Khái niệm về đa luồng trong java

**Đa luồng (multithreading) trong java** là một tiến trình thực hiện nhiều luồng đồng thời.

Luồng (thread) về cơ bản là một tiến trình con (sub-process). Nó là đơn vị nhỏ nhất của tiến trình. Đa tiến trình (multiprocessing) và đa luồng (multithreading) cả hai được sử dụng để tạo ra hệ thống đa nhiệm (multitasking).

Nhưng chúng ta sử dụng đa luồng nhiều hơn đa tiến trình bởi vì các luồng chia sẻ một vùng bộ nhớ chung. Chúng không phân bổ vùng bộ nhớ riêng biệt để tiết kiệm bộ nhớ, và chuyển đổi ngữ cảnh giữa các luồng mất ít thời gian hơn tiến trình.

Đa luồng trong java được sử dụng hầu hết trong các game, hoạt hình,...

## Ưu điểm của đa luồng trong java

1) Nó không chặn người sử dụng vì các luồng là độc lập và ta có thể thực hiện nhiều công việc cùng một lúc.

2) Ta có thể thực hiện nhiều hoạt động với nhau để tiết kiệm thời gian.

3) Luồng là độc lập vì vậy nó không ảnh hưởng đến luồng khác nếu ngoại lệ xảy ra trong một luồng duy nhất.

### **Đa nhiệm dựa trên tiến trình (Process) - Đa tiến trình (Multiprocessing)**

* Mỗi tiến trình có địa chỉ riêng trong bộ nhớ, tức là mỗi tiến trình phân bổ vùng nhớ riêng biệt.
* Tiến trình là nặng.
* Sự giao tiếp giữa các tiến trình có tri phí cao.
* Chuyển đổi từ tiến trình này sang tiến trình khác đòi hỏi thời gian để đăng ký việc lưu và tải, các bản đồ bộ nhớ, các danh sách cập nhật, vv.

### **Đa nhiệm dựa trên luồng (Thread) - Đa luồng (MultiThreading)**

* Các luồng chia sẻ không gian địa chỉ ô nhớ giống nhau.
* Luồng là nhẹ.
* Sự giao tiếp giữa các luồng có chi phí thấp.

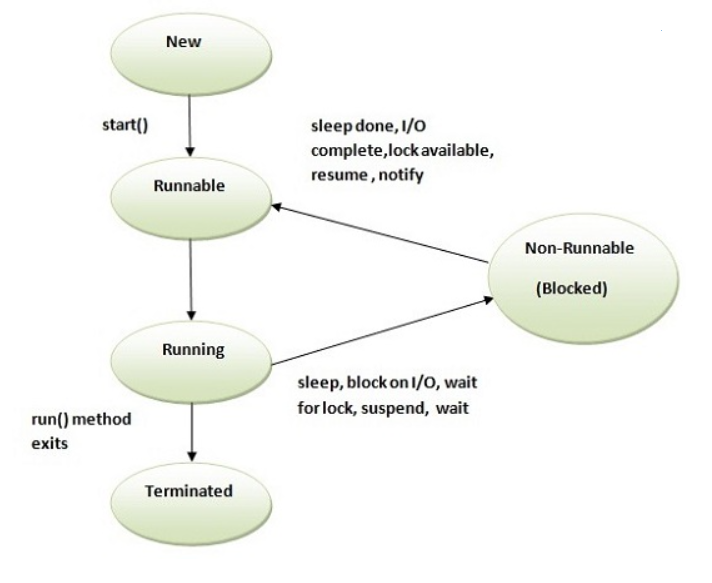
Vòng đời của thread (các trạng thái) trong java

Một thread có thể nằm trong một trong năm trạng thái. Theo Sun, chỉ có 4 trạng thái trong vòng đời của thread trong java đó là new, runnable, non-runnable và terminated. Không có trạng thái run.

Nhưng để hiểu rõ hơn các thread, chúng ta sẽ tìm hiểu thread ở trong 5 trạng thái.

Vòng đời của thread trong java được kiểm soát bởi JVM. Các trạng thái của thread java như sau:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



## 1. New

Thread ở trạng thái new nếu ta tạo một thể hiện của lớp Thread nhưng trước khi gọi phương thức start().

## 2. Runnable

Thread ở trạng thái runnable sau khi gọi phương thức start(), nhưng trình lên lịch (scheduler) của thread đã không chọn nó là thread đang chạy.

## 3. Running

Thread ở trạng thái running nếu trình lên lịch của thread đã chọn nó.

## 4. Non-Runnable (Blocked)

Đây là trạng thái khi thread vẫn còn sống, nhưng hiện tại không được chọn để chạy.

## 5. Terminated

Một thread ở trong trạng thái terminated hoặc dead khi phương thức run() của nó bị thoát.

Làm thế nào để tạo thread trong java

Có hai cách để tạo ra một thread:

1. Bởi extends lớp Thread
2. Bởi implements Runnable interface.

Lớp Thread

Lớp Thread cung cấp các constructor và phương thức để tạo và thực hiện các hoạt động trên một thread. Lớp Thread extends từ lớp Object và implements Runnable interface.

Các constructor thường được sử dụng của lớp Thread

1. Thread()
2. Thread(String name)
3. Thread(Runnable r)
4. Thread(Runnable r,String name)

Các phương thức thường được sử dụng của lớp Thread

1. **public void run():**Được sử dụng để thực hiện hành động cho một thread..
2. **public void start():**Bắt đầu thực hiện thread. JVM gọi phương thức run() trên thread.
3. **public void sleep(long miliseconds):**Làm cho thread hiện tại tạm ngừng thực thi cho số mili giây quy định.
4. **public void join():**Đợi cho một thread chết.
5. **public void join(long miliseconds):**Đợi cho một thread chết với các mili giây quy định.
6. **public int getPriority():**Trả về mức độ ưu tiên của thread.
7. **public int setPriority(int priority):**Thay đổi mức độ ưu tiên của thread.
8. **public String getName():**Trả về tên của thread.
9. **public void setName(String name):**Thay đổi tên của thread.
10. **public Thread currentThread():**Trả về tham chiếu của thread đang được thi hành.
11. **public int getId():**Trả về id của thread.
12. **public Thread.State getState():**Trả về trạng thái của thread.
13. **public boolean isAlive():**Kiểm tra nếu thread còn sống.
14. **public void yield():**Làm cho các đối tượng thread đang thực thi để tạm thời tạm dừng và cho phép các thread khác để thực thi.
15. **public void suspend():**Được sử dụng để hoãn lại các thread (không dùng nữa).
16. **public void resume():**Được sử dụng để tiếp tục các thread đang bị hoãn (không dùng nữa).
17. **public void stop():**Được sử dụng để dừng thread (không dùng nữa).
18. **public boolean isDaemon():**Kiểm tra nếu thread là một luồng hiểm.
19. **public void setDaemon(boolean b):**Đánh dấu thread là luồng hiểm hoặc luồng người dùng.
20. **public void interrupt():**Ngắt thread.
21. **public boolean isInterrupted():**Kiểm tra nếu thread đã bị ngắt.
22. **public static boolean interrupted():**Kiểm tra nếu thread hiện tại đã bị ngắt.

Runnable interface

Giao tiếp Runnable nên được cài đặt bởi bất kỳ lớp nào mà thể hiển của lớp đó dự định sẽ được thực thi bởi thread. Giao tiếp Runnable chỉ có một phương thức run().

