Java Guideline

Kiến thức Java

- Java Heap memory
- Java Garbage collection - GC
- Java Virtual Machine - JVM
- Java Stack memory
- Java Heap - Stack memory
- Catch an SQL exception with the error code ORA-00942 in Java

Java Heap memory

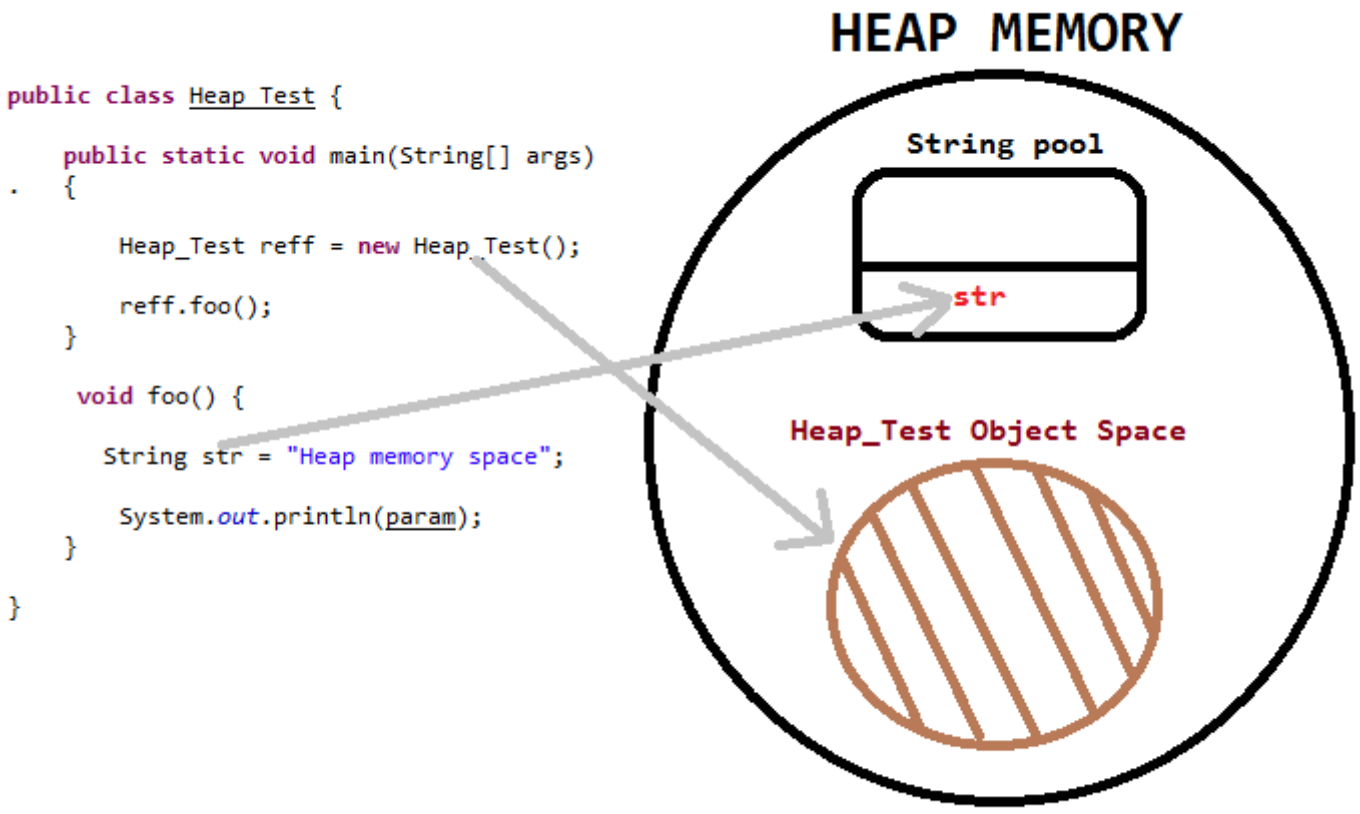
1. Java Heap là gì

Khi JVM start up, nó được cấp phát bộ nhớ Memory từ Hệ điều hành. JVM sử dụng memory cho tất cả nhu cầu của nó và 1 phần bộ nhớ này được gọi là Heap memory. Heap trong java được sinh ra ở cuối cùng của vùng địa chỉ và di chuyển lên trên, bất cứ khi nào chúng ta sử dụng toán tử `new` hoặc khai báo đối tượng sẽ được cấp phát bộ nhớ từ Heap, còn khi object chết hoặc GC, memory sẽ quay trở lại Heap.

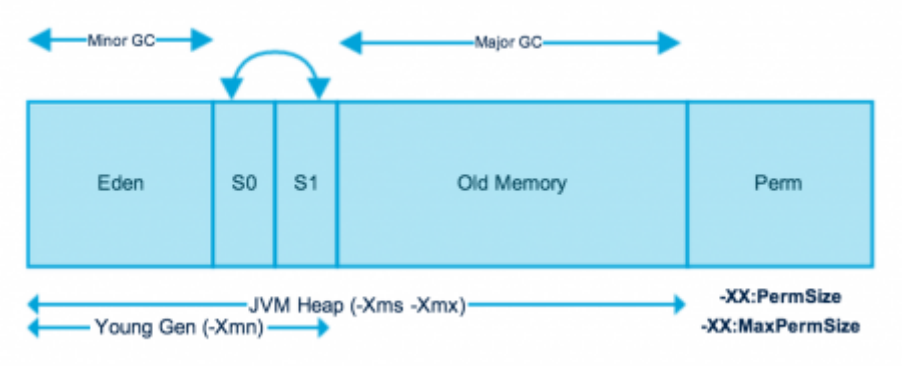
Bộ nhớ Heap trong Java được dùng để cấp phát bộ nhớ cho các đối tượng, các lớp JRE lúc thực thi. Bất cứ khi nào, chúng ta tạo đối tượng, nó sẽ được tạo trong bộ nhớ Heap.

Với những đối tượng không còn được tham chiếu nữa thì trình thu thập rác (Garbage Collection) sẽ giải phóng bộ nhớ mà các đối tượng đó sử dụng.

Đối tượng được tạo trong bộ nhớ Heap có phạm vi truy cập toàn cục, tức là chúng ta có thể truy cập đối tượng đó ở bất kỳ đâu trong ứng dụng



2. Java Heap memory



Heap đôi khi được chia làm 2 vùng (hay thế hệ) gọi là **Nursey** (hay `young space`) và vùng `old space`.

Nursery là một phần của *Heap* để dành cho việc cấp phát cho các đối tượng mới. Khi *nursery* bắt đầu đầy, rác sẽ được thu gom bằng một *process* thu gom rác đặc biệt gọi là `young collection`, nơi mà các đối tượng sống đủ lâu trong *nursery* được `promoted` và di chuyển lên vùng `old space`, do đó giải phóng *nursery* để cấp phát các đối tượng khác.

Khi `old space` trở nên đầy, rác sẽ được thu gom bởi process khác được gọi là `old collection`. Lý do đằng sau 1 `nursery` là hầu hết các đối tượng đều là tạm thời và ngắn hạn. Một process `young collection` được thiết kế để nhanh chóng tìm các đối tượng mới được cấp phát mà vẫn tồn tại (*alive*) và di chuyển chúng khỏi *nursery*. Thông thường, `young collection` process sẽ giải phóng một số lượng bộ nhớ nhất định nhanh hơn `old collection` process hay `garbage collection` process của một *heap* mà không có vùng *nursery* (còn gọi là `single-generational heap`)

Từ các version release mới của JVM, có một phần của *nursery* được giữ lại gọi là `keep area`. Vùng này sẽ chứa các đối tượng được cấp phát gần đây nhất và chưa được thu gom cho đến khi `young collection` process tiếp theo được chạy. Điều này sẽ ngăn cản việc đối tượng được di chuyển lên `old space` vì các đối tượng chỉ vừa được cấp phát ngay trước khi `young collection` chạy.

`Permanent generation` của heap được sử dụng để lưu trữ String pool và siêu dữ liệu (metadata) khác nhau theo yêu cầu của JVM liên quan đến Class, phương thức và các java primitives khác.

3. Phân loại lưu trữ Object trong Heap

Trong quá trình cấp phát đối tượng, `JVM` sẽ phân biệt giữa các đối tượng nhỏ (*small object*) và lớn (*large object*). Giới hạn khi một đối tượng được xem là lớn phụ thuộc vào các yếu tố như JVM version, độ lớn *heap size*, chiến lược GC và `platform` được sử dụng, nhưng thường sự khác nhau giữa 2 loại đối tượng là khoảng **128kB**.

Các đối tượng nhỏ sẽ được cấp phát trong vùng gọi là `thread local areas` (**TLAs**), **TLAs** là các khối rỗng (*free chunks*) từ *heap* và được giao cho một `Java thread` sử dụng độc quyền. *Thread* này có thể cấp phát các đối tượng trong **TLA** của nó mà không cần phải đồng bộ với các *thread* khác. Khi **TLA** bắt đầu đầy, *thread* chỉ đơn giản yêu cầu một **TLA** mới.

