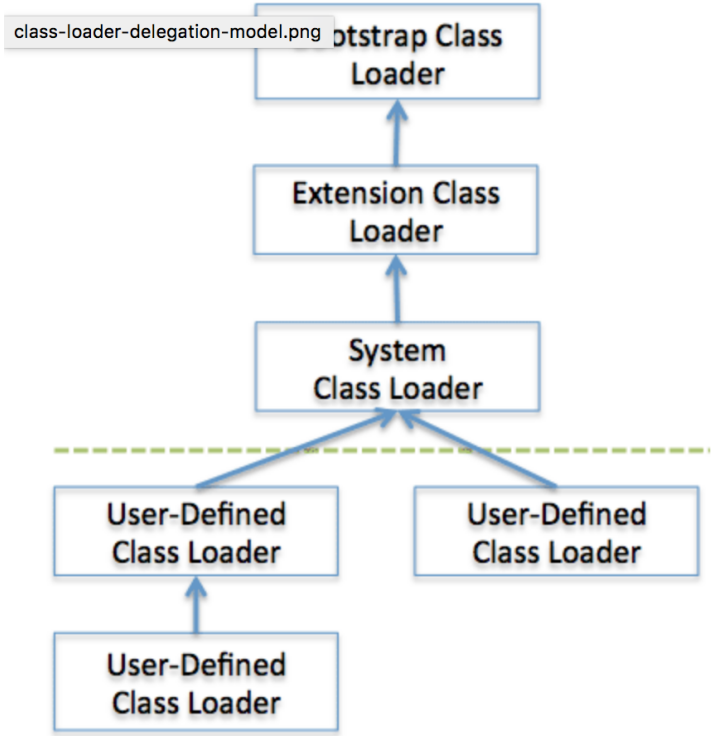
```
Class<?> c = loadClass(String nomClasse, boolean resolveIt);
```

Class Loader



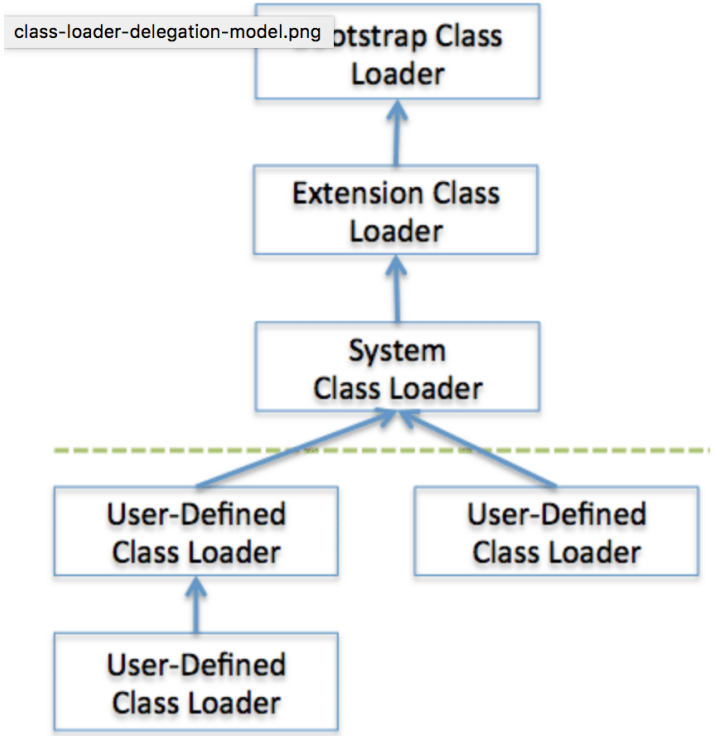
- **Bootstrap Class Loader:** ce CL est créé au démarrage de la JVM. Il s'occupe de charger les classes des APIs standard (dans le JRE), y inclues les classes objets. Contrairement aux autres CLs, celui ci n'est pas implémenté en JAVA.

Class Loader



- **Extension Class Loader:** ce CL s'occupe des classes d'extensions installées et des APIs non standards.

Class Loader



- **System Class Loader:** ce CL s'occupe des classes dans le CLASSPATH.

Class Loader

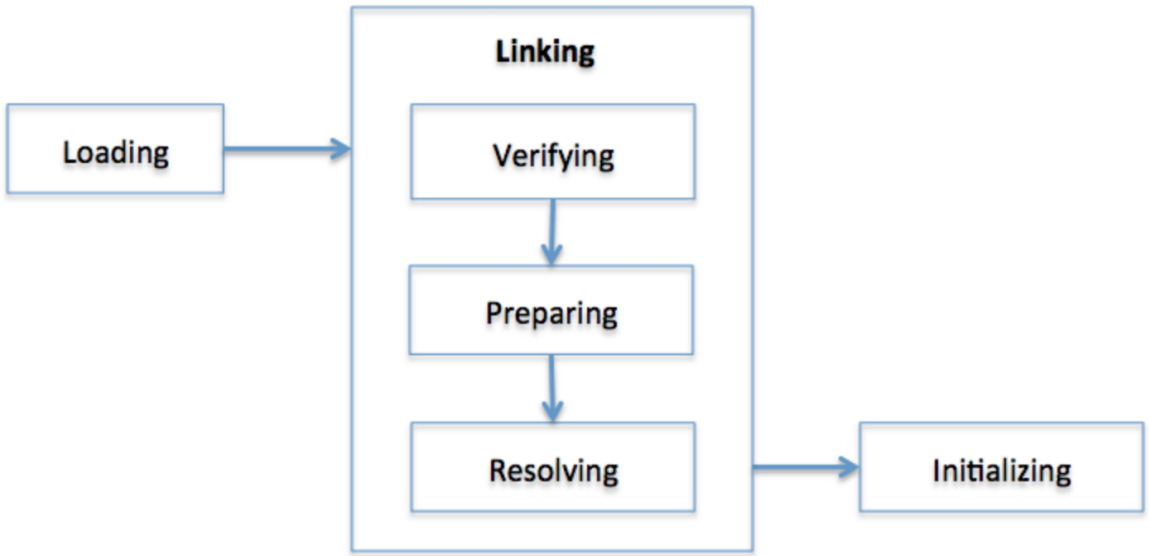
L'implémentation par défaut de la méthode **loadClass(String nomClasse, boolean resolu)** procède premièrement dans l'ordre suivant:

1. Elle appelle la méthode **findLoadedClass(nomClasse)** qui vérifie dans le cache du CL si la classe est déjà chargée;
2. Sinon, elle appelle ensuite la méthode **loadClass** du CL parent;
3. Sinon, elle appelle finalement la méthode **findClass(nomClasse)** pour trouver la cette classe nouvellement invoquée (la stratégie standard est de chercher le fichier nomClasse.class).

Si la classe nomClasse n'est pas trouvée (resolu==0), l'exception **ClassNotFoundException** est envoyée.

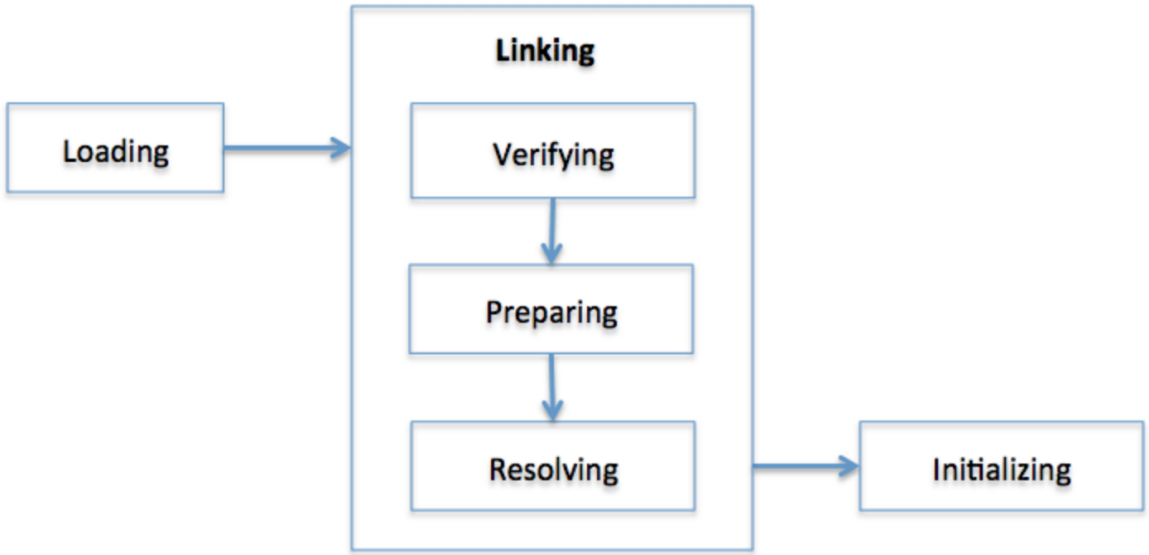
Class Loader

Si **loadClass(String nomClasse, boolean resolu)** trouve une classe non-chargée, alors elle appelle la méthode **resolveClass(Classe)**, qui procède comme suit:



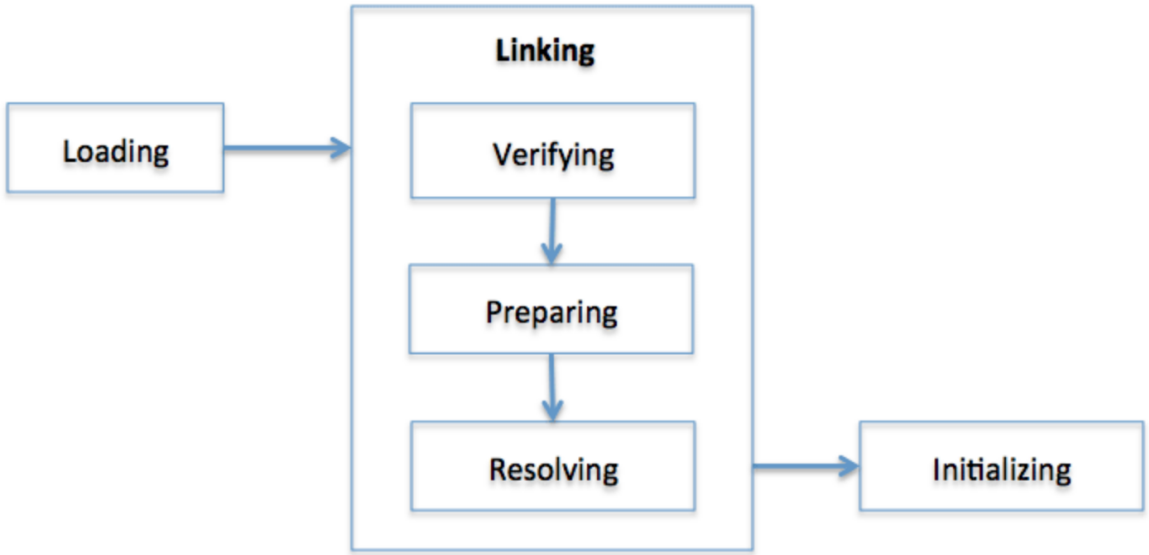
Class Loader

Loading: La classe est obtenue d'un fichier et chargée dans la mémoire de la JVM (Runtime Data Area).