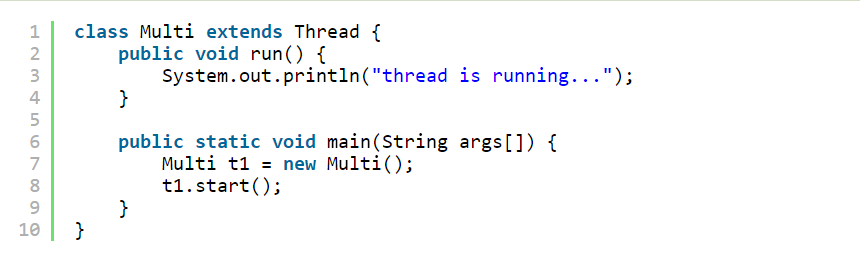
1. **public void run():**Được sử dụng để thực hiện hành động cho một thread.

Start một thread

Phương thức start() của lớp Thread được sử dụng để bắt đầu một thread mới được tạo. Nó thực hiện các nhiệm vụ sau:

* Start một new thread(với new callstack).
* Thread chuyển từ trạng thái New sang trạng thái Runnable.
* Khi thread được một cơ hội để thực thi, phương thức run() của nó sẽ chạy.

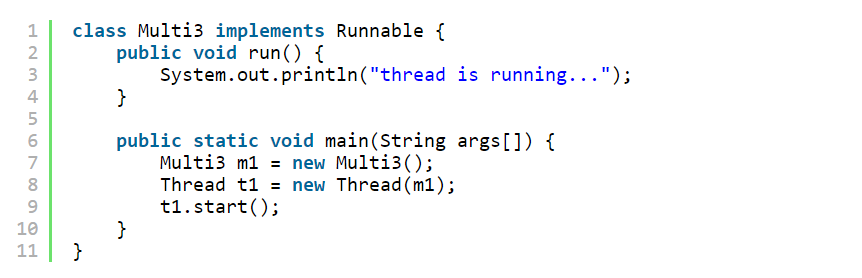
## Ví dụ về thread trong java bởi extends lớp Thread



Output:



## Ví dụ về thread trong java bởi extends Runnable interface



Output:



Nếu bạn không extends lớp Thread, đối tượng lớp của bạn sẽ không được xử lý như là một đối tượng của thread. Đối tượng của lớp mà implements Runnable để phương thức run() của lớp đó có thể được thực thi.

## Định nghĩa

**Thread scheduler trong java** hay trình lên lịch của thread trong java là một phần của JVM có nhiệm vụ quyết định thread nào nên được chạy.

Không có sự đảm bảo rằng thread có trạng thái runnable nào sẽ được chọn để chạy bởi thread scheduler.

Chỉ có một thread tại một thời điểm có thể chạy trong một xử lý duy nhất.

Thread scheduler chủ yếu sử dụng việc lập kệ hoạch ưu tiên hoặc phân chia thời gian để lên lịch các luồng.

## Sự khác biệt giữa lập kệ hoạch ưu tiên và phân chia thời gian

Theo kế hoạch ưu tiên, nhiệm vụ ưu tiên cao nhất được thực hiện cho đến khi nó đi vào trạng thái đợi hoặc trạng thái chết hoặc nhiệm vụ ưu tiên cao hơn xuất hiện. Theo thời gian được phân chia, một công việc thực hiện cho một thời gian phân chia định trước và sau đó nhập lại vào pool chứa các nhiệm vụ đã sẵn sàng. Trình lên lịch xác định nhiệm vụ nào cần thực hiện tiếp theo, dựa trên các ưu tiên và các yếu tố khác.

# **Phương thức sleep trong java**

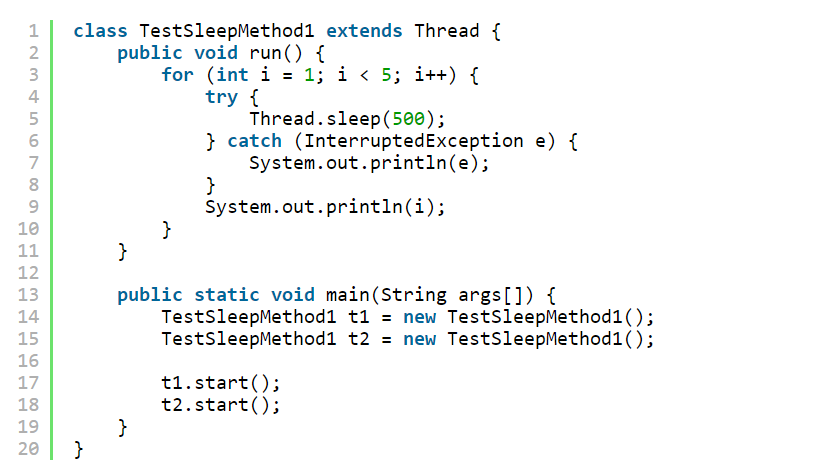
Phương thức sleep() của lớp Thread được sử dụng để tạm ngưng một thread cho một khoảng thời gian nhất định.

### **Cú pháp của phương thức sleep trong java**

Lớp Thread cung cấp hai phương thức để tạm ngưng một thread:

* public static void sleep(long miliseconds)throws InterruptedException
* public static void sleep(long miliseconds, int nanos)throws InterruptedException

## Ví dụ về phương thức sleep trong java

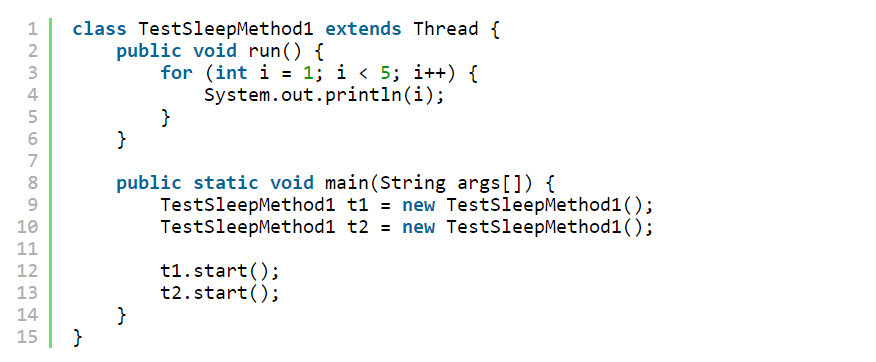


Output:



Như bạn biết rõ rằng tại một thời điểm chỉ có một thread được thực thi. Nếu bạn tạm ngưng một thread trong một khoảng thời gian nhất định, thì thread scheduler chọn một thread khác để thực thi.

## Trường hợp không sử dụng phương thức sleep()



Output:

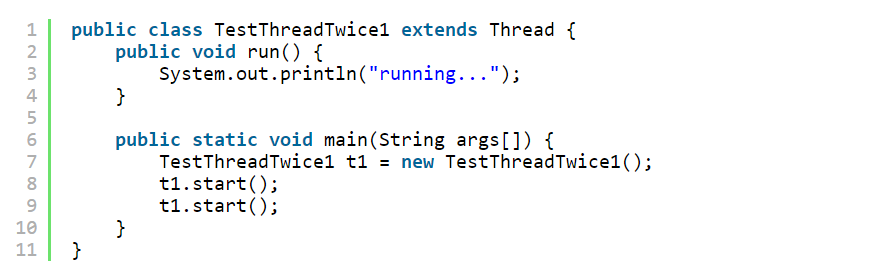


# **Start một thread hai lần**

## Có thể start một thread hai lần không?

Không. Sau khi start một thread, nó không bao giờ có thể được start lại. Nếu bạn làm như vậy, một ngoại lệ IllegalThreadStateException sẽ xảy ra.