Các đối tượng lớn sẽ không vừa bên trong một **TLA** được cấp phát trực tiếp trên *heap*. Khi một *Nursery* được dùng, các đối tượng lớn sẽ được phân bố trực tiếp trên `old space`. Sự phân bố các đối

tượng lớn yêu cầu sự đồng bộ giữa các `Java thread`, mặc dù JVM dùng một hệ thống `caches` của các khối rỗng khác kích thước để giảm nhu cầu đồng bộ và tăng tốc độ cấp phát.

4. Tăng dung lượng Heap như thế nào?

Bằng cách thêm tham số khi chạy ứng dụng, chúng ta có thể tăng/giảm dung lượng bộ nhớ Heap:

```
java -Xmx120m -Xms30m -Xmn10m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar
Java2Demo.jar
```

Trong đó:

- Xms: Dung lượng bộ nhớ được cấp phát khi JVM được start up
- Xmx: Dung lượng bộ nhớ tối đa mà JVM được cấp phát
- Xmn: Kích thước của không gian heap Young generation.
- XX:PermSize, XX:MaxPermSize: Dung lượng bộ nhớ cấp phát và bộ nhớ tối đa của Permanent generation

Kích thước mặc định của không gian Heap trong Java là 128MB trên hầu hết JVM của 32 bit nhưng nó rất khác nhau từ JVM đến JVM.

Ví dụ: Mặc định tối đa và kích thước heap bắt đầu cho Hệ điều hành Solaris 32 bit (Phiên bản nền tảng SPARC) là -Xms = 3670K và -Xmx = 64M và giá trị mặc định của các tham số kích thước heap trên hệ thống 64 bit đã tăng lên khoảng 30%.

Ngoài ra, nếu bạn đang sử dụng trình thu gom rác thông lượng trong Java 1.5 kích thước heap tối đa mặc định của JVM sẽ là bộ nhớ vật lý / 4 và kích thước vùng heap ban đầu mặc định sẽ là bộ nhớ vật lý / 16. Một cách khác để tìm kích thước heap mặc định của JVM là khởi động một ứng dụng với các tham số heap mặc định và theo dõi bằng cách sử dụng JConsole có sẵn trên JDK 1.5 trở đi, trên tab VMSummary, bạn sẽ có thể thấy kích thước heap tối đa.

Nhân tiện, có thể tăng kích thước không gian heap java dựa trên nhu cầu ứng dụng của bạn và tôi luôn khuyến nghị điều này để tránh sử dụng các giá trị heap JVM mặc định, nếu ứng dụng của bạn lớn và nhiều đối tượng được tạo, bạn có thể thay đổi kích thước của vùng heap bằng cách sử dụng các tùy chọn JVM -Xms và -Xmx. Xms biểu thị kích thước bắt đầu của Heap trong khi -Xmx biểu thị kích thước tối đa của Heap trong Java.

Có một tham số khác gọi là -Xmn biểu thị Kích thước của thế hệ Heap Java mới. Chỉ có điều là bạn không thể thay đổi kích thước của Heap trong Java một cách linh hoạt, bạn chỉ có thể cung cấp tham số Kích thước Heap Java trong khi bắt đầu JVM. Tôi đã chia sẻ một số tùy chọn JVM hữu ích hơn liên quan đến không gian Heap Java và bộ sưu tập Rác trên bài đăng của tôi 10 tùy chọn JVM mà lập trình viên Java phải biết, bạn có thể thấy hữu ích.

10 điểm về không gian heap Java

1. Bộ nhớ Heap Java là một phần của bộ nhớ được hệ điều hành cấp cho JVM.

2. Bất cứ khi nào chúng ta tạo các đối tượng, chúng được tạo bên trong Heap của Java.

3. Không gian Heap Java được chia thành ba vùng hoặc thế hệ cho mục đích thu gom rác được gọi là Thế hệ mới (New Generation), Thế hệ cũ (Old Generation) hoặc Thế hệ cho thuê(Rental Generation) hoặc Không gian Perm (Perm Space). Tạo vĩnh viễn là rác được thu thập trong toàn bộ máy chủ trong điểm nóng JVM.

4. Bạn có thể tăng hoặc thay đổi kích thước của không gian Heap Java bằng cách sử dụng các tùy chọn dòng lệnh JVM -Xms, -Xmx và -Xmn. Đừng quên thêm từ "M" hoặc "G" sau khi chỉ định kích thước để biểu thị Mega hoặc Gig. Ví dụ: bạn có thể đặt kích thước java heap thành 258MB bằng cách thực hiện theo lệnh java -Xmx256m HelloWord.

5. Bạn có thể sử dụng JConsole hoặc Runtime.maxMemory (), Runtime.totalMemory (), Runtime.freeMemory () để truy vấn các kích thước lập trình Heap trong Java.

6. Bạn có thể sử dụng lệnh "jmap" để lấy kết xuất Heap trong Java và "jhat" để phân tích kết xuất heap đó.

7. Không gian Heap Java khác với Stack được sử dụng để lưu trữ phân cấp cuộc gọi và các biến cục bộ.

8. Trình thu gom rác Java chịu trách nhiệm khôi phục bộ nhớ từ các đối tượng chết và trở về không gian Heap Java.

9. Nếu bạn gặp lỗi java.lang.OutOfMemoryError, đôi khi đó chỉ là vấn đề tăng kích thước heap, nhưng nếu nó bị lặp lại, hãy tìm nguyên nhân gây rò rỉ bộ nhớ trong Java.

10. Sử dụng các công cụ Phân tích kết xuất Profiler và Heap để hiểu không gian Heap Java và dung lượng bộ nhớ được phân bổ cho từng đối tượng.

Java Garbage collection - GC

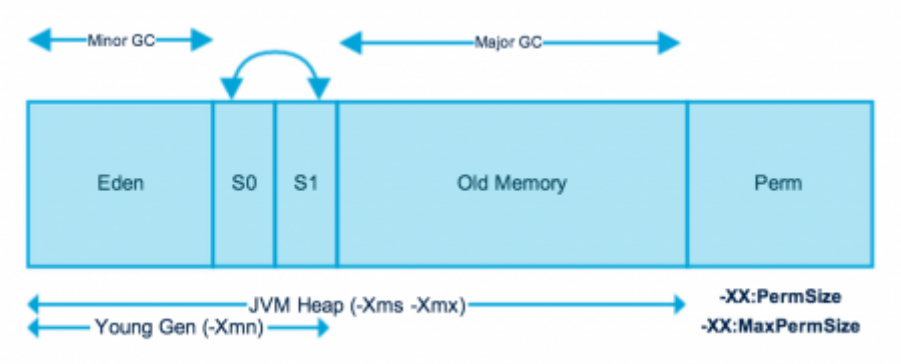
1. Garbage collection là gì?

Garbage collection là quá trình xác định và loại bỏ các Object không được sử dụng (unreferenced) để giải phóng không gian của Heap để cấp phát cho các đối tượng mới.

Garbage collector là chương trình chạy nền, nó theo dõi toàn bộ các Object trong bộ nhớ (Heap) và tìm ra những Object nào không được dùng nữa (không có Object nảo reference đến nó). Toàn bộ những Object không có reference sẽ bị xóa.

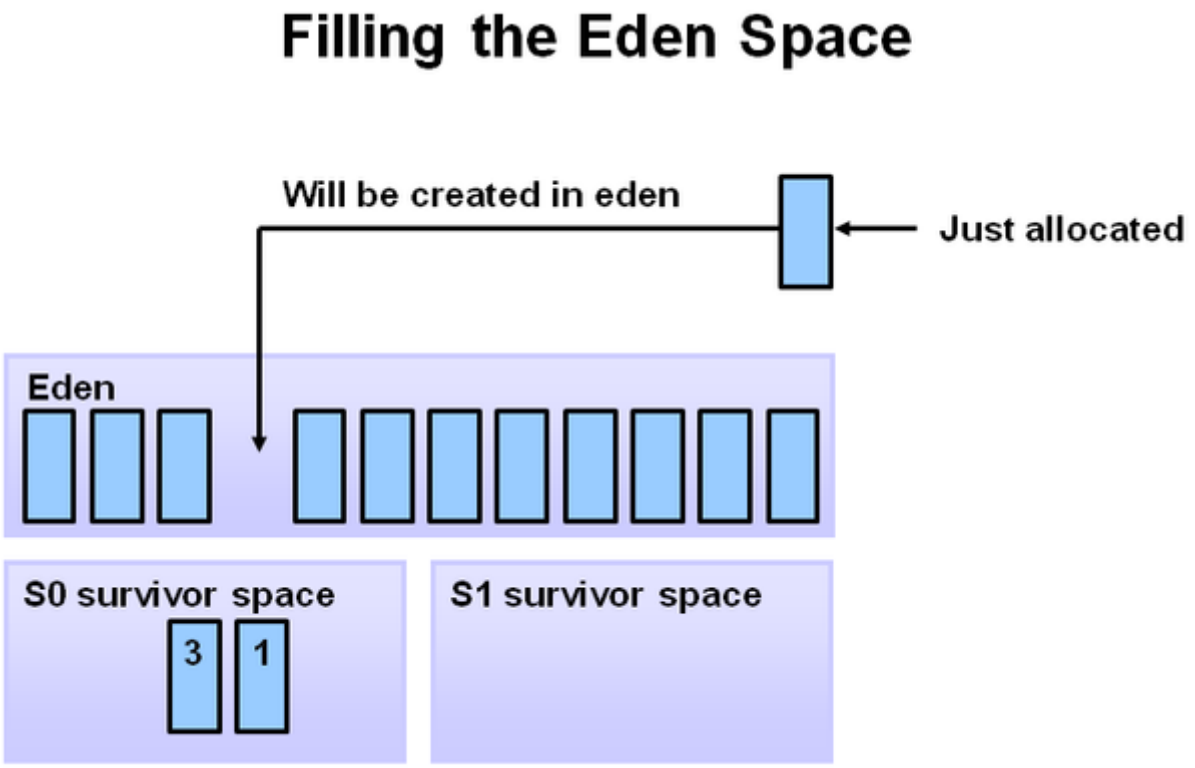
2. Các thành phần của bộ nhớ Heap

Bộ nhớ Heap được chia thành các phần nhỏ như hình dưới đây.