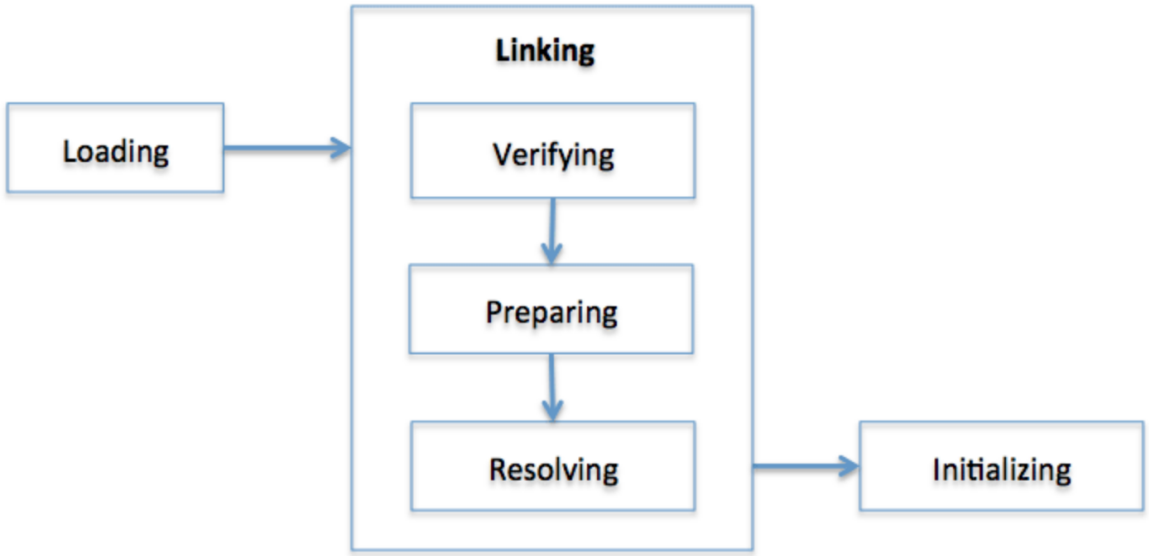
Class Loader

Verifying: Vérifier que le bytecode de la classe est conforme aux spécifications du langage Java et de la JVM. Cette partie est la plus compliquée et la plus longue dans ce process; La majorité des tests du TCK verifie si la JVM détecte bien tous les erreurs évoquées par un chargement d'une classe non-conforme (Code opérations valides, signatures des méthodes...).



Class Loader

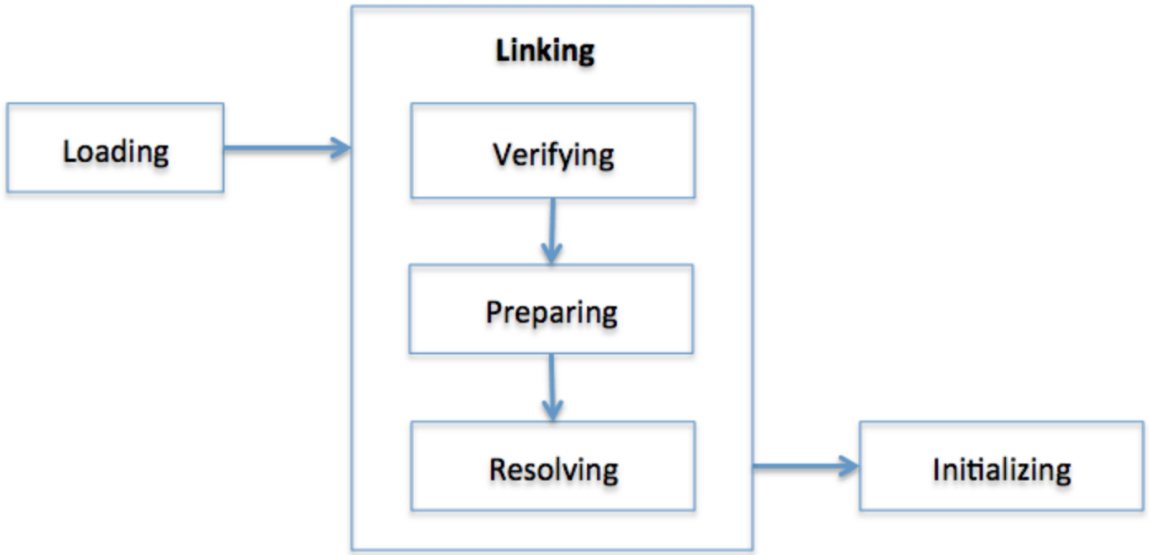
Preparing: Préparer la structure de données qui associe de l'espace mémoire nécessaire pour cette classe, décrivant les attributs, les méthodes et les interfaces définis dedans.



Class Loader

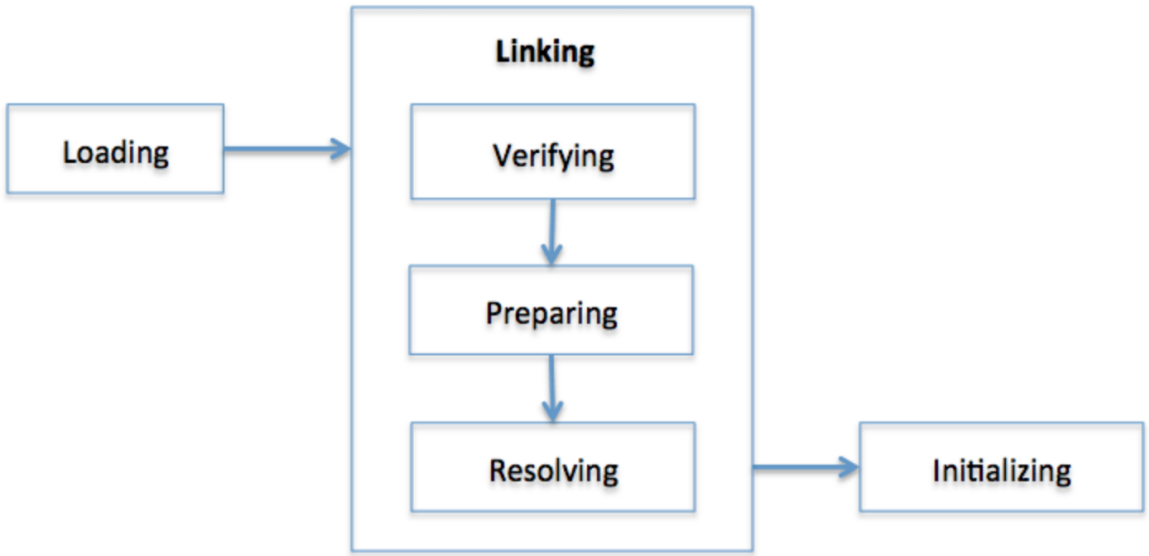
Resolving: Résolution de toutes les références symboliques de la classe par des références directes.

- **IllegalAccessError** (accès à un champs non autorisé)
- **InstantiationError** (instantiation d'une classe abstraite)
- **NoSuchFieldError, NoSuchMethodError, UnsatisfiedLinkError** (méthode native non implémentée)



Class Loader

Initializing: Création des champs statiques et initiliazition à leur valeur par défaut, pas d'exécution de code à ce stade.



Class Loader

CLASSPATH

- C'est une variable, composée de plusieurs chemins désignant des répertoires ou/et des fichiers jar.
- Utilisé par le compilateur et la JVM pour retrouver les classes invoquées par le programme. Ils sont pas autorisés de charger des classes ailleurs dans le disque.
- Pour rajouter un répertoire au CLASSPATH:
  - Configurer la variable système CLASSPATH.

    set CLASSPATH=classpath1;classpath2...
  - Passer le CLASSPATH en paramètre (l'option -classpath) à chaque fois qu'on utilise un outil JDK (typiquement à la compilation et l'exécution).

    sdkTool -classpath classpath1;classpath2...
  - Utiliser l'interface de l'IDE.

Class Loader

JAR

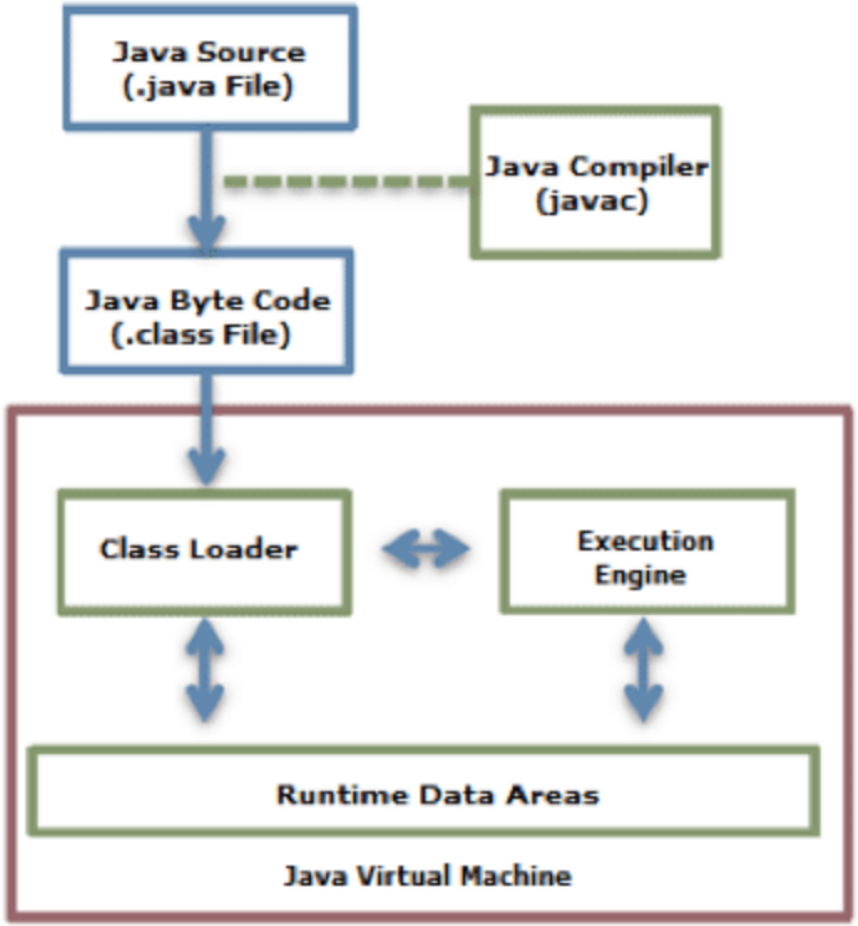
- Un fichier JAR est un type de fichier compressé (similaire aux fichiers ZIP)
- Les fichiers JAR contiennent normalement un certain nombre de classes Java et certaines méta-données (un fichier Manifeste)
- Si le CLASSPATH contient des fichiers jar, les classes sont cherchées et chargées directement depuis l'intérieur de l'archive, la racine de l'arborescence correspondant à la racine de l'archive.