Ví dụ minh họa:



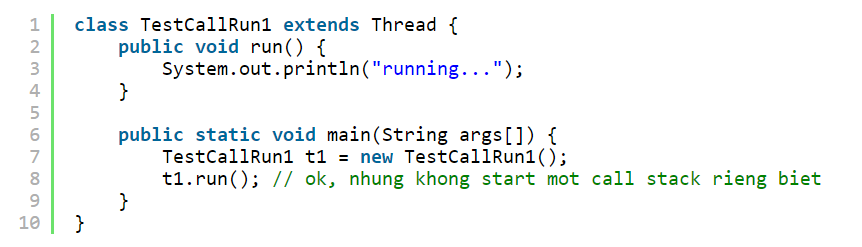
Output:



# **Gọi phương thức run()**

Chuyện gì xảy ra khi gọi trực tiếp phương thức run() thay vì phương thức start()

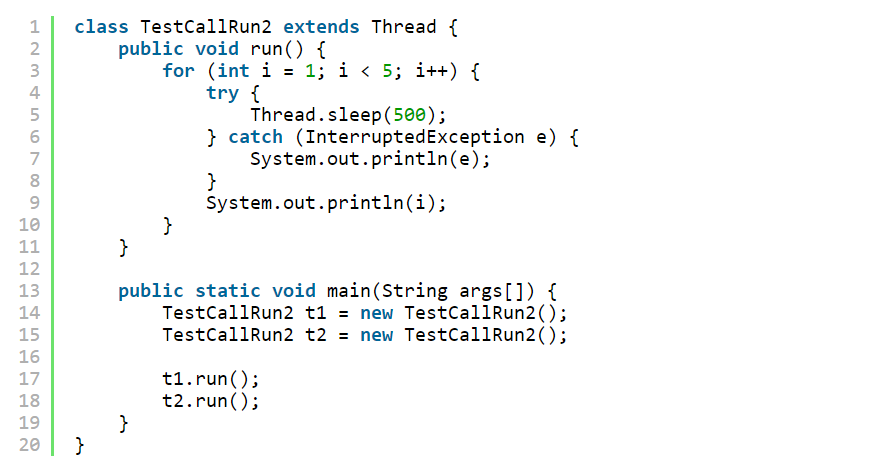
* Mỗi thread start trong một call stack riêng biệt.
* Việc gọi phương thức run() từ luồng chính, phương thức run() được xếp vào call stack chứ không phải khởi tạo một call stack mới.



Output:



Vấn đề khi gọi trực tiếp phương thức run()



Output:



Như bạn thấy trong chương trình trên, không có sự chuyển đổi ngữ cảnh vì ở đây t1 và t2 sẽ được coi như đối tượng thông thường chứ không phải đối tượng luồng.

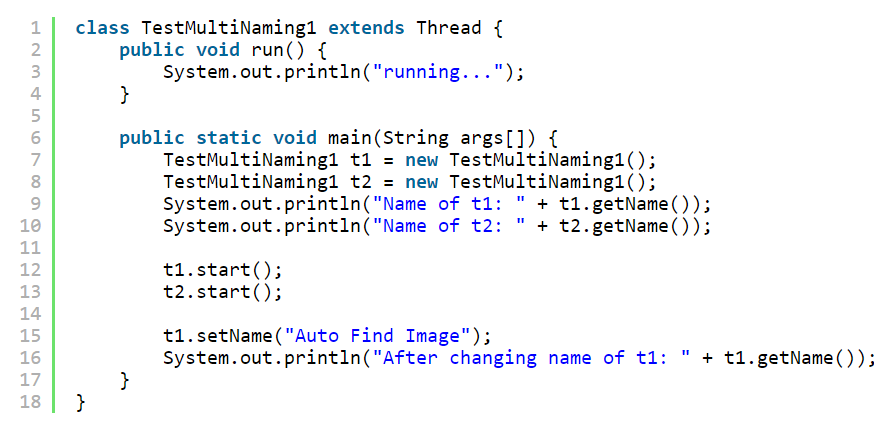
# **Đặt tên cho thread và thread hiện tại**

## Đặt tên cho Thread

Lớp Thread cung cấp các phương thức để thay đổi và lấy ra tên cho một thread. Theo mặc định, mỗi thread có một tên, tức là thread-0, thread-1, v.v. Bởi chúng ta có thể thay đổi tên của thread bằng cách sử dụng phương thức setName(). Cú pháp của setName() và getName() như sau:

1. **public String getName():** được sử dụng để trả về tên của một thread.
2. **public void setName(String name):** được sử dụng để thay đổi tên của một thread.

### **Ví dụ về việc đổi tên cho thread**



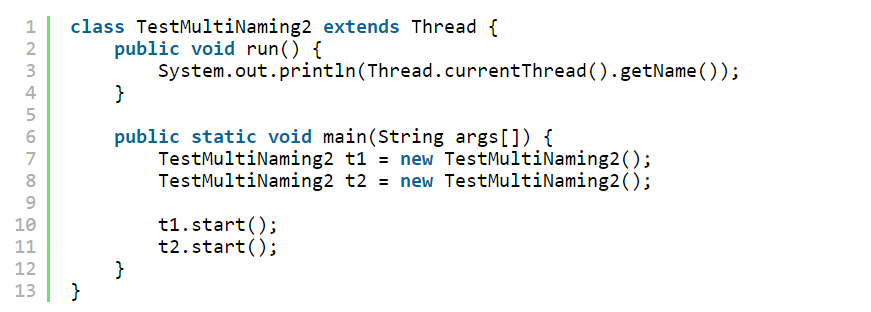
Output:



## Thread hiện tại

Phương thức currentThread() trả về một tham chiếu của thread hiện đang thực thi.

### **Ví dụ về phương thức currentThread()**



Output:



# **Mức ưu tiên của một Thread**

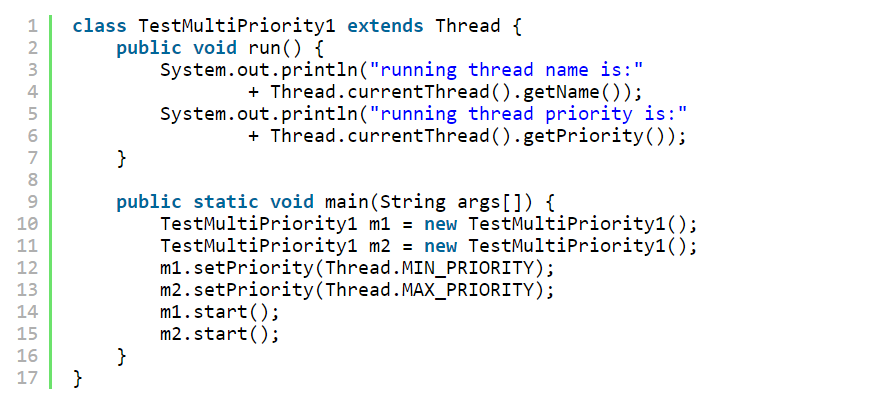
Mỗi thread có một mức ưu tiên (priority). Ưu tiên được đại diện bởi một số từ 1 đến 10. Trong hầu hết các trường hợp, lịch trình của thread được sắp xếp theo thứ tự ưu tiên của chúng (được gọi là lập kế hoạch ưu tiên). Nhưng nó không được bảo đảm bởi vì nó phụ thuộc vào thông số kỹ thuật của JVM.