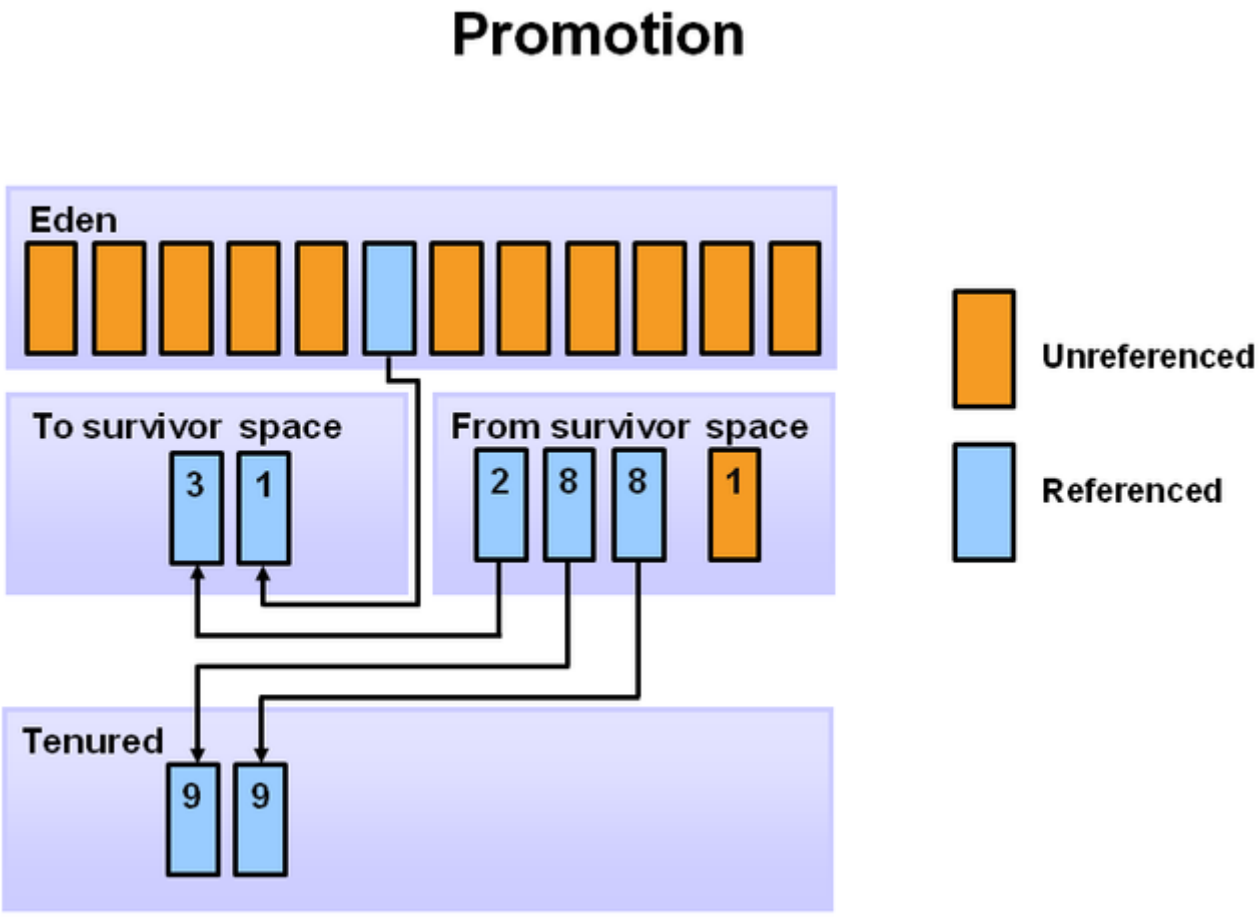


- Young Generation Là nơi chứa toàn bộ Object mới được khởi tạo. Khi vùng nhớ Young generation đầy thì garbage collectior là Minor GC hoạt động. Vùng Young generation lại được chia thành 3 vùng nhỏ hơn là Eden và 2 vùng Survivor là S0, S1.

Ban đầu mọi Object mới tạo được chứa ở vùng Eden, khi Eden đầy thì Minor GC chuyển chúng sang vùng S0, S1.



Minor GC liên tục theo dõi các Object ở S0, S1. Sau "nhiều" chu kỳ quét mà Object vẫn còn được sử dùng thì chúng mới được chuyển sang vùng nhớ **Old generation**. Old generation được quản lý bởi garbage collectior khác là Major GC.



Hình trên mô phỏng 2 Object được chuyển từ vùng Young generation sang Old generation sau 9 chu kỳ quét của Minor GC. Những ô màu vàng tượng chưng cho những Object đã không còn được sử dụng (unreferenced). Chúng sẽ được xóa khi Minor GC hay Majo GC clear vùng nhớ nó quản lý.

- Permanent Generation: Mô hình vùng nhớ Heap có vùng Perm (Permanent Generation), Perm không phải một phần của Heap. Perm không chứa Object, nó chứa metadata của JVM như các thư viện Java SE, mô tả các class và các method của ứng dụng.

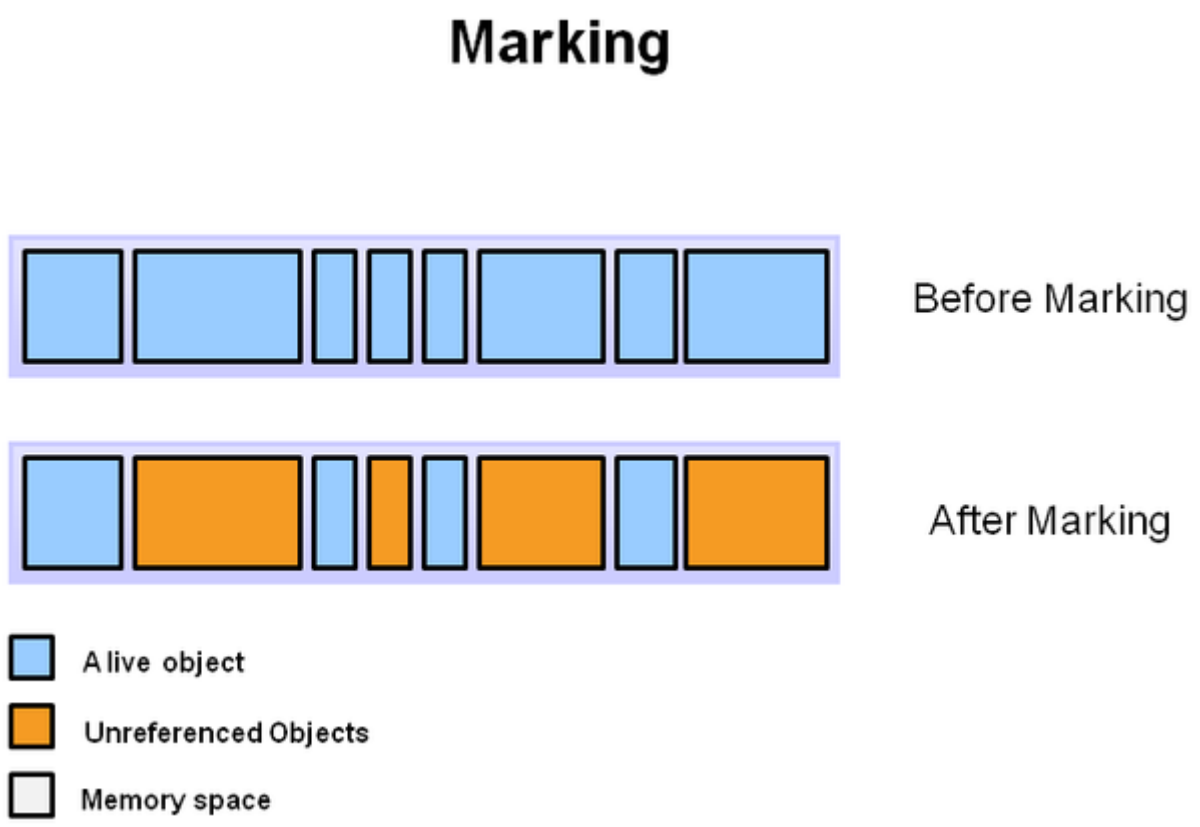
3. Garbage Collector của Java làm việc như thế nào?

Trong Java, bộ garbage collector làm việc hoàn toàn tự động. Đồng nghĩa với việc lập trình viên không cần gọi các lệnh "dọn" bộ nhớ như trong C/C++.