JAR

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`
To invoke an applet packaged as a JAR file	`<applet code=AppletClassName.class archive="JarFileName.jar" width=width height=height> </applet>`

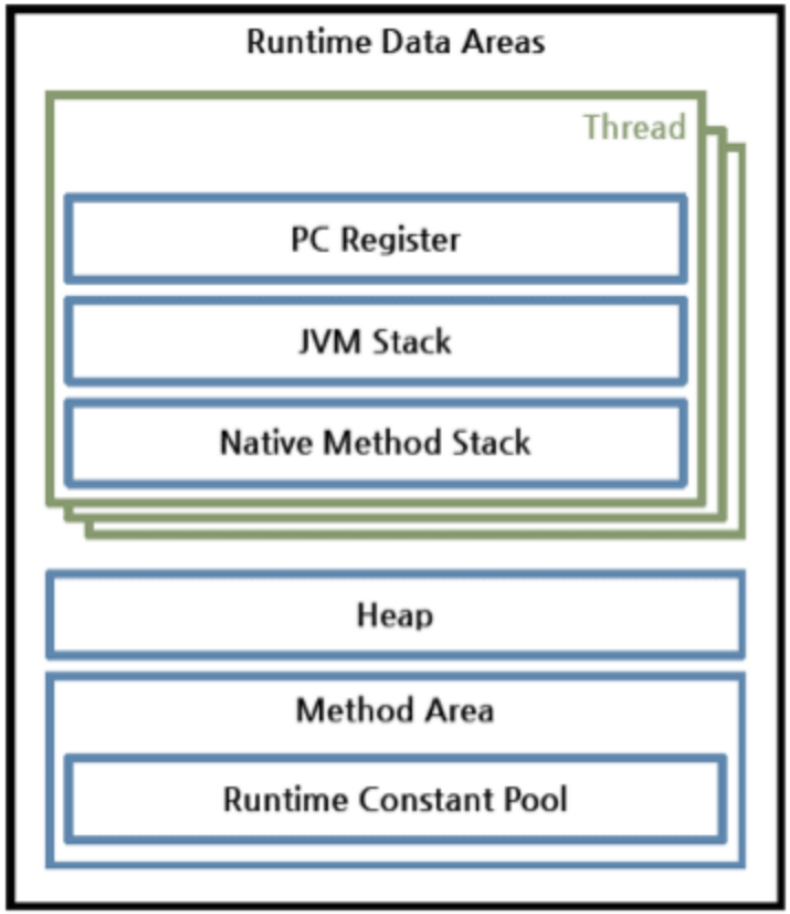
Runtime Data Area

Runtime Data Area est l'espace mémoire assigné à la JVM par l'OS durant son exécution.



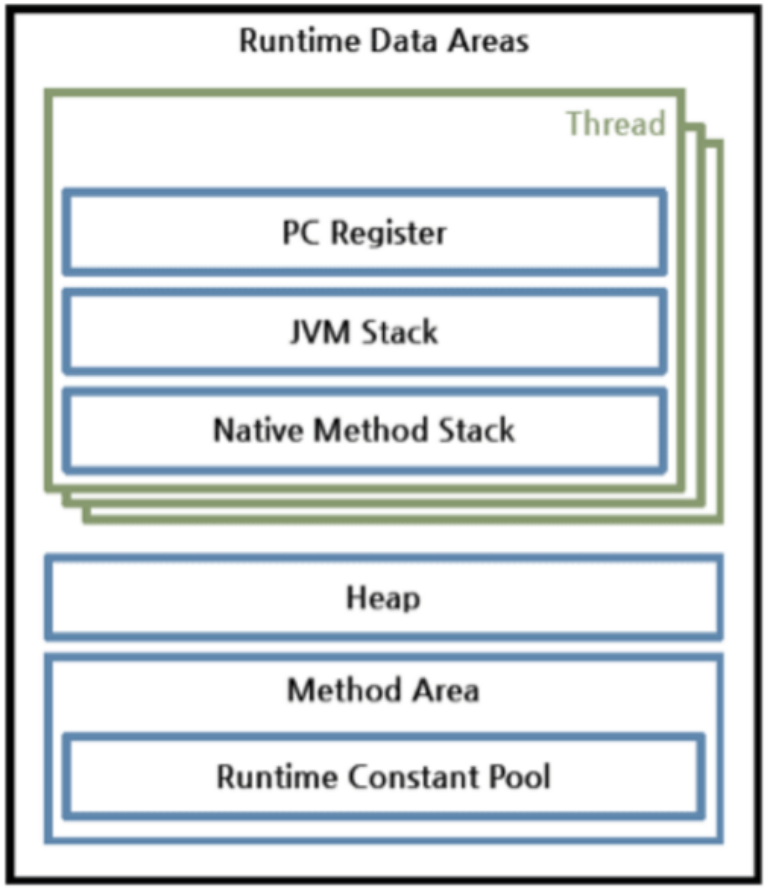
Runtime Data Area

Composants du Runtime Data Area



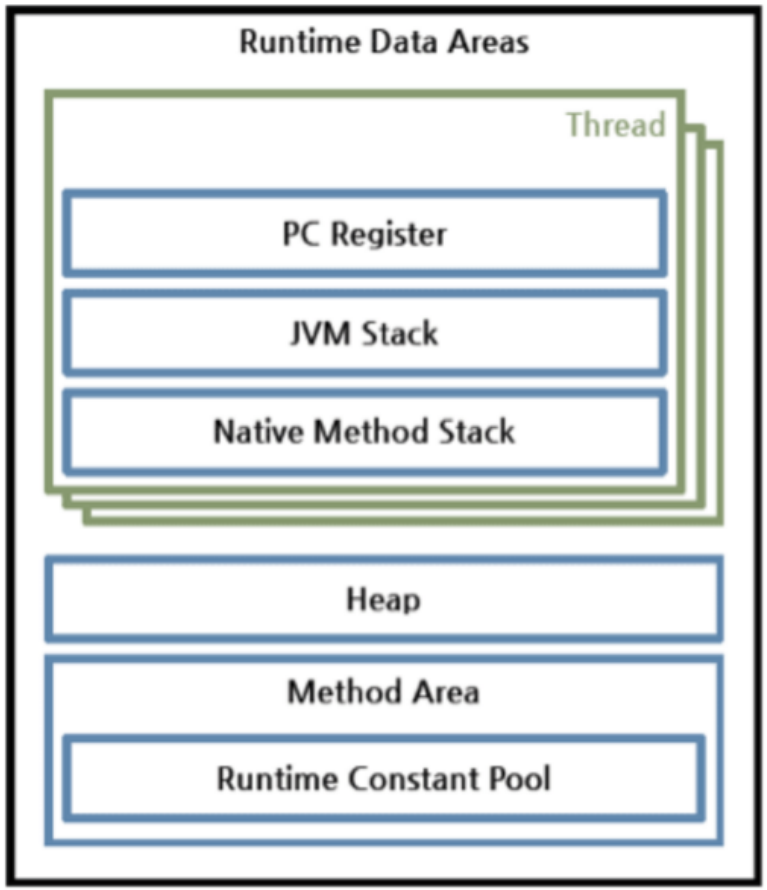
Composants du Runtime Data Area

- **PC (Program Counter) register:** Créé quand le thread commence, il contient l'adresse de l'instruction JVM en cours d'exécution.



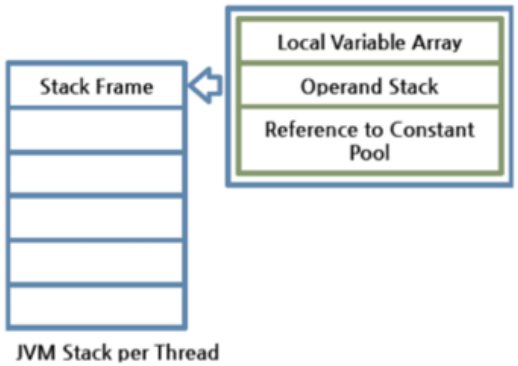
Composants du Runtime Data Area

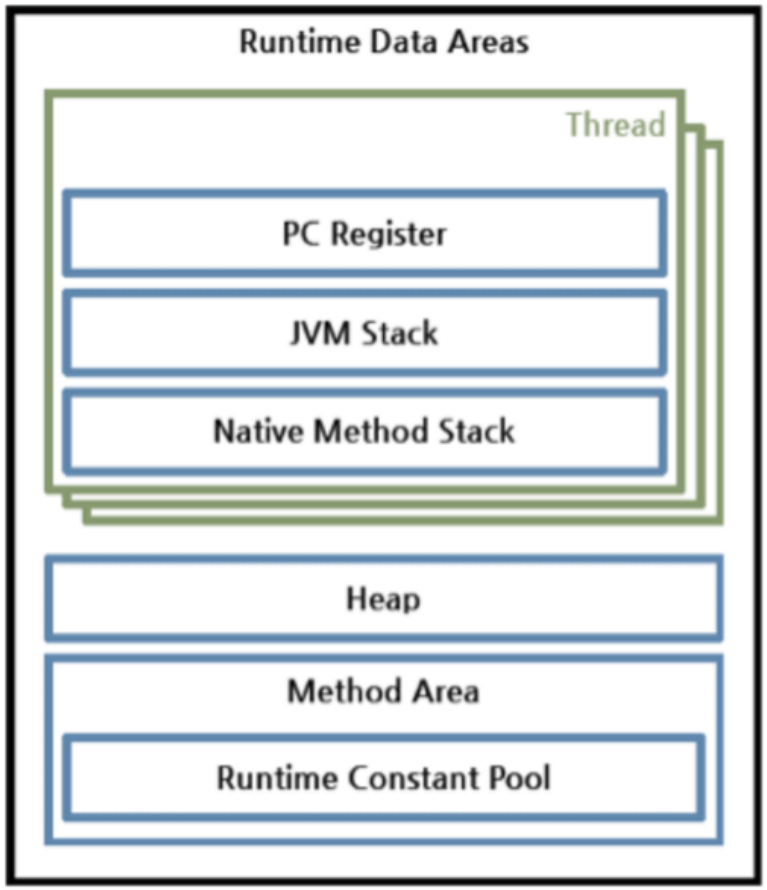
- **PC (Program Counter) register:** Créé quand le thread commence, il contient l'adresse de l'instruction JVM en cours d'exécution.
- **Native method Stack:** La pile utilisée pour exécuter du code native (C/C++) appelé via la JNI (Java Native Interface).



Composants du Runtime Data Area

- **PC (Program Counter) register:** Créé quand le thread commence, il contient l'adresse de l'instruction JVM en cours d'exécution.
- **Native method Stack:** La pile utilisée pour exécuter du code native (C/C++) appelé via la JNI (Java Native Interface).
- **JVM Stack:** Créé quand le thread commence, c'est une pile où la JVM empile et dépile des structures de type Stack Frame. Chaque Stack Frame correspond à une méthode en cours d'exécution dans le thread.

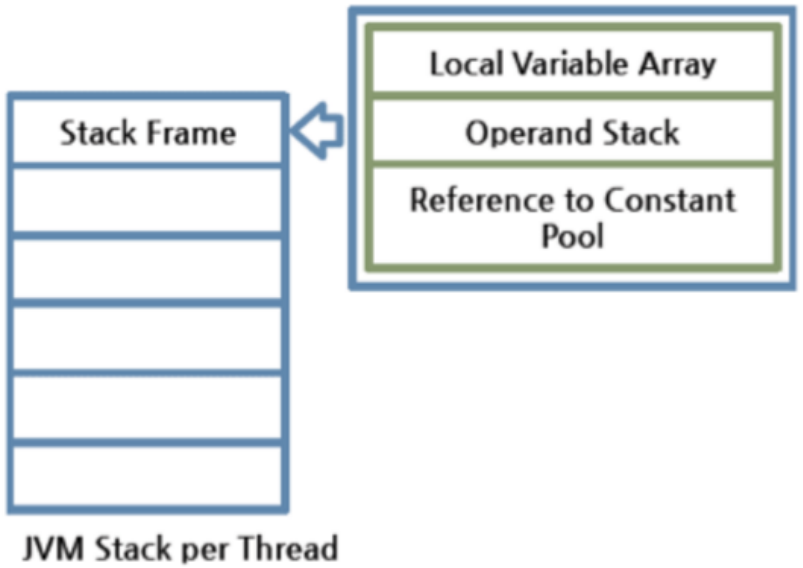




Composants du Runtime Data Area

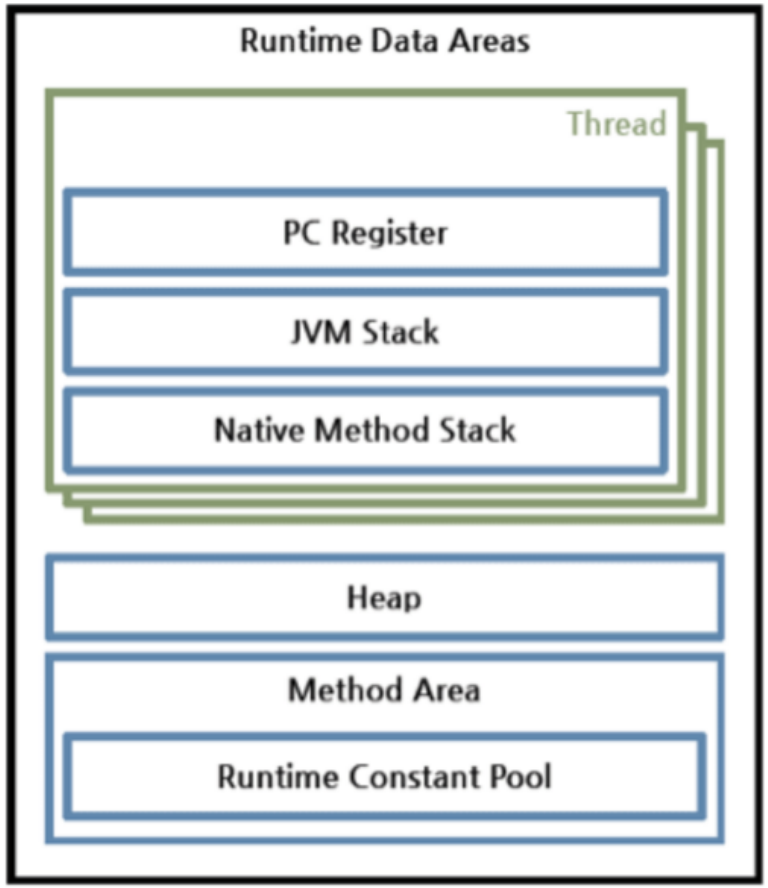
Stack Frame:

- Local Variable Array: index 0 correspond à l'instance à laquelle la méthode appartient. A partir de l'index 1, les paramètres de la méthode sont stockés, ensuite ses variables locales.
- Operand Stack: c'est là où la méthode fait ses "calculs". Pour faire ceci, des données sont échangées entre la piles des opérandes et l'Array des variables locales. C'est également aussi où on empile et dépile les valeurs retournées par les autres méthodes appelées. La taille de cette pile est déterminée durant la compilation.



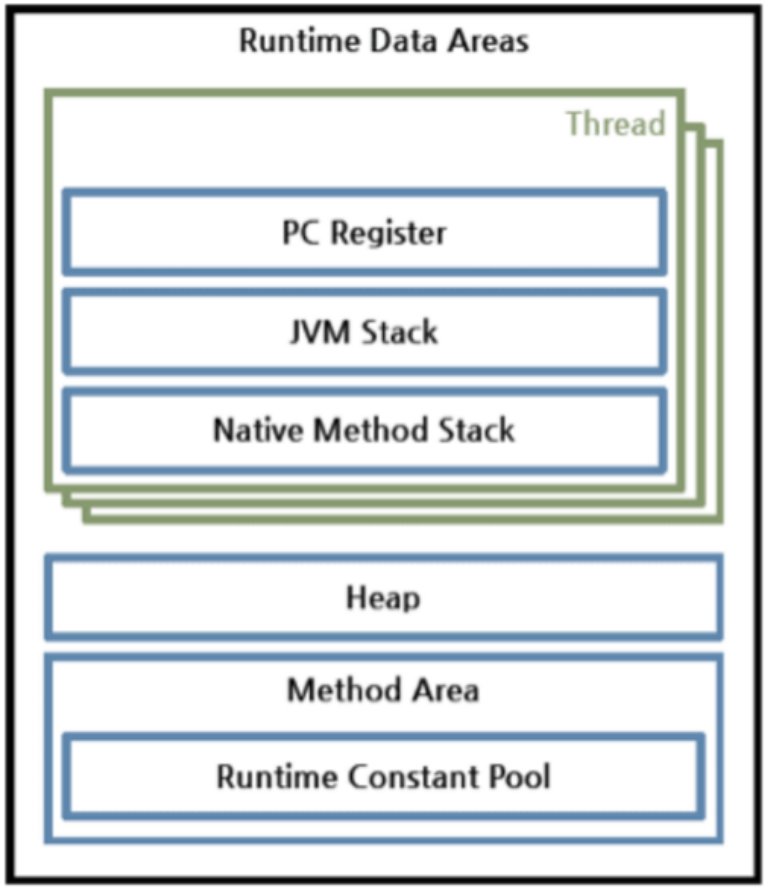
Composants du Runtime Data Area

- **Heap:** L'espace où on stocke des instances ou des objets. C'est là où les Garbage Collectors travaillent le plus.



Composants du Runtime Data Area

- **Heap:** L'espace où on stocke des instances ou des objets. C'est là où les Garbage Collectors travaillent le plus.
- **Method Area:**
  - Runtime Constant Pool qui contient une description des classes: références aux méthodes, attributs, constantes...
  - Description des méthodes, des attributs, variables statiques ainsi que le bytecode des méthodes.



Garbage Collectors

- 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 délocalisation des ressources inutilisées, d'éviter des bugs de double délocalisation, voire même de perdre du temps à désallouer.
- Le principe est de collecter du tas (Heap) du Runtime Data Area les objets qui ne sont plus utilisés/référencés.

Garbage Collectors

- 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.
- Détection des miettes en calculant l'atteignabilité par:
  - Décomptage.
  - Traces.
- Attention à la fragmentation du tas!

Garbage Collectors

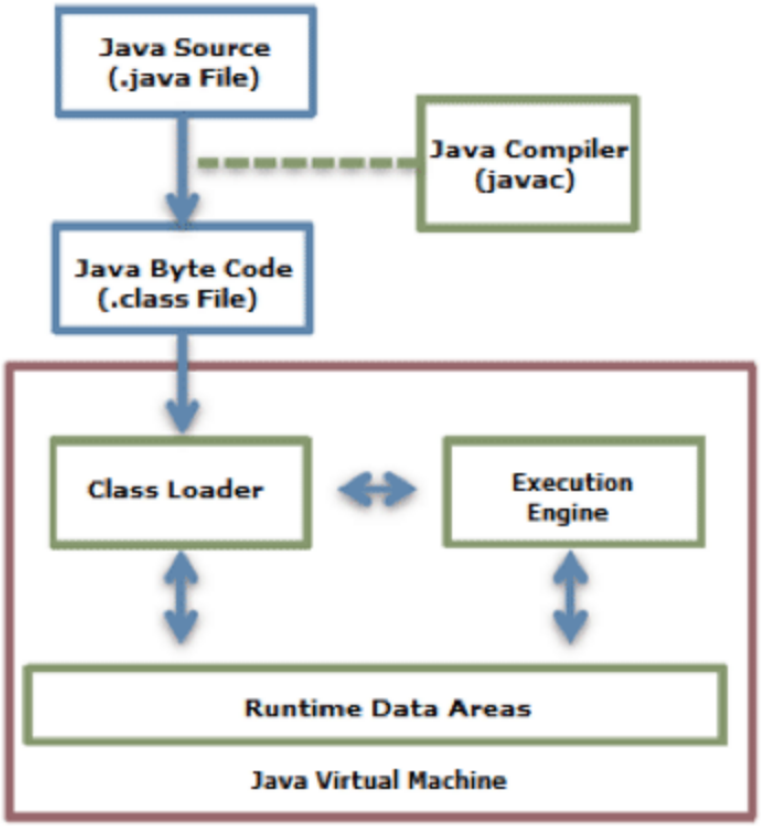
- A la destruction, la JVM appelle la méthode **finalize**, dernière méthode étant appellée:

  protected finalize() throws Throwable ...

- Cela permet à l'instance de libérer les ressources qu'elle utilise. Par exemple, une classe utilisant un fichier temporaire, peut supprimer ce fichier du disque lors de l'appel à la méthode finalize().
- Pour demander explicitement le nettoyage de la mémoire, il faut appeler la méthode System.gc().
- La méthode finalize() est devenu Deprecated à partir de JAVA 9.

Moteur d'exécution

Le moteur d'exécution joue le rôle du CPU de la JVM en exécutant les instructions assignées dans le Runtime Data Area.



Moteur d'exécution

Afin de pré-traiter le bytecode, le moteur d'exécution utilise deux stratégies:

- Interpréteur: lit, interprète puis exécute les instructions bytecode un par un. Malgré que l'interprétation se fait rapidement, l'inconvénient de cette méthode est son temps d'exécution.

Moteur d'exécution

Afin de pré-traiter le bytecode, le moteur d'exécution utilise deux stratégies:

- Interpréteur: lit, interprète puis exécute les instructions bytecode un par un. Malgré que l'interprétation se fait rapidement, l'inconvénient de cette méthode est son temps d'exécution.
- JIT (Just-In-Time) compilateur: Ce compilateur a été introduit afin d'améliorer le temps d'exécution du bytecode. Le moteur d'exécution joue d'abord le rôle d'interpréteur, puis au moment approprié, le JIT compilateur intervient afin de compiler le bytecode en code native et après l'exécuter par le moteur d'exécution sans passer par la phase d'interprétation. Ce type d'exécution est plus rapide (code native compilé est stocké dans le cache), mais la compilation de JIT coûte du temps.

Moteur d'exécution

Afin de pré-traiter le bytecode, le moteur d'exécution utilise deux stratégies:

- Interpréteur: lit, interprète puis exécute les instructions bytecode un par un. Malgré que l'interprétation se fait rapidement, l'inconvénient de cette méthode est son temps d'exécution.
- JIT (Just-In-Time) compilateur: Ce compilateur a été introduit afin d'améliorer le temps d'exécution du bytecode. Le moteur d'exécution joue d'abord le rôle d'interpréteur, puis au moment approprié, le JIT compilateur intervient afin de compiler le bytecode en code native et après l'exécuter par le moteur d'exécution sans passer par la phase d'interprétation. Ce type d'exécution est plus rapide (code native compilé est stocké dans le cache), mais la compilation de JIT coûte du temps.

$\Rightarrow$ Compiler seulement les méthodes fréquentes.