## 3 hằng số được định nghĩa trong lớp Thread

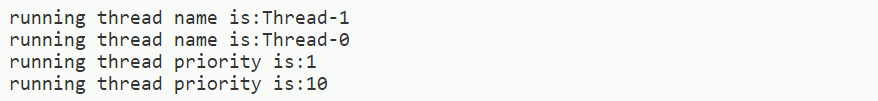
1. public static int MIN\_PRIORITY
2. public static int NORM\_PRIORITY
3. public static int MAX\_PRIORITY

Mức độ ưu tiên mặc định của một thread là 5 (NORM\_PRIORITY). Giá trị của MIN\_PRIORITY là 1 và giá trị MAX\_PRIORITY là 10.

### **Ví dụ về mức ưu tiên của một Thread**



Output:



# **Luồng hiểm trong java**

**Luồng hiểm trong java** (thread daemon) là một luồng cung cấp dịch vụ cho luồng người dùng. Thời gian sống của nó phụ thuộc vào các luồng người dùng. Tức là khi tất cả các luồng người dùng chết, JVM sẽ tự động chấm dứt luồng này.

Có nhiều luồng hiểm java chạy tự động, ví dụ: Gc, finalizer, v.v.

Ta có thể xem tất cả các chi tiết bằng cách gõ jconsole trong dấu nhắc lệnh. Công cụ jconsole cung cấp thông tin về các lớp nạp, sử dụng bộ nhớ, các luồng đang chạy, v.v.

## Các điểm cần nhớ về luồng hiểm trong java

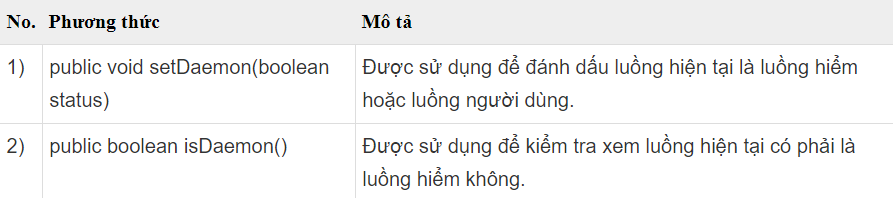
* Nó cung cấp dịch vụ cho các luồng người dùng.
* Thời gian sống của nó phụ thuộc vào các luồng người dùng.
* Nó là một luồng có mức ưu tiên thấp.

### **Tại sao JVM kết thúc luồng hiểm nếu không có luồng người dùng?**

Mục đích duy nhất của luồng hiểm là nó cung cấp dịch vụ cho luồng người dùng cho các tác vụ nền background supporting task (background supporting task). Nếu không có luồng người dùng, tại sao JVM vấn tiếp tục chạy luồng này? Đó là lý do tại sao JVM kết thúc luồng hiểm nếu không có thread người dùng nào đang được thực thi.

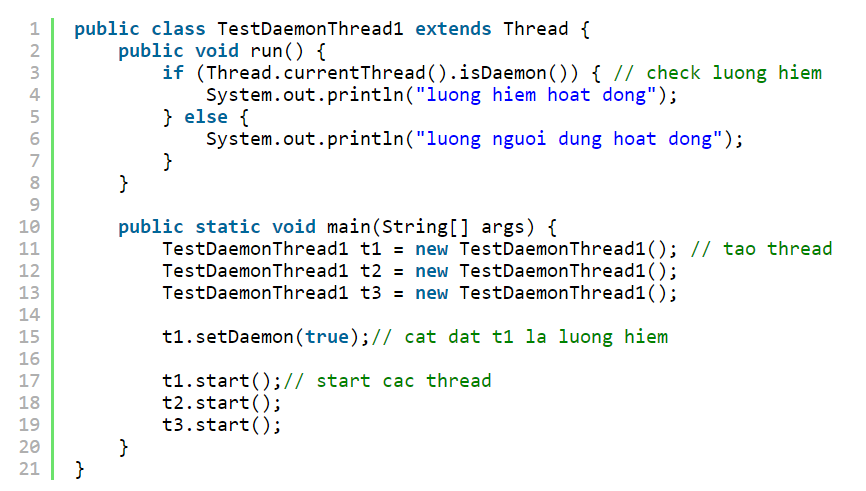
### **Các phương thức của luồng hiểm**

Lớp java.lang.Thread cung cấp hai phương thức cho luồng hiểm trong java.



## Ví dụ về luồng hiểm trong java

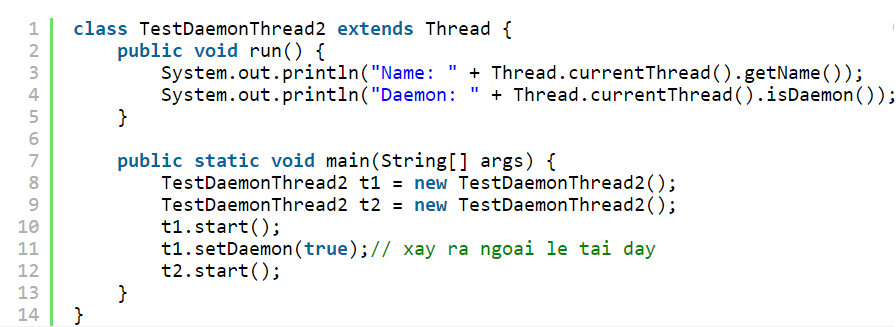
File: TestDaemonThread1.java



Output:



File: TestDaemonThread2.java



Output:



# **Thread pool trong java**

**Thread pool trong java** đại diện cho một nhóm các luồng đang chờ đợi công việc và tái sử dụng nhiều lần.

Trong trường hợp thread pool, một nhóm các thread có kích thước cố định được tạo ra. Một thread từ thread pool được kéo ra và được service provider phân công một công việc. Sau khi hoàn thành công việc, thread đó được chứa trong thread pool một lần nữa.

### **Ưu điểm của Thread Pool trong java**

**Hiệu năng tốt hơn:** Tiết kiệm thời gian vì không cần phải tạo thread mới.

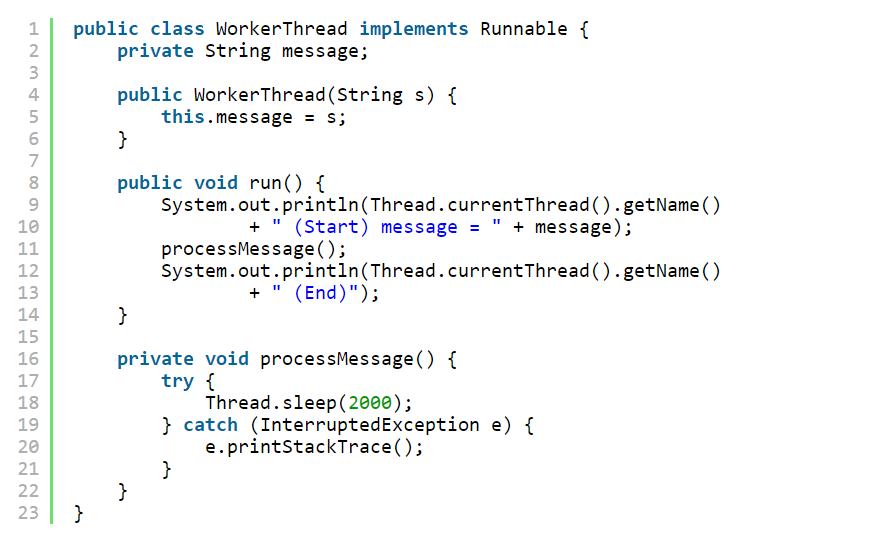
### **Sử dụng thời gian thực**

Nó được sử dụng trong Servlet và JSP, nơi container tạo một thread pool để xử lý các yêu cầu.