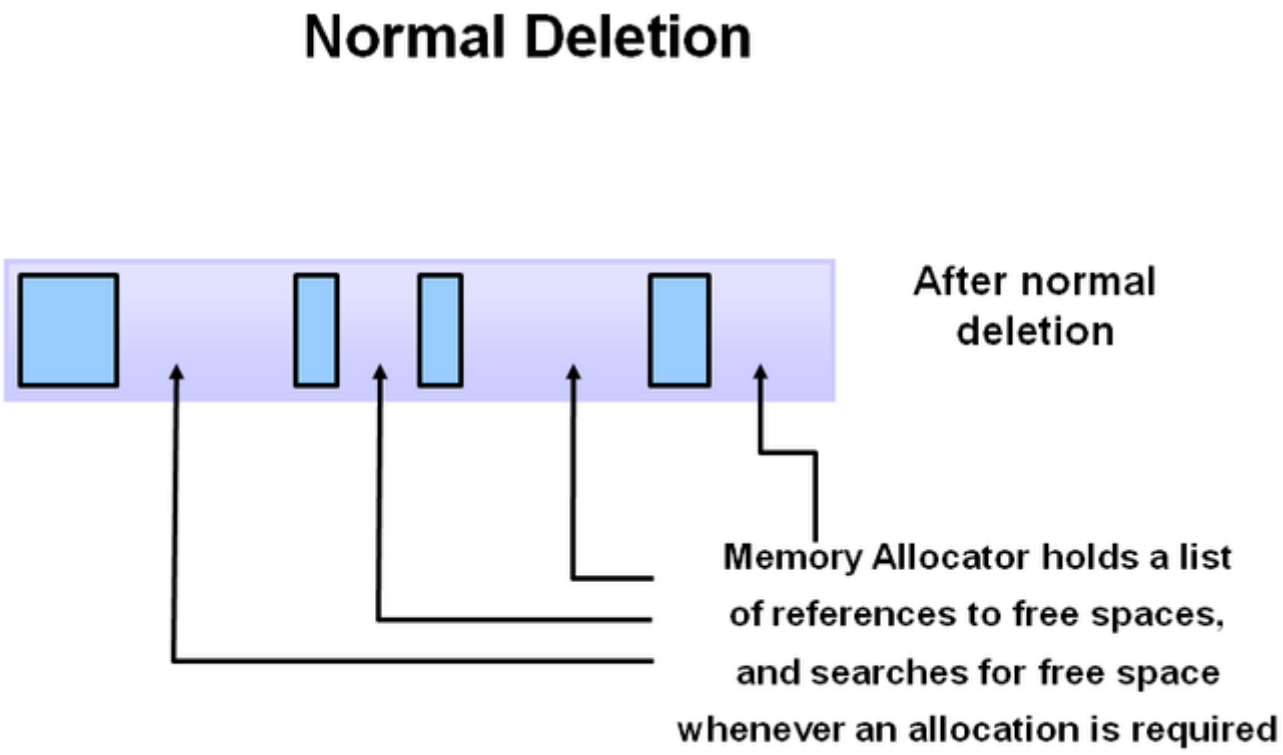
Phần Implementation của garbage collector nằm trong JVM. Mỗi JVM lại có một cách implement garbage collector khác nhau, phù hợp với những đặc tính của JVM đó (JVM Hotspot vs JRockit...)

Quá trình thu gom rác cơ bản thông qua 3 bước sau:

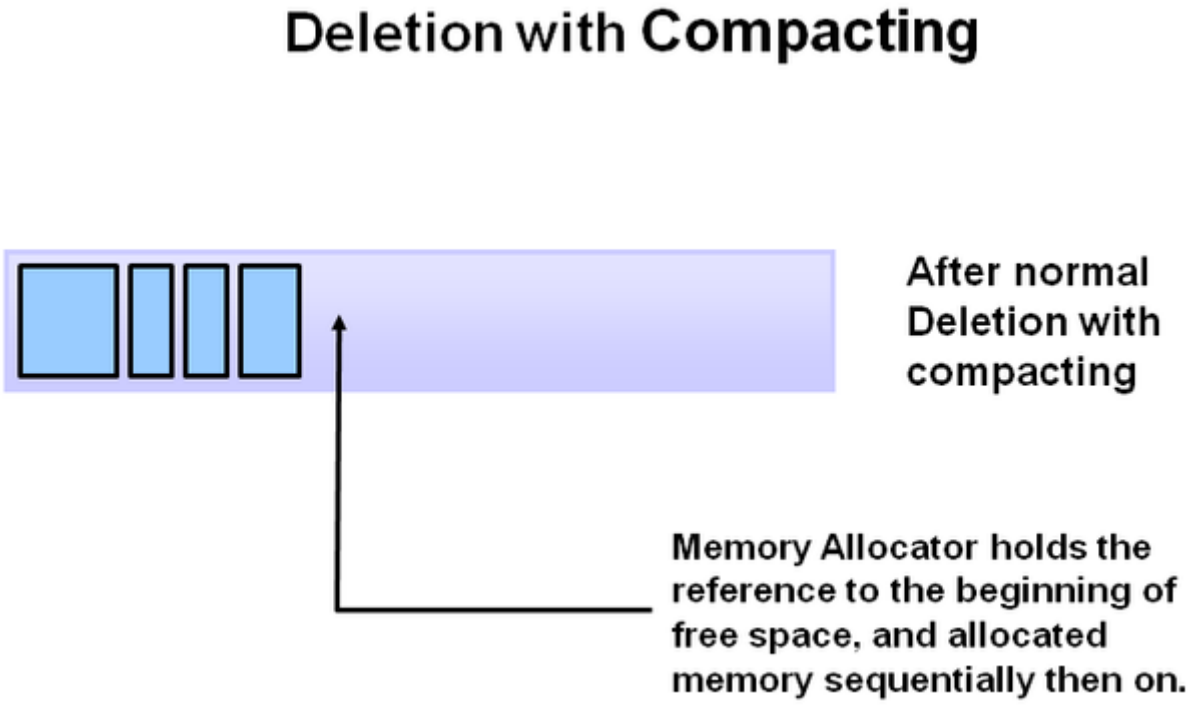
1. Marking: Là bước đánh dấu những Object còn sử dụng và những Object không còn sử dụng.



2. Normal deleting: Trình Garbage Collector sẽ xóa các Object không còn sử dụng.



3. Deletion with Compacting: Sau khi những Object không còn được sử dụng bị xóa, những Object còn được sử dụng sẽ được "gom" lại gần nhau. Điều đó làm tăng hiệu xuất sử dụng bộ nhớ trống để cấp phát cho những Object mới.



Java Virtual Machine - JVM

1. Java Virtual Machine là gì?

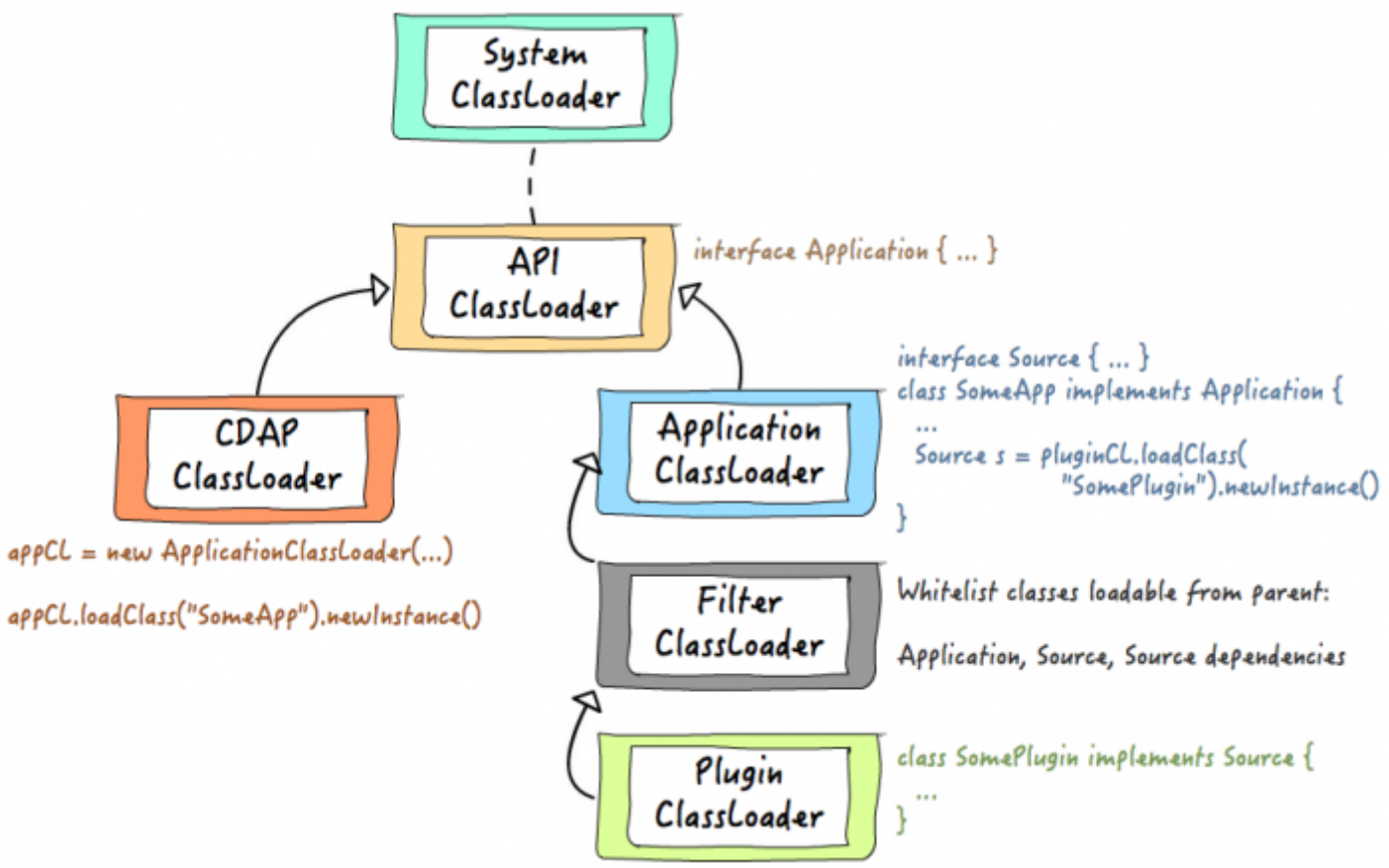
Java Virtual Machine (Viết tắt là JVM) là môi trường dùng để chạy ứng dụng được viết bằng ngôn ngữ lập trình Java:

- 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 trử 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 sữ 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.interm(). 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 Subsytem: 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 nhiện load, liên kết và khởi tạo file .class khí 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 gnas 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 seperate 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)

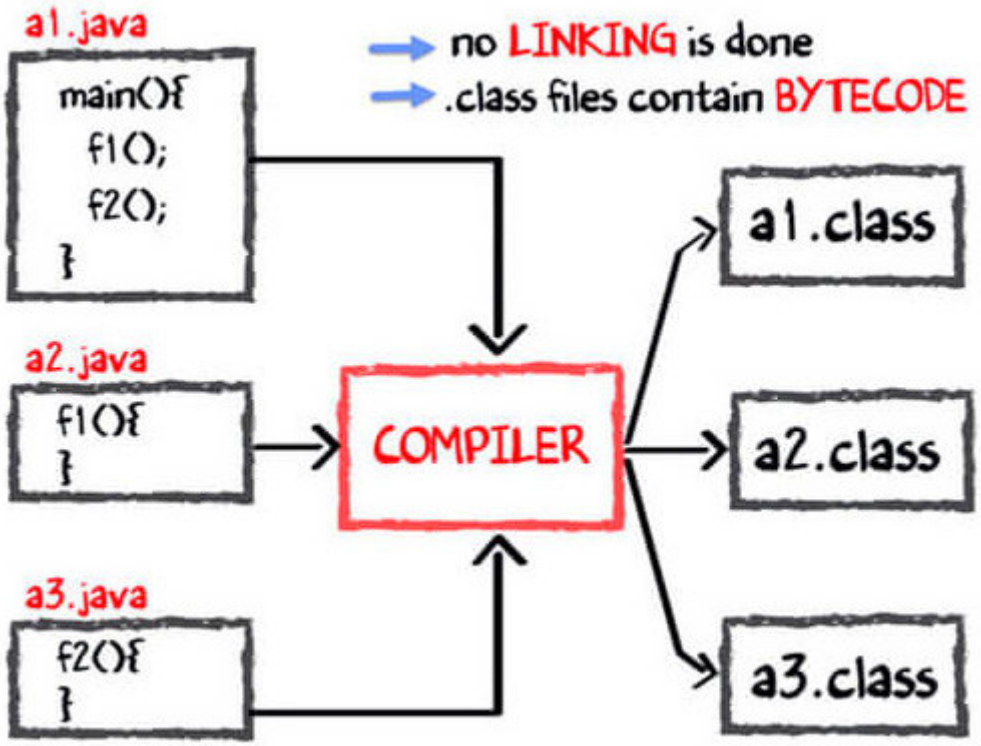
Bytecode sẽ được assign cho Runtime Data Area, và sẽ được thực thi bởi Execution Engine. Execution Engine đọc bytecode từng mảng một.

- **Interpreter** : Trình thông dịch 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 Compilier** : 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

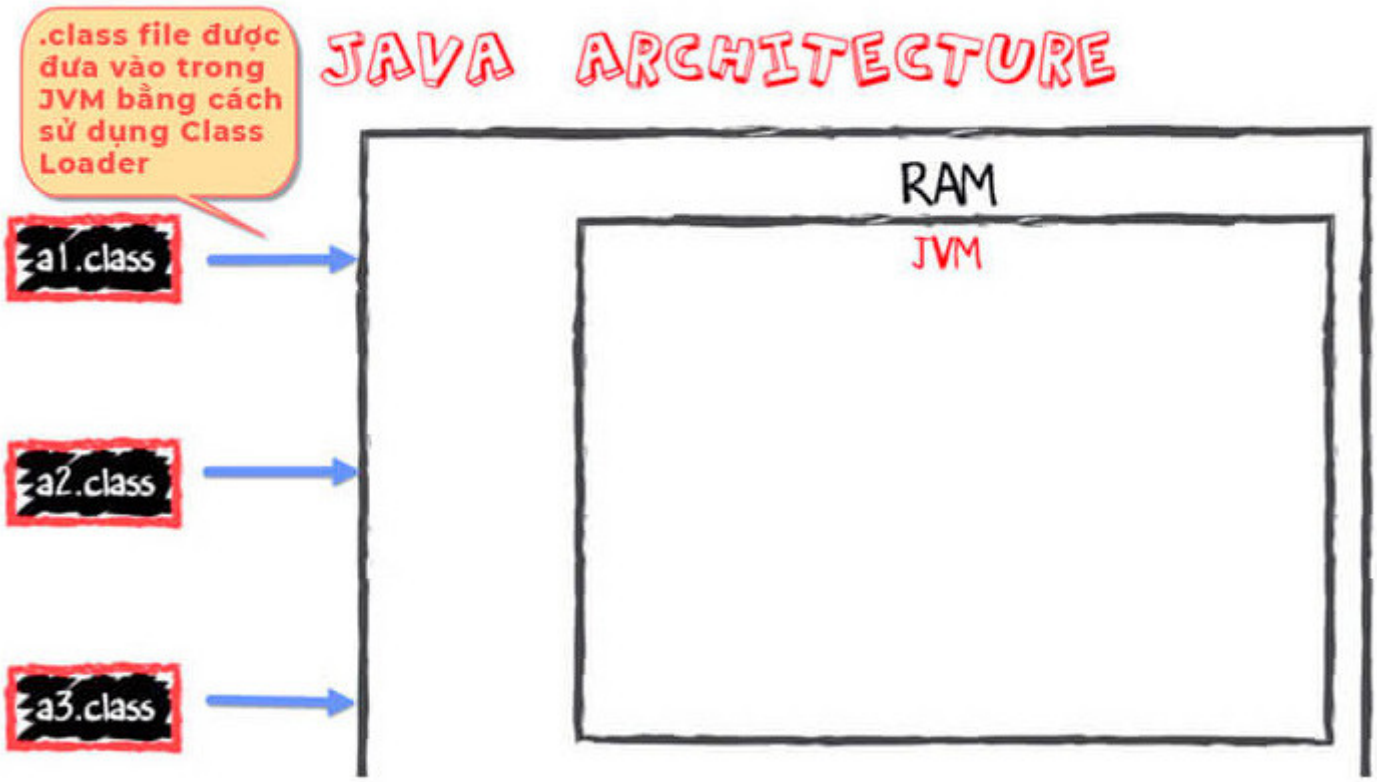
Hãy xem xét quá trình này cho JAVA. Trong main của bạn, bạn có hai method f1 và f2.

- 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

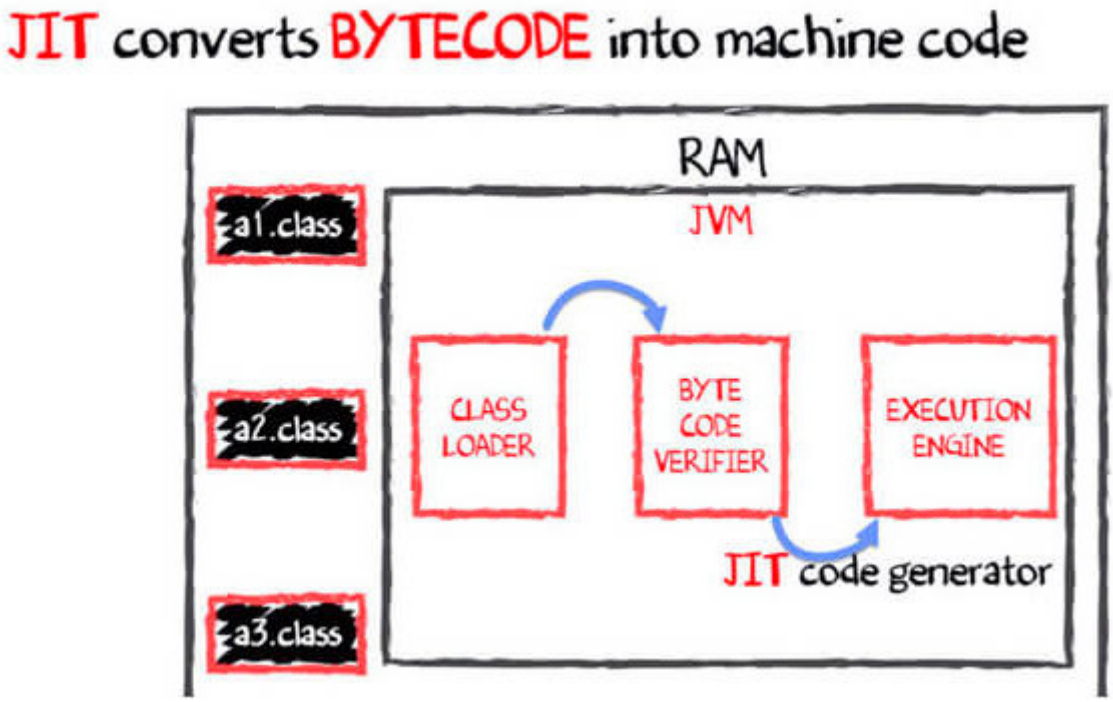


Trình biên dịch sẽ biên dịch ba tệp và tạo ra 3 tệp .class tương ứng chứa BYTE code. Không giống như C, không có liên kết được thực hiện.

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.



Java Stack memory

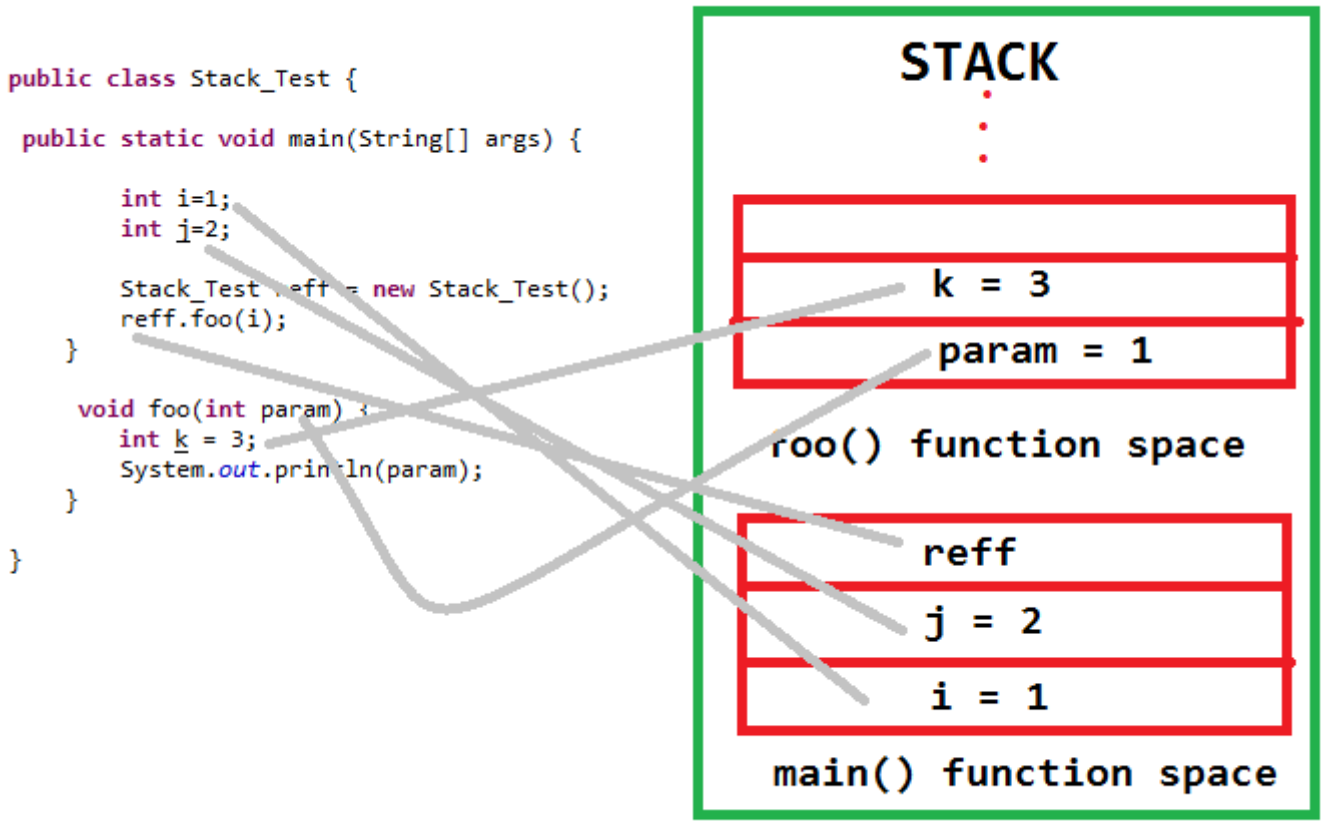
1. Java Stack là gì?

`Stack` là một vùng nhớ được sử dụng để lưu trữ các tham số và các biến local của phương thức mỗi khi một phương thức được gọi ra. Các tham số và các biến local của một phương thức tạo thành một bản ghi kích hoạt, còn được gọi là một stack frame. Các bản ghi kích hoạt được đẩy vào một stack khi phương thức được gọi và đẩy ra khỏi stack khi phương thức trả về. Sự tồn tại tạm thời của các biến này quyết định thời gian sống của các biến.

2. Đặc điểm của của Java Stack memory

- Stack memory được sử dụng cho quá trình thực thi của mỗi thread.
- Bất cứ khi nào gọi 1 hàm, một khối bộ nhớ mới sẽ được tạo trong Stack cho hàm đó để lưu các biến local. Khi hàm thực hiện xong, khối bộ nhớ cho hàm sẽ bị xoá, và giải phóng bộ nhớ trong stack.
- Stack memory bao gồm các giá trị cụ thể của method: các biến local/nguyên thủy và các tham chiếu tới các đối tượng chứa ở trong heap memory được tham chiếu bởi method.
- Stack memory được tham chiếu theo thứ tự LIFO (Last In First Out – vào cuối cùng thì ra đầu tiên). Tức là lưu trữ kiểu ngăn xếp (stack). Khi có một method được thực thi, một block được tạo ra trong stack memory để chứa các biến nguyên thủy local và các tham chiếu tới các object. Khi method kết thúc, block đó sẽ không còn được sử dụng và được phục vụ cho method tiếp theo.
- Stack memory có kích thước rất nhỏ so với Heap memory.

3. Giải thích cách hoạt động của Java Stack



Step 1. Khi chạy chương trình, một thread sẽ khởi tạo và sẽ gọi hàm main ở dòng 1. Một khối bộ nhớ được tạo trong stack cho hàm main().

- `int i=1` : Một biến local được tạo, loại primitive được lưu trong cùng khối bộ nhớ của hàm main()

- `int j=2` : Một biến local được tạo, loại primitive được lưu trong vùng nhớ tiếp theo của stack main()

- `Stack_Test() reff = new Stack_Test()` : một đối tượng được tạo loại Object sẽ được lưu trong bộ nhớ Heap và biến tham chiếu `reff` được lưu trong Stack của hàm main()

Step 2. Hàm `foo()` thì được gọi, vì vậy nó sẽ tạo một khối mới bộ nhớ trong stack cho hàm foo()

- `int param` : Một biến local được tạo (Giá trị được truyền vào), loại primitive được lưu trong cùng khối bộ nhớ của hàm foo()

- `int k=3` : Một biến local được tạo, loại primitive được lưu trong vùng nhớ tiếp theo của stack foo()

- Hàm foo() sẽ kết thúc sau khi hàm `System.out.println()` được thực thi, vì vậy bộ nhớ trong stack cho hàm foo() sẽ được giải phóng.

Step 3. Theo quy luật LIFO, foo() vào sau chết trước, và sau đó hàm main() cũng kết thúc, bộ nhớ trong stack cho hàm main() cũng được giải phóng.

Step 4. Chương trình kết thúc.

4. Chương trình gây lỗi Stack Overflow Error

Chương trình sau là ví dụ gây ra lỗi Stack Overflow Error trong Java, do thực hiện hàm đệ quy dẫn tới không thể giải phóng được bộ nhớ Stack gây tràn bộ nhớ.

```
public class Show_StackOverFlowError {

   public static void main(String[] args) {

      methodOne();
   }

   public static void methodOne(){
     System.out.println("Method One");
     methodTwo();
   }

   public static void methodTwo(){

     System.out.println("Method Two");
     methodOne();
   }
}
```

Output:

```
Method One
Method Two
Method One
Method Two
Method One
Method Two
.
.
.
.
.
.

Exception in thread "main" java.lang.StackOverflowError
at java.base/sun.nio.cs.UTF_8$Encoder.encodeLoop(UTF_8.java:695)
```

```
at java.base/java.nio.charset.CharsetEncoder.encode(CharsetEncoder.java:578)
at java.base/sun.nio.cs.StreamEncoder.implWrite(StreamEncoder.java:292)
at java.base/sun.nio.cs.StreamEncoder.implWrite(StreamEncoder.java:281)
at java.base/sun.nio.cs.StreamEncoder.write(StreamEncoder.java:125)
at java.base/java.io.OutputStreamWriter.write(OutputStreamWriter.java:211)
at java.base/java.io.BufferedWriter.flushBuffer(BufferedWriter.java:120)
at java.base/java.io.PrintStream.write(PrintStream.java:526)
at java.base/java.io.PrintStream.print(PrintStream.java:666)
at java.base/java.io.PrintStream.println(PrintStream.java:803)
at Show_StackOverFlowError.methodOne(Show_StackOverFlowError.java:16)
at Show_StackOverFlowError.methodTwo(Show_StackOverFlowError.java:22)
at Show_StackOverFlowError.methodOne(Show_StackOverFlowError.java:17)
at Show_StackOverFlowError.methodTwo(Show_StackOverFlowError.java:22)
at Show_StackOverFlowError.methodOne(Show_StackOverFlowError.java:17)
```

Java Heap - Stack memory

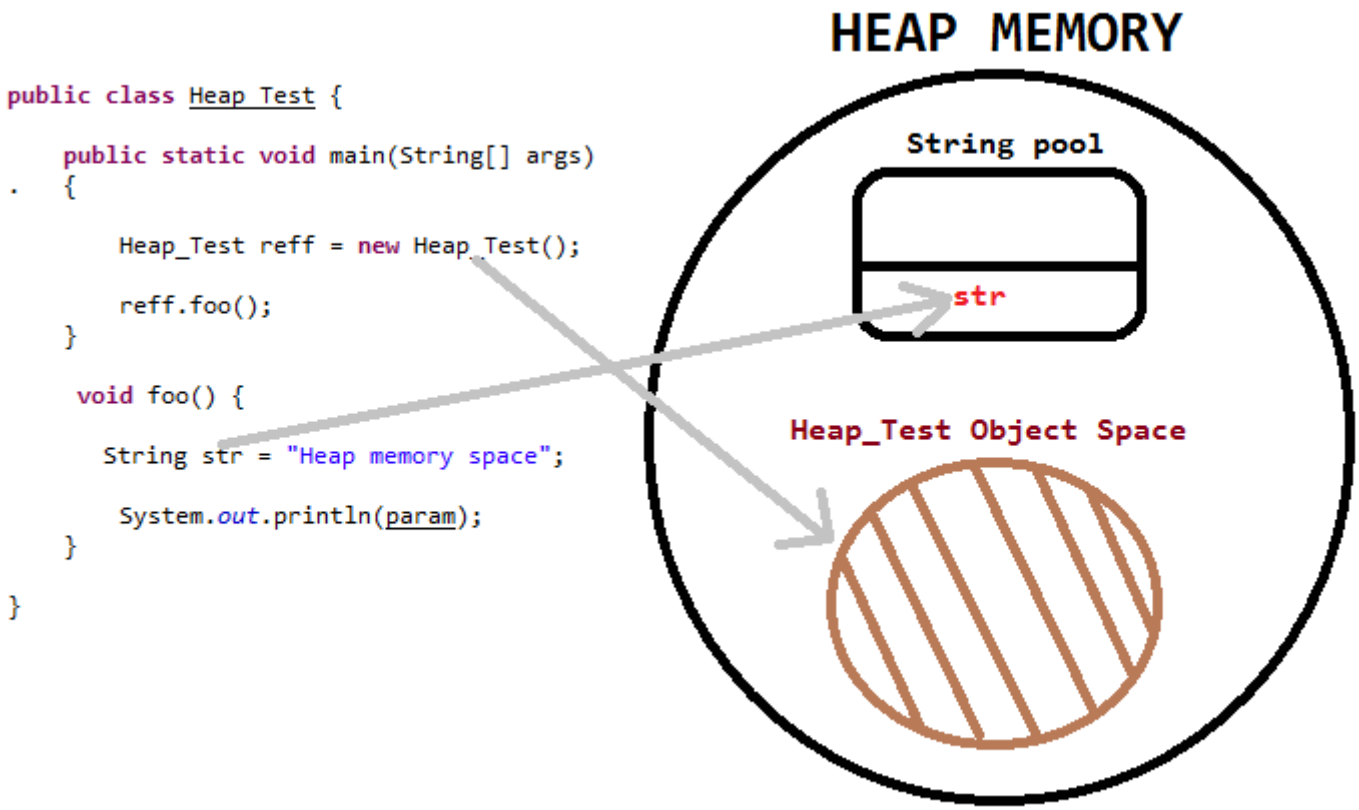
Phân biệt cách hoạt động và cấp phát của Heap và Stack memory trong Java

Xem lại:

- Heap memory
- Stack memory

1. Heap memory

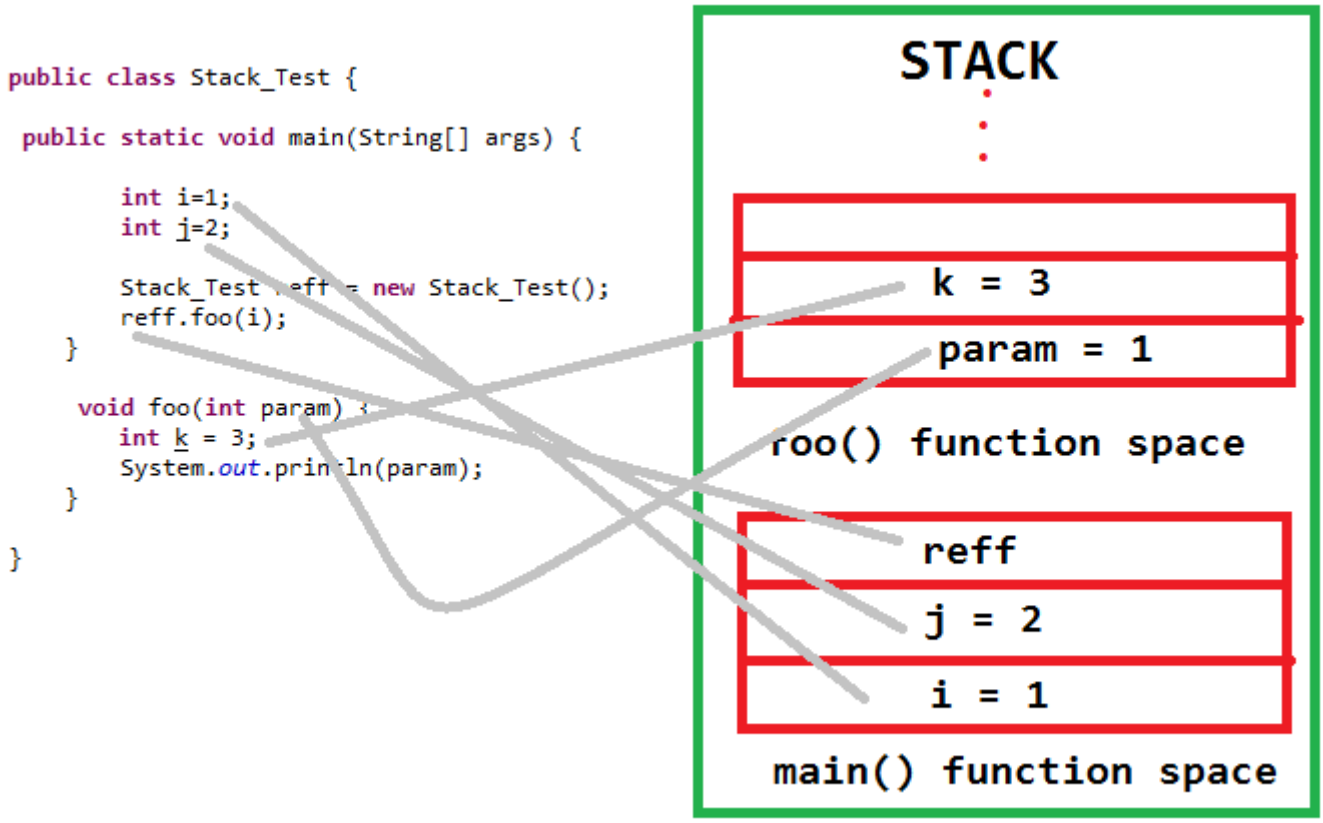
Heap là một vùng nhớ trong bộ nhớ được sử dụng để lưu trữ các đối tượng khi từ khóa new được gọi ra, các biến static và các biến toàn cục (biến instance).



2. Stack memory

Stack là một vùng nhớ được sử dụng để lưu trữ các tham số và các biến local của phương thức mỗi khi một phương thức được gọi ra. Các tham số và các biến local của một phương thức tạo thành một bản ghi kích hoạt, còn được gọi là một stack frame. Các bản ghi kích hoạt được đẩy vào một stack khi phương thức được gọi và đẩy ra khỏi stack khi phương thức trả về. Sự tồn tại tạm thời của

các biến này quyết định thời gian sống của các biến.



3. Heap and Stack memory

Chương trình sau là ví dụ cách quản lý và cấp phát bộ nhớ Heap và Stack trong Java

```
public class Heap_Stack {

        //main() method thread creates space in stack memory
        public static void main(String[] args) {

            // primitive datatype created inside main() method space in stack memory
            int i=1;

            // Object created in heap memory and its refference obj in stack memory
            Object obj = new Object();

            // Heap_Stack Object created in heap memory and its refference objnew in stack
memory
            Heap_Stack objnew = new Heap_Stack();
```

```
            // New space for foo() method created in the top of the stack memory
            objnew.foo(obj);

        }

        private void foo(Object p) {

           //  String for p.toString() is created in String Pool and refference str created in
stack memory
            String str = p.toString();

            System.out.println(str);
        }
}
```

Step 1. Khi chạy chương trình, một thread sẽ khởi tạo và sẽ gọi hàm main ở dòng 1. Một khối bộ nhớ được tạo trong stack cho hàm main().

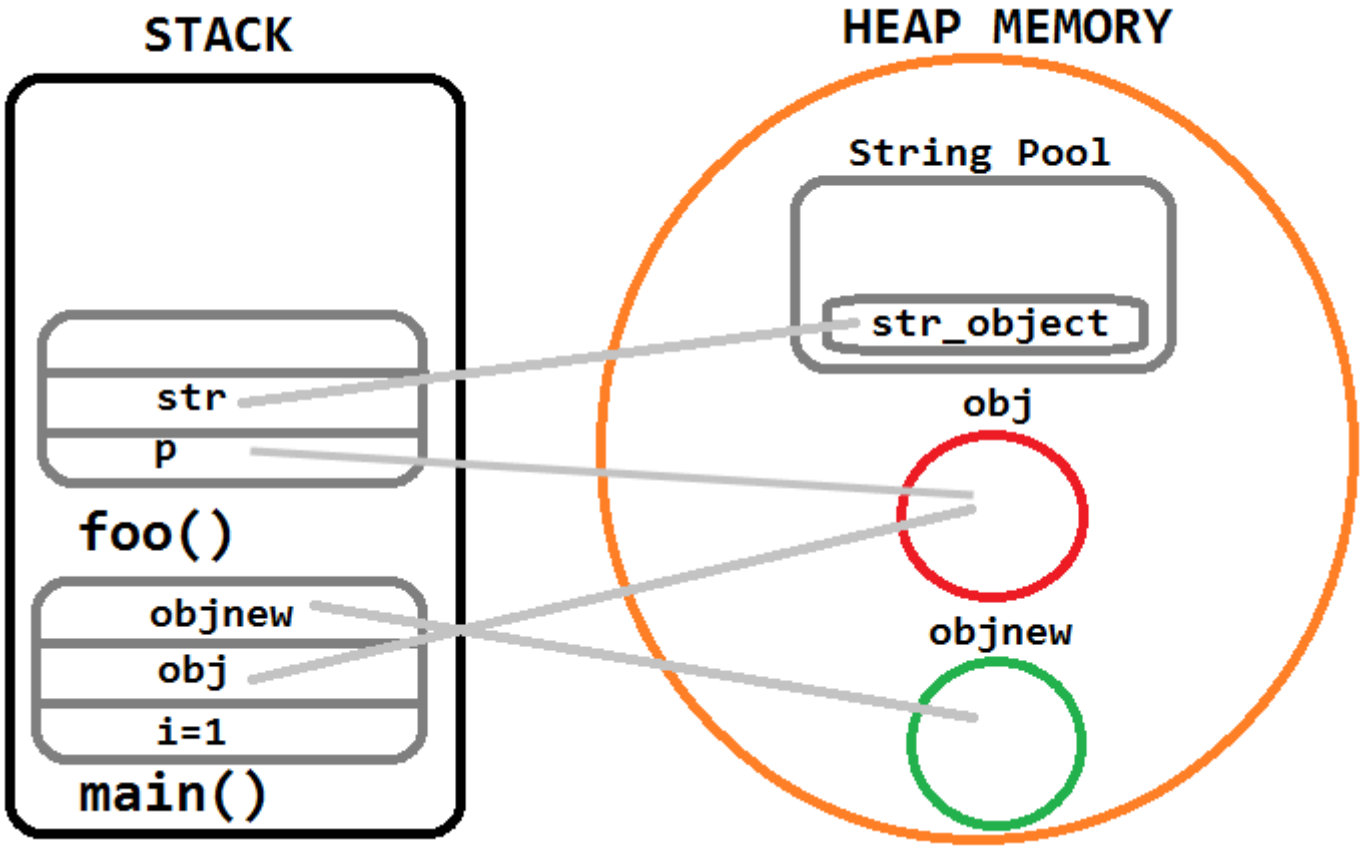
- `int i=1` : Một biến local được tạo, loại primitive được lưu trong cùng khối bộ nhớ của hàm main()
- `Object obj = new Object()` : một đối tượng được tạo loại Object sẽ được lưu trong bộ nhớ Heap và biến tham chiếu `obj` được lưu trong Stack của hàm main()
- `Heap_Stack objnew = new Heap_Stack()` : một đối tượng được tạo loại Object sẽ được lưu trong bộ nhớ Heap và biến tham chiếu `objnew` được lưu trong Stack của hàm main()

Step 2. Hàm `foo()` thì được gọi, vì vậy nó sẽ tạo một khối mới bộ nhớ trong stack cho hàm foo()

- `Object p` : Một biến local `p` được tạo (Giá trị được truyền vào), loại tham chiếu được lưu trong cùng khối bộ nhớ của hàm foo()
- `String str=p.toString()` : Một biến String được tạo được lưu trong String Pool, biên tham chiếu `str` được lưu trong vùng nhớ tiếp theo của stack foo()
- Hàm foo() sẽ kết thúc sau khi hàm `System.out.println()` được thực thi, vì vậy bộ nhớ trong stack cho hàm foo() sẽ được giải phóng.

Step 3. Theo quy luật LIFO, foo() vào sau chết trước, và sau đó hàm main() cũng kết thúc, bộ nhớ trong stack cho hàm main() cũng được giải phóng.

Step 4. Chương trình kết thúc.



4. Heap vs Stack memory

Sự khác nhau Java Heap và Stack memory

#	Heap Memory	Stack Memory
1	**Java Heap Memory** là bộ nhớ được sử dụng ở runtime để lưu các Objects. Bất cứ khi nào ở đâu trong chương trình của bạn khi bạn tạo Object thì nó sẽ được lưu trong Heap (thực thi toán tử new).	**Stack Memory** là bộ nhớ để lưu các biến local trong hàm và lời gọi hàm ở runtime trong một Thread java. Các biến local bao gồm: loại nguyên thuỷ (primitive), loại tham chiếu tới đối tượng trong heap (reference), khai báo trong hàm, hoặc đối số được truyền vào hàm.
2	Thời gian sống của bộ nhớ Heap dài hơn so với Stack. Thời gian sống của object phụ thuộc vào Garbage Collection của java. Garbage Collection sẽ chạy trên bộ nhớ Heap để xoá các Object không được sử dụng nữa, nghĩa là object không được referece trong chương trình.	Thường có thời gian sống ngắn.

3	Các objects trong Heap đều được truy cập bởi **tất cả các các nơi trong ứng dụng, bởi các threads khác nhau**.	Stack chỉ được sử dụng cho **một Thread duy nhất**. Thread ngoài không thể truy cập vào được.
4	Cơ chế quản lý của Heap thì phức tạp hơn. Heap đuơc phân làm 2 loại Young-Generation, Old-Generation. Đọc thêm về Garbage Collection để hiểu rõ hơn.	Cơ chế hoạt động là **LIFO** (Last-In-First-Out), chạy sau chết trước.
5	Dung lượng Heap thường lớn hơn Stack.	Bộ nhớ stack thường nhỏ.
6	Sử dụng -Xms và -Xmx để định nghĩa dung lượng bắt đầu và dung lượng tối đa của bộ nhớ heap.	Dùng -Xss để định nghĩa dung lượng bộ nhớ stack.
7	Khi Heap bị đầy chương trình hiện lỗi `java.lang.OutOfMemoryError`: Java Heap Space	Khi stack bị đầy bộ nhớ, chương trình phát sinh lỗi: `java.lang.StackOverFlowError`
8	Truy cập vùng nhớ Heap chậm hơn Stack.	Truy cập stack nhanh hơn Heap
9	Dung lượng sử dụng của Heap sẽ tăng giảm phụ thuộc vào Objects sử dụng.	Bất cứ khi nào gọi 1 hàm, một khối bộ nhớ mới sẽ được tạo trong Stack cho hàm đó để lưu các biến local. Khi hàm thực hiện xong, khối bộ nhớ cho hàm sẽ bị xoá, và giải phóng bộ nhớ trong stack.

Catch an SQL exception with the error code ORA-00942 in Java

Để bắt ngoại lệ SQL bằng mã lỗi ORA-00942 trong Java, bạn có thể sử dụng lớp SQLException cùng với cách xử lý ngoại lệ dành riêng cho Oracle. Lỗi ORA-00942 chỉ ra rằng bảng hoặc dạng xem bạn đang cố truy cập không tồn tại hoặc bạn không có các quyền cần thiết để truy cập vào nó.

Đây là ví dụ xử lý ngoại lệ cụ thể này trong Java:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class OracleSQLExceptionExample {
    public static void main(String[] args) {
        Connection connection = null;
        Statement statement = null;

        try {
            // Initialize the database connection
            connection =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:yourdb", "username",
"password");
            statement = connection.createStatement();

            // Attempt to execute an SQL statement that may cause ORA-00942
            // Replace "YourTable" with the actual table name you are trying to access
            String sql = "SELECT * FROM YourTable";
            statement.executeQuery(sql);
        } catch (SQLException e) {
            if (e.getErrorCode() == 942) {
                // ORA-00942: Table or view does not exist
```

```
                System.out.println("Caught ORA-00942: Table or view does not exist");
                // You can handle this exception here or log it as needed.
            } else {
                // Handle other SQL exceptions
                e.printStackTrace();
            }
        } finally {
            try {
                if (statement != null) {
                    statement.close();
                }
                if (connection != null) {
                    connection.close();
                }
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```