

# Java Virtual Machine - JVM

## 1. Java Virtual Machine là gì?

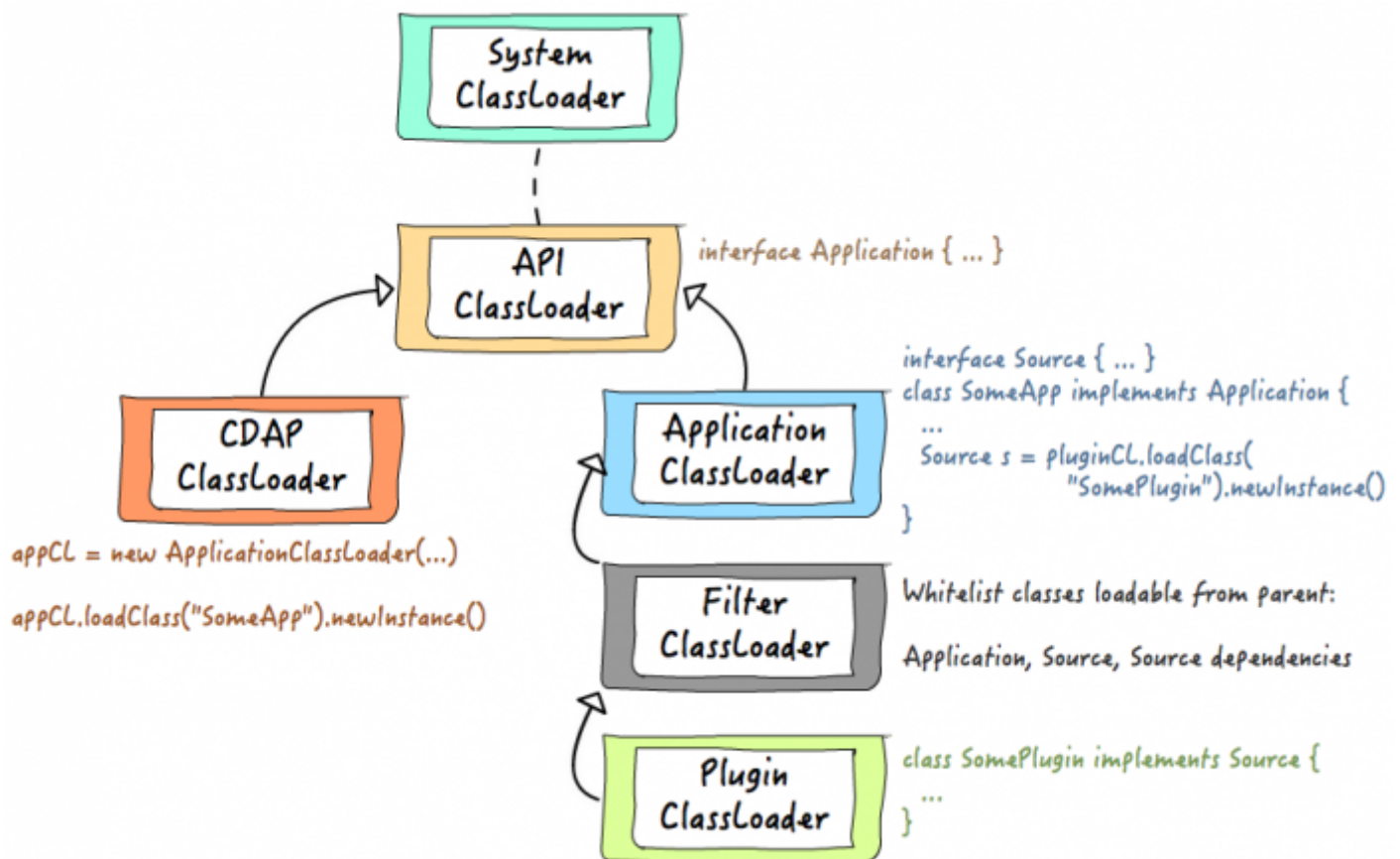
**Java Virtual Machine** (Viết tắt là JVM) là môi trường dùng để chạy ứng dụng được viết bằng ngôn ngữ lập trình Java:

- Một Engine cung cấp Runtime Environment (môi trường chạy mã) để điều khiển mã Java hoặc các ứng dụng.
- Nó chuyển đổi Java Bytecode thành ngôn ngữ máy tính hiểu được.
- JVM là một phần của JRE (Môi trường chạy Java - Java Run Environment). Nó là viết tắt của Java Virtual Machine

Nhờ có JVM mà Java có thể chạy trên nhiều Platform khác nhau. JVM giống như một cái máy ảo, muốn khởi chạy Java thì bắt buộc phải chạy trên cái máy ảo này. Cứ với mỗi Platform ta sẽ có một JVM tương ứng, ví dụ như Ubuntu thì sẽ có bản JVM cho Ubuntu, Windows thì có JVM cho Windows. Và cơ chế hoạt động của JVM ở mọi nền tảng là hoàn toàn như nhau cho nên ứng dụng Java viết trên Window chạy được trên JVM của Window, khi đem cái ứng dụng đó qua Ubuntu thì chỉ cần cài JVM lên Ubuntu là ứng dụng được.

## 2. Các thành phần chính của Java Virtual Machine

- Class Loader: là một hệ thống con của JVM, làm nhiệm vụ tải các lớp được định nghĩa. Nó thực hiện ba chức năng chính: Loading, Linking và Initialization (Khởi tạo)
- Class Area: lưu trữ cấu trúc của các lớp, thuộc tính, phương thức của lớp, và code của các phương thức.
- Heap: là vùng nhớ lưu trữ các đối tượng được khởi tạo trong quá trình thực thi.
- Stack: chứa các frame. Mỗi frame chứa các biến cục bộ và thực thi một hàm gọi và trả kết quả về. Mỗi tiến trình có một Stack riêng, được khởi tạo cùng lúc với tiến trình. Mỗi frame sẽ được tạo khi một hàm được gọi và hủy khi việc gọi hàm kết thúc.
- Programming Counter Register chứa địa chỉ của máy chủ ảo đang thực thi
- Native Method Stack: chứa các hàm của hệ thống được sử dụng trong chương trình
- Execution Engine: là một hệ thống bao gồm: bộ xử lý ảo, trình thông dịch (đọc Java byte code và thực thi các chỉ thị), JIT compiler biên dịch mã byte code sang mã máy. Các nhiệm vụ chính của JVM bao gồm: tải code, kiểm tra code, thực thi code, cung cấp môi trường runtime.



### 3. Bộ nhớ trong Java JVM

Khi thực hiện cấp phát một bộ nhớ hoặc một đối tượng mới có thể được tạo và đặt vào vùng nhớ Heap. Khi ứng dụng của bạn không còn tham chiếu tới đối tượng này nữa thì Java garbage collector cho phép xóa đối tượng này đi để sử dụng lại vùng nhớ đó.

- **Java Heap:** JVM giúp lưu tất cả đối tượng đã được tạo ra bởi toán tử “new” trong ứng dụng Java vào trong vùng nhớ Heap ngay tại thời điểm chạy.

- **Java Stack:** Các phương thức và tham chiếu tới đối tượng địa phương được lưu trữ trong Stack. Mỗi Thread sẽ được quản lý một stack. Khi phương thức được gọi, nó được đưa vào đỉnh của Stack. Stack lưu trữ trạng thái của phương thức bao gồm: dòng code thực thi, tham chiếu tới đối tượng địa phương. Khi phương thức chạy xong, vùng nhớ (dòng code thực thi, tham chiếu tới đối tượng địa phương) được đẩy ra khỏi stack và tự động giải phóng.

- **Java Perm:** Lưu trữ thông tin của Class được nạp vào và một vài tính năng khác như StringPool (vùng nhớ của biến String) thường được tạo bởi phương thức String.intern(). Khi ứng dụng của bạn chạy, Perm space được lấp đầy nhanh chóng.

### 4. Cơ chế làm việc của Java Virtual Machine

JVM được chia thành 3 module chính:

- Class-Loader Subsystem: chuyên tìm kiếm và load các file .class vào vùng nhớ của Java.
- Runtime Data Area: vùng nhớ hệ thống cấp phát cho Java Virtual Machine.

- Execution Engine: chuyển các lệnh của JVM trong file .class thành các lệnh của máy, hệ điều hành tương ứng và thực thi chúng.

## 4.1. Class Loader Subsystem

Chịu trách nhiệm load, liên kết và khởi tạo file .class khi nó refer đến một class lần đầu tiên trong thời gian chạy (không phải thời gian biên dịch).

### - Loading:

Các class sẽ được load bởi thành phần này. BootStrap class Loader, Extension class Loader, và Application class Loader là 3 trình nạp class sẽ giúp thực hiện được điều đó.

- **Boot Strap ClassLoader** : chịu trách nhiệm load các class từ classpath, đó là các file **.jar**. Ưu tiên cao nhất sẽ được trao cho trình nạp class này.
- **Extension ClassLoader** : Chịu trách nhiệm load các class nằm trong thư mục **jre\lib**
- **Application ClassLoader** : Chịu trách nhiệm load các class từ được cấu hình ở các đường dẫn (path) được đề cập trong biến môi trường.

Các trình nạp class bên trên tuân theo thuật toán phân cấp trong khi load các class.

### - Linking:

- **Verify** : Bytecode verifier (trình xác minh bytecode) sẽ kiểm tra xem byte code có được tạo ra phù hợp hay không. Nếu xác minh là không thành công, sẽ thông báo lỗi verify.
- **Prepare** : Đối với tất cả các static variables memory sẽ được phân bổ và gán với các giá trị mặc định.
- **Resolve** : All symbolic memory references are replaced with the original references from Method Area.

### - Initialization:

Đây là giai đoạn cuối của Class Loading. Trong giai đoạn này các biến tĩnh (static variables) sẽ được gán với các giá trị ban đầu và static block sẽ được thực thi.

## 4.2. Runtime Data Area

Runtime Data Area được chia thành 5 thành phần chính:

- **Method area** : Tất cả dữ liệu level class được lưu trữ ở đây, bao gồm static variables
- **Heap area** : Tất cả các đối tượng và các biến, arrays sẽ được lưu trữ ở đây. Method area và Heap area chia sẻ memory cho nhiều luồng, do đó dữ liệu được lưu trữ trở nên không an toàn.
- **Stack area** : Đối với mỗi luồng, một separate runtime stack sẽ được tạo. Khi mỗi phương thức được gọi, mỗi lời gọi sẽ được thực hiện trong stack memory gọi là Stack Frame. Tất cả local variables sẽ được tạo trong stack memory. Stack area là thread-safe, vì nó không chia sẻ resource. Stack Frame được chia làm 3 phần nhỏ:

- **PC Registers** : Với mỗi luồng sẽ được chia vào PC Register riêng để giữ địa chỉ của lệnh thực thi hiện tại sau khi lệnh được thực thi. Sau đó PC Register sẽ cập nhật lệnh tiếp theo.
- **Native Method stacks** : Đối với mỗi luồng, một ngăn xếp phương thức riêng sẽ được tạo.

### 4.3. Execution Engine (Công cụ thực thi)

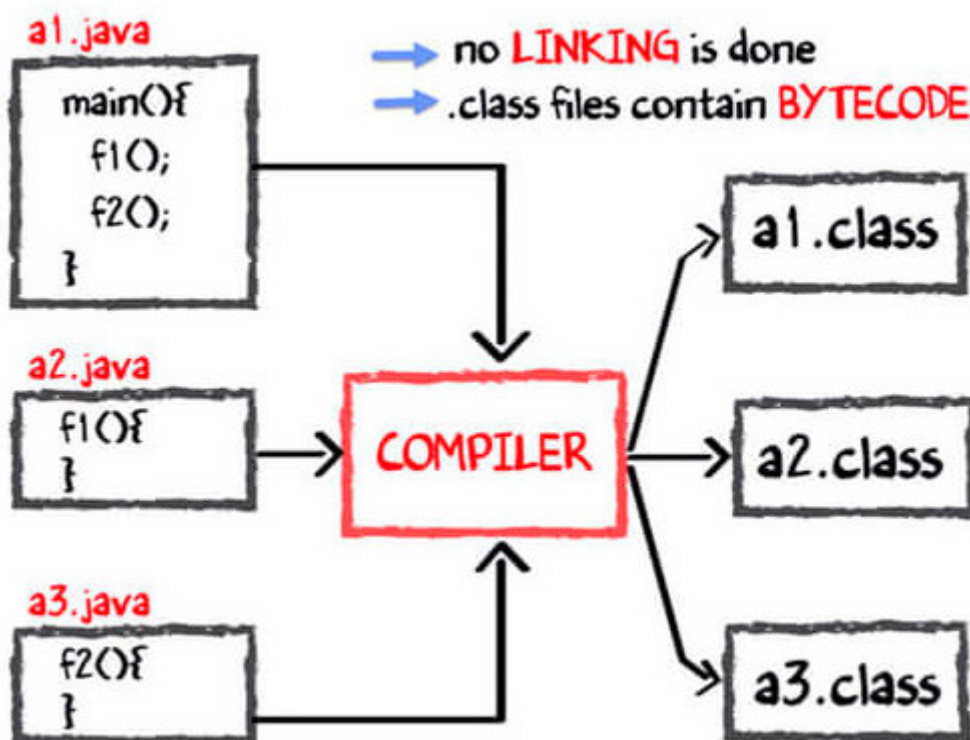
Bytecode sẽ được assign cho Runtime Data Area, và sẽ được thực thi bởi Execution Engine. Execution Engine đọc bytecode từng mảng một.

- **Interpreter** : Trình thông dịch bytecode nhanh, nhưng thực thi chậm. Nhược điểm của trình thông dịch là khi một phương thức được gọi nhiều lần, mỗi lần cần một thông dịch mới.
- **JIT Compiler** : Trình biên dịch JIT vô hiệu hóa nhược điểm của trình thông dịch (interpreter). Execution Engine sẽ sử dụng trợ giúp của trình thông dịch trong việc chuyển đổi bytecode, nhưng khi thấy mã lặp lại, nó sẽ sử dụng trình biên dịch JIT. Nó biên dịch toàn bộ bytecode và thay đổi nó thành mã gốc. Mã này sẽ được sử dụng khi các phương thức bị gọi lặp lại nhiều lần, điều này giúp cải thiện hiệu năng của hệ thống.
- **Garbage Collector** : Thu thập và loại bỏ các đối tượng được khởi tạo nhưng không sử dụng.

## 5. Biên dịch và thực thi mã Java trong Java VM

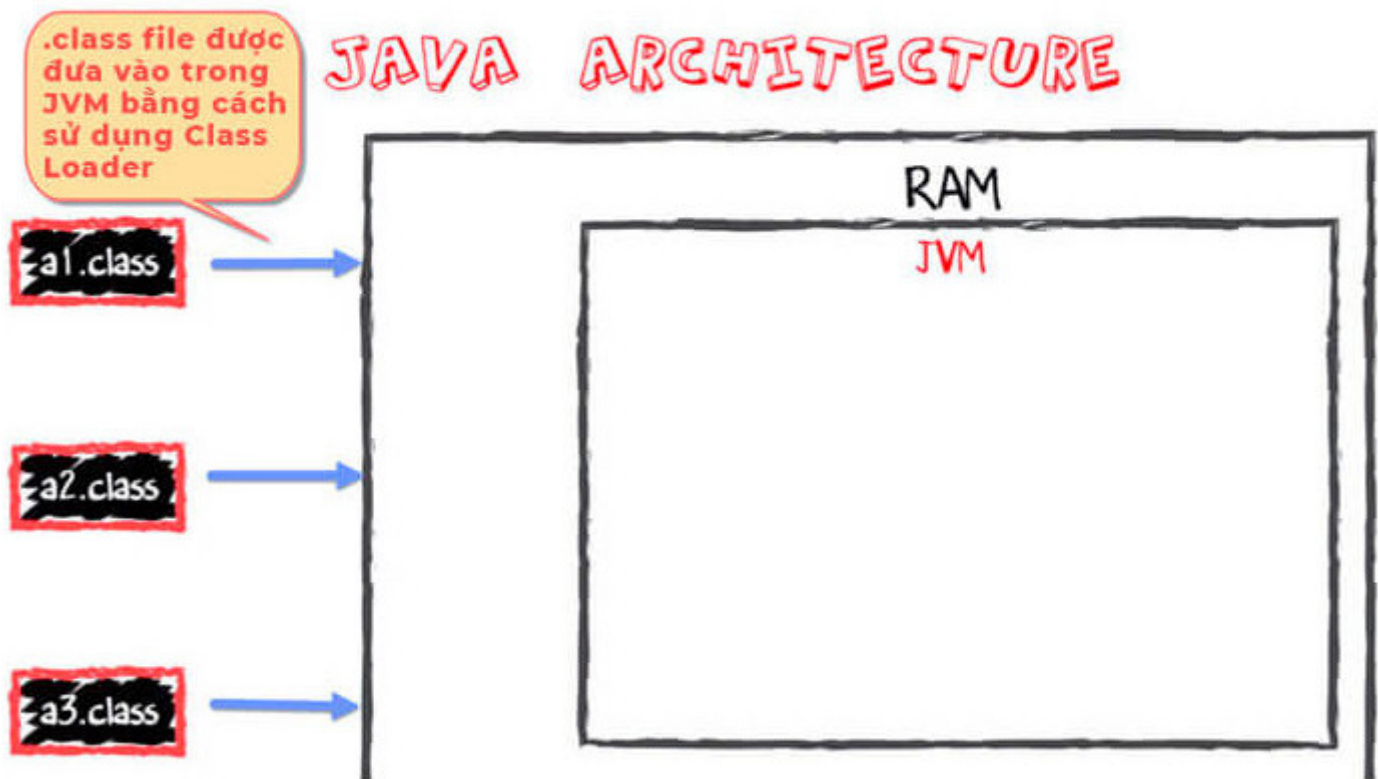
Hãy xem xét quá trình này cho JAVA. Trong main của bạn, bạn có hai method f1 và f2.

- Phương thức main được lưu trữ trong tệp a1.java
- f1 được lưu trữ trong tệp dưới dạng a2.java
- f2 được lưu trữ trong tệp dưới dạng a3.java



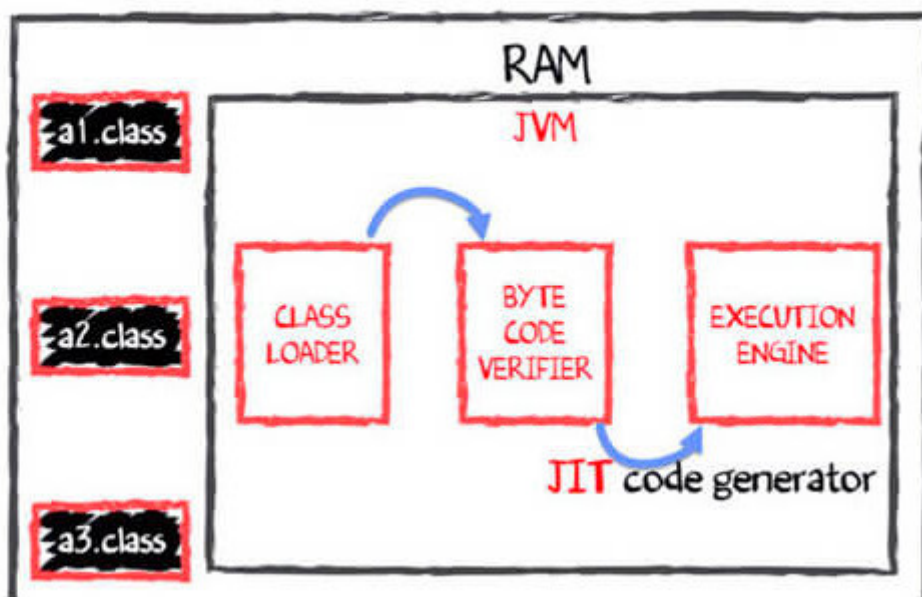
Trình biên dịch sẽ biên dịch ba tệp và tạo ra 3 tệp .class tương ứng chứa BYTE code. Không giống như C, không có liên kết được thực hiện.

Java VM hoặc Máy ảo Java nằm trên RAM. Trong quá trình thực thi, sử dụng class loader, class files được đưa vào RAM. Tại đây BYTE code được xác minh cho tính bảo mật.



Tiếp theo, Execution Engine sẽ chuyển đổi bytecode thành mã máy gốc. Đây chỉ là trong thời gian biên dịch. Đây là một trong những lý do chính tại sao Java tương đối chậm.

**JIT** converts **BYTECODE** into machine code



Revision #1

Created 2 November 2019 18:32:55 by Laptrinh.vn

Updated 2 November 2019 18:49:18 by Laptrinh.vn