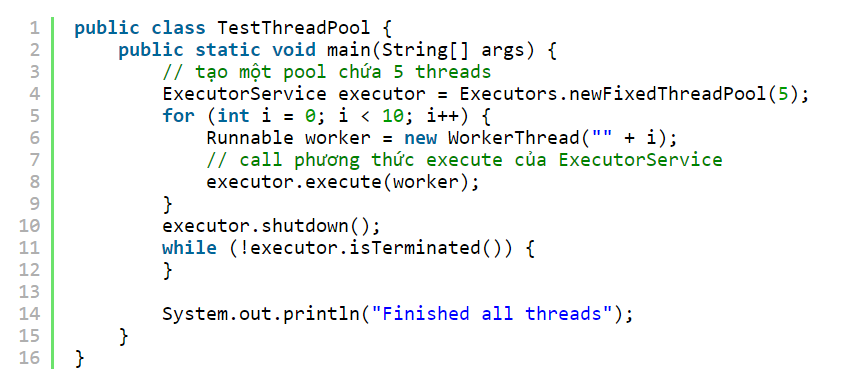
## Ví dụ về Thread Pool trong java

Dưới đây một ví dụ đơn giản về thread pool trong java sử dụng ExecutorService và Executors.

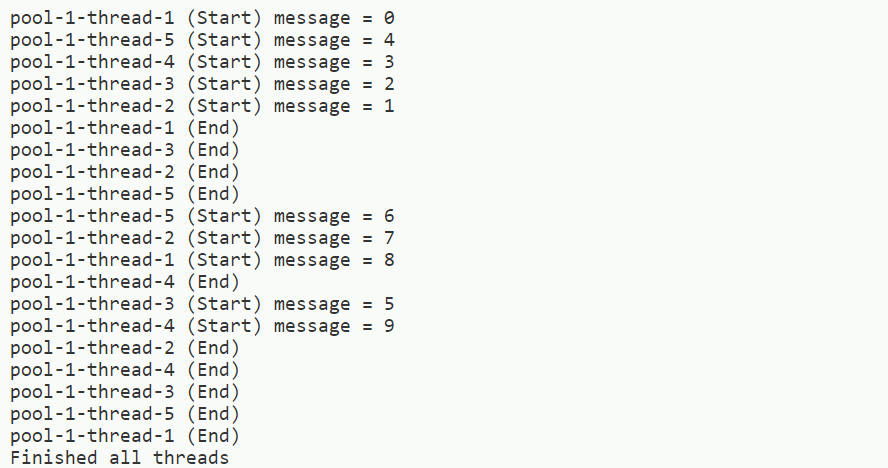
File: WorkerThread.java



File: TestThreadPool.java



Output:

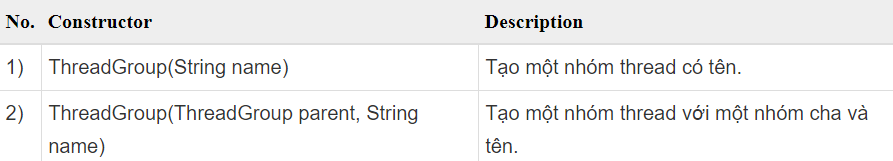


# **Thread group trong java**

Java cung cấp một cách thuận tiện để nhóm nhiều thread trong một đối tượng duy nhất. Bằng cách đó, chúng ta có thể suspend, resume hoặc interrupt một nhóm các thread bằng việc gọi một phương thức duy nhất. Thread group trong java được thực hiện bởi lớp java.lang.ThreadGroup.

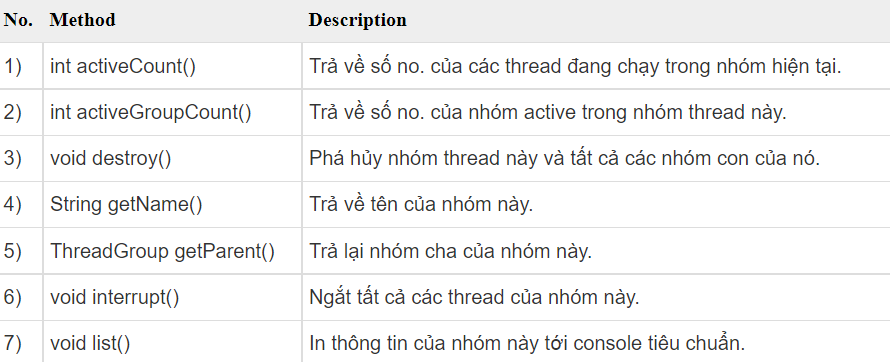
## Các constructor của lớp ThreadGroup

Chỉ có 2 constructor của lớp ThreadGroup

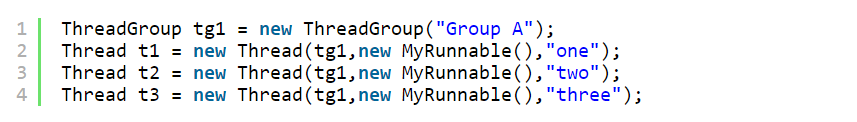


## Các phương thức quan trọng của lớp ThreadGroup

Có nhiều phương thức trong lớp ThreadGroup. Một danh sách các phương thức quan trọng được đưa ra dưới đây.



Dưới đây là code để nhóm các thread



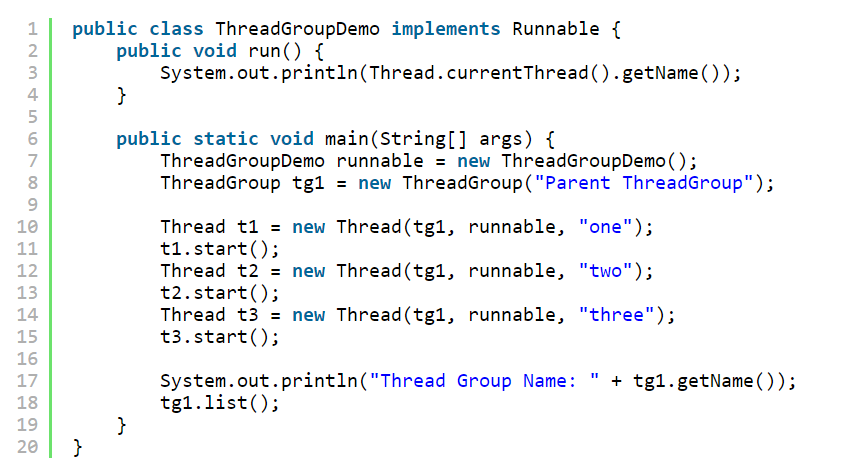
Bây giờ tất cả 3 thread thuộc về một nhóm. Ở đây, tg1 là tên của thread group, MyRunnable là lớp inplements giao tiếp Runnable và "one", "two" và "three" là các tên luồng.

Bây giờ, chúng ta có thể ngắt tất cả các thread bằng một dòng code duy nhất như sau:

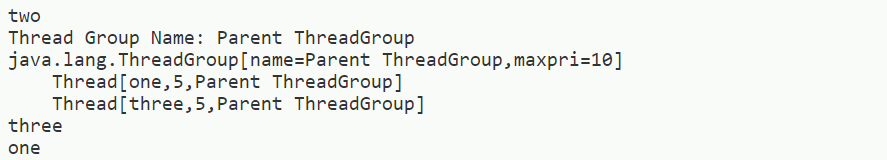


## Ví dụ về thread group trong java

File: ThreadGroupDemo.java



Output:



# **Shutdown Hook trong java**

Shutdown Hook có thể được sử dụng để thực hiện dọn sạch tài nguyên hoặc lưu trữ trạng thái khi JVM tắt bình thường hoặc đột ngột. Thực hiện dọn sạch tài nguyên có nghĩa là đóng file log, gửi một số cảnh báo hoặc cái gì đó khác. Vì vậy, nếu ta muốn thực hiện một vài dòng code trước khi JVM tắt, sử dụng shutdown hook.

## Khi nào JVM đóng lại?

JVM đóng lại khi:

* Người dụng nhấn ctrl+c trên command prompt
* Phương thức System.exit(int) được gọi
* Người dùng logoff
* Người dùng shutdown vv.

## Phương thức addShutdownHook(Thread hook)

Phương thức addShutdownHook(Thread hook) của lớp RunTime được sử dụng để đăng ký thread với Virtual Machine. Cú pháp:



Đối tượng của lớp RunTime có thể tạo bằng cách gọi phương thức static getRuntime(). Ví dụ:

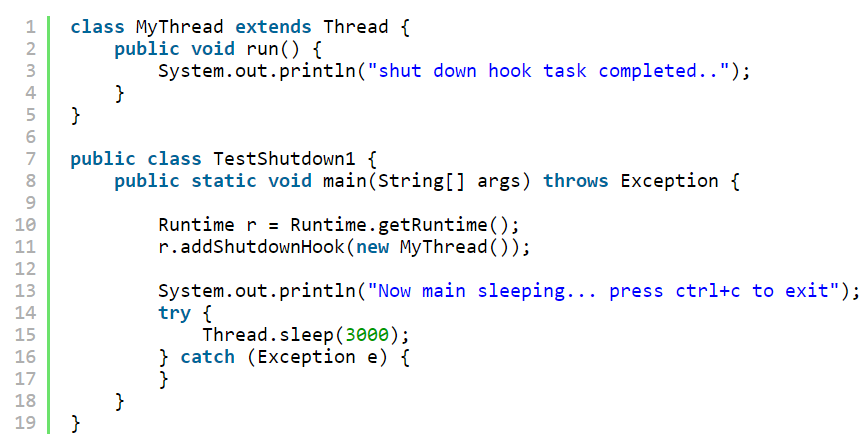


### **Phương thức Factory**

Phương thức trả về thể hiện của một lớp được gọi là phương thức factory.

## Ví dụ về Shutdown Hook trong java

File: TestShutdown1.java

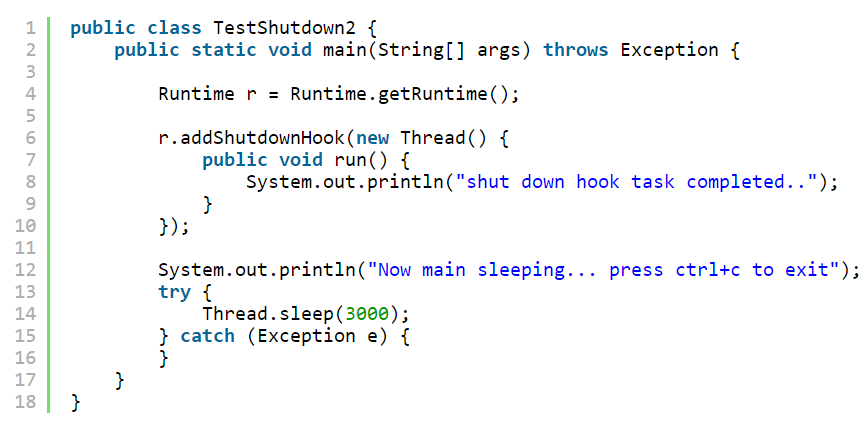


Output:



## Ví dụ tương tự về Shutdown Hook bởi lớp ẩn danh (annonymous)

File: TestShutdown2.java



Output:



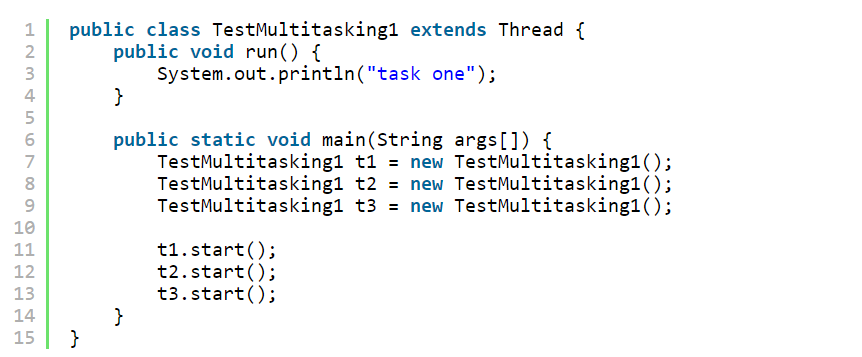
# **Thực hiện nhiều task**

## Làm thế nào để thực hiện task duy nhất bởi nhiều thread?

Nếu bạn phải thực hiện task duy nhất bởi nhiều thread, chỉ có một phương thức run(). Ví dụ:

**Chương trình thực hiện một task bằng nhiều thread**

File: TestMultitasking1.java

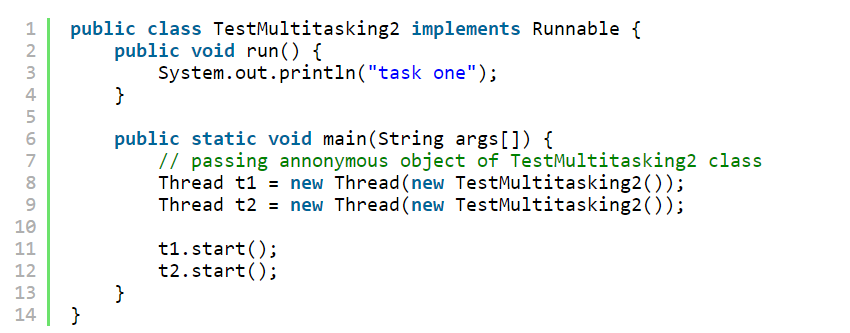


Output:



**Chương trình thực hiện một task bằng nhiều thread**

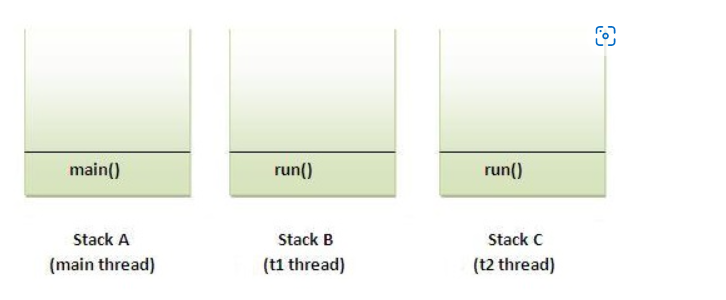
File: TestMultitasking2.java



Output:



**Chú ý: Mỗi thread chạy trong một callstack riêng biệt.**

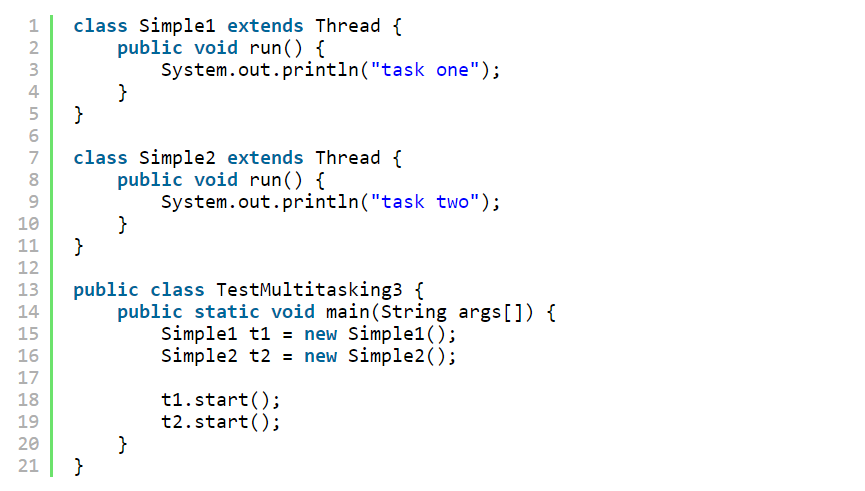


## Cách thực hiện nhiều task bằng nhiều thread

Nếu bạn phải thực hiện nhiều nhiệm vụ bởi nhiều luồng, có nhiều phương thức run(). Ví dụ:

**Chương trình thực hiện hai nhiệm vụ theo hai luồng**

File: TestMultitasking3.java



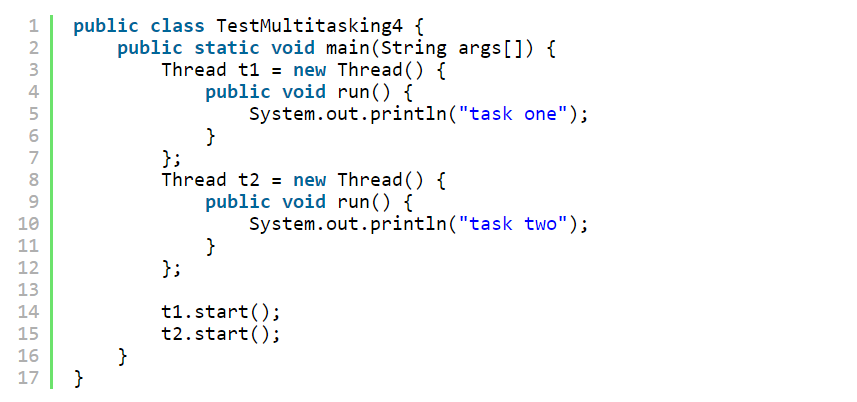
Output:



### **Ví dụ tương tự như ở trên bởi lớp nặc danh extends lớp Thread:**

**Chương trình thực hiện hai nhiệm vụ theo hai luồng**

File: TestMultitasking4.java



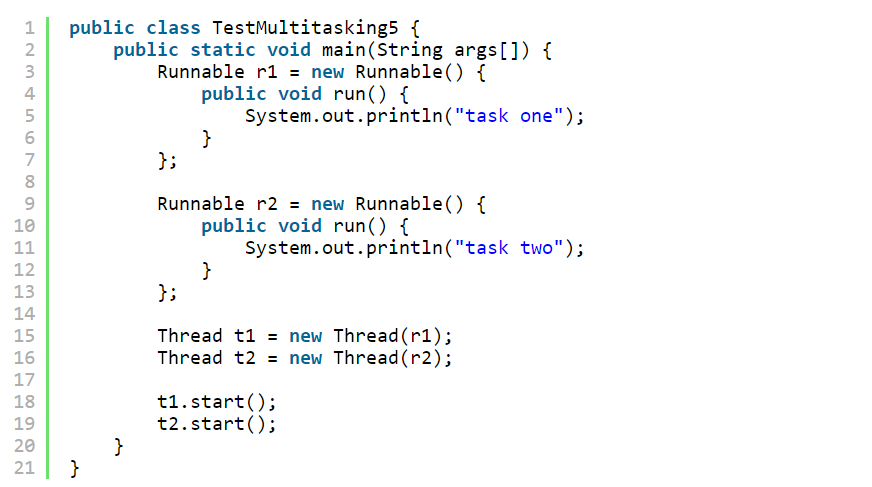
Output:



### **Ví dụ tương tự như ở trên bởi lớp nặc danh implements Runnable interface:**

**Chương trình thực hiện hai nhiệm vụ theo hai luồng**

File: TestMultitasking5.java



Output:



# **Lớp Runtime trong java**

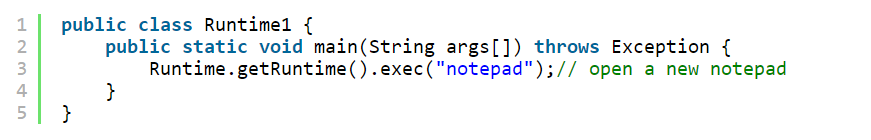
**Lớp Runtime trong java** được sử dụng để tương tác với môi trường runtime. Lớp Java Runtime cung cấp các phương thức để thực hiện một tiến trình, gọi GC, lấy tổng số và bộ nhớ free. Chỉ có một thể hiện của lớp java.lang.Runtime có sẵn cho một ứng dụng java.

Phương thức Runtime.getRuntime() trả về thể hiện duy nhất của lớp Runtime

## Các phương thức quan trọng của lớp Java Runtime



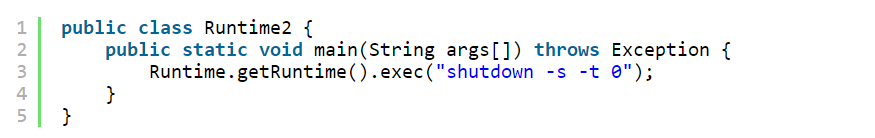
### **Phương thức exec() của lớp Java Runtime**



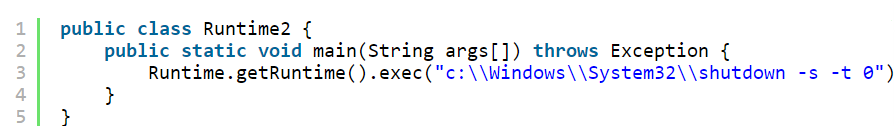
### **Làm thế nào để tắt hệ thống trong Java**

Bạn có thể sử dụng lệnh shutdown -s để tắt máy. Đối với hệ điều hành Windows, bạn cần phải cung cấp đường dẫn đầy đủ của lệnh tắt máy ví dụ: C:\\ Windows\\ System32\\shutdown.

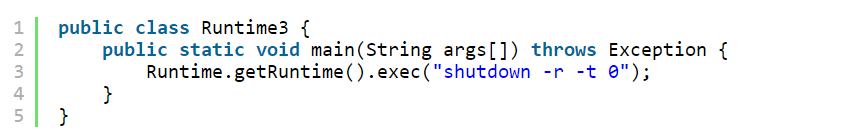
Ở đây bạn có thể sử dụng -s để tắt hệ thống, -r để khởi động lại hệ thống và -t để xác định thời gian hẹn giờ.



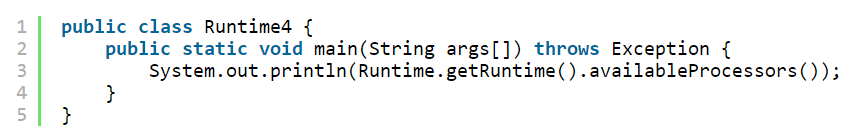
### **Tắt hệ điều hành trong java**



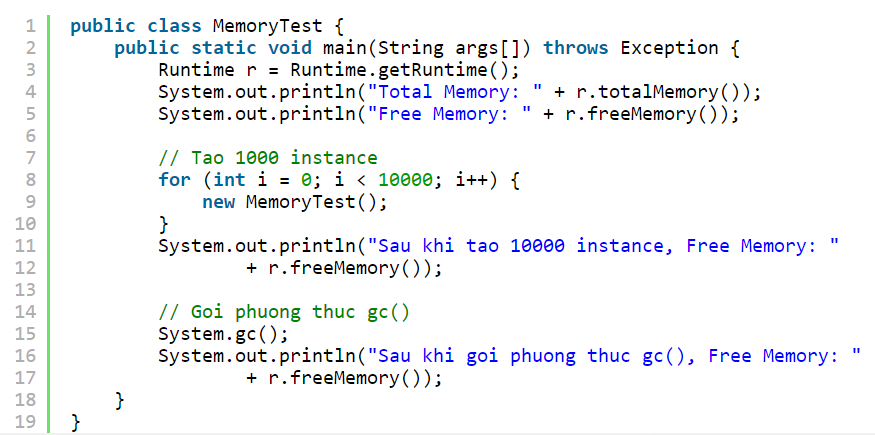
### **Khởi động lại hệ thống trong java**



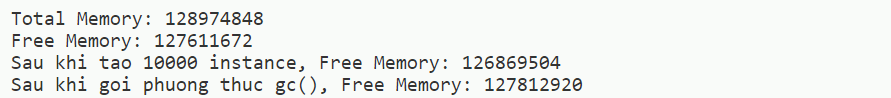
### **Phương thức availableProcessors() của lớp Runtime**



## Ví dụ về phương thức freeMemory() và totalMemory() của lớp Runtime trong java



Output:



# **Phương thức đồng bộ trong java**

## Đồng bộ trong java

**Đồng bộ trong java** (Synchronization in java) là khả năng kiểm soát truy cập của nhiều luồng đến bất kỳ nguồn tài nguyên chia sẻ (shared resource).

Giả sử có nhiều luồng muốn truy cập cùng một biến cùng một thời điểm. Ví dụ một luồng muốn đọc, trong khi luồng khác cố gắng thay đổi dữ liệu dẫn dến dữ liệu bị sai lệch. Trong trường hợp này, Java Synchronization là lựa chọn tốt khi chúng ta muốn cho phép chỉ một luồng được truy cập vào tài nguyên chia sẻ đó.

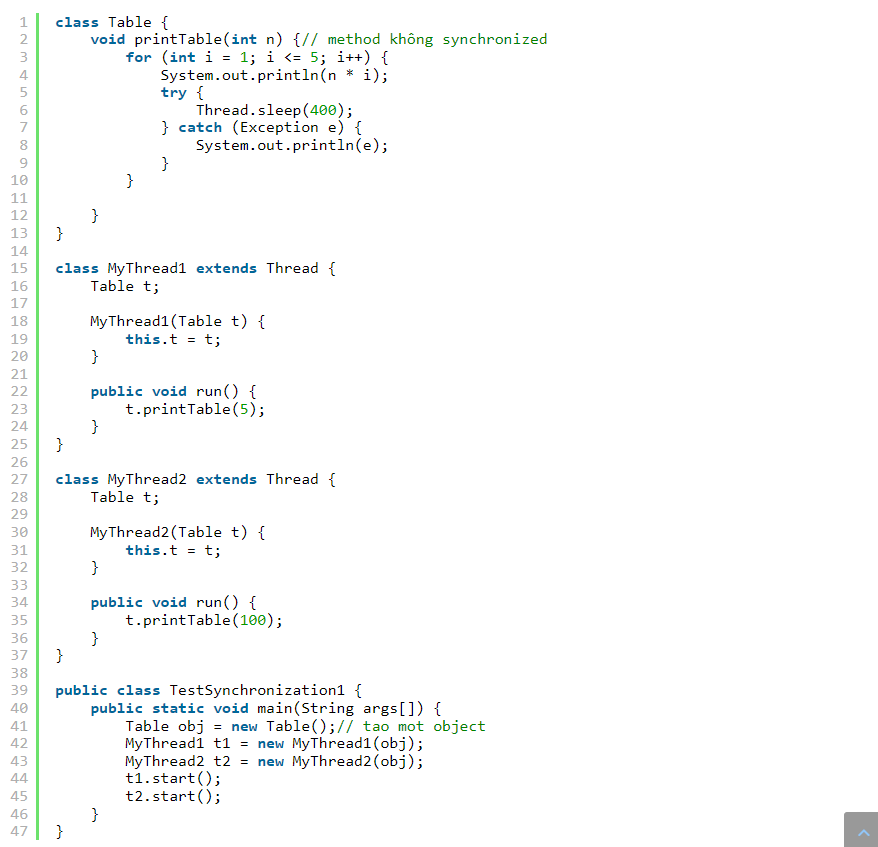
## Tại sao sử dụng Synchronization

Việc đồng bộ hóa chủ yếu được sử dụng để:

1. Để tránh sự can thiệp của luồng khác.
2. Để đảm bảo vấn đề nhất quán của chương trình.

## Vấn đề xảy ra khi không sử dụng đồng bộ

Trong ví dụ này, không có sự đồng bộ, vì vậy đầu ra không phù hợp. Hãy xem ví dụ:



Output:



## Phương thức đồng bộ (synchronized method)

Nếu bạn khai báo bất kỳ phương pháp nào với từ khóa synchronized, nó được gọi là phương thức đồng bộ.

Phương thúc đồng bộ được sử dụng để khóa một đối tượng cho bất kỳ tài nguyên được chia sẻ.

Khi một luồng gọi một phương thức đồng bộ, nó sẽ tự động khóa cho đối tượng đó và giải phóng nó khi luồng hoàn thành nhiệm vụ.

### **Ví dụ về phương thức đồng bộ (synchronized method)**

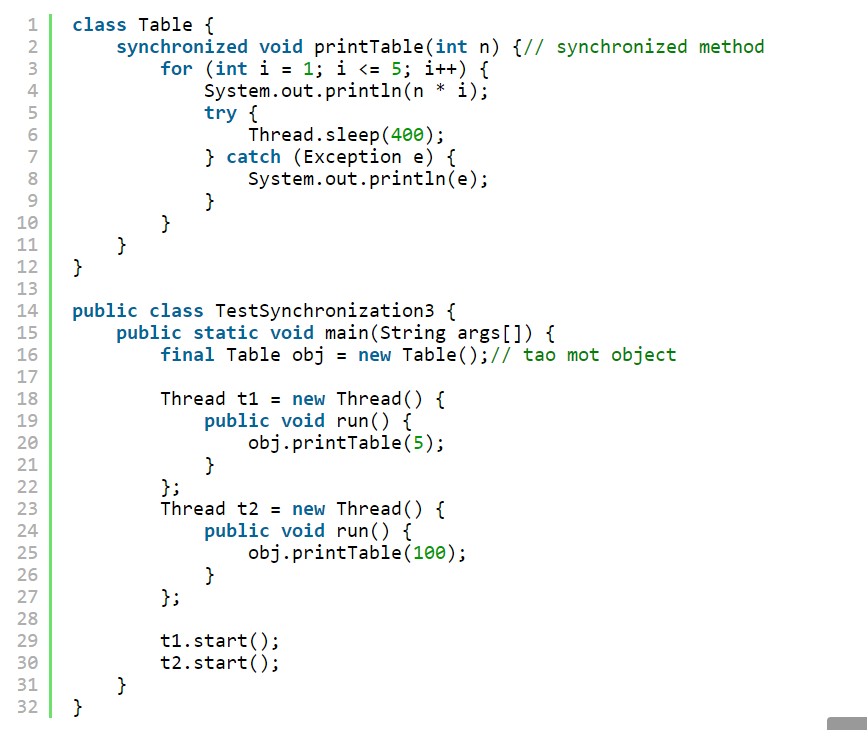


Output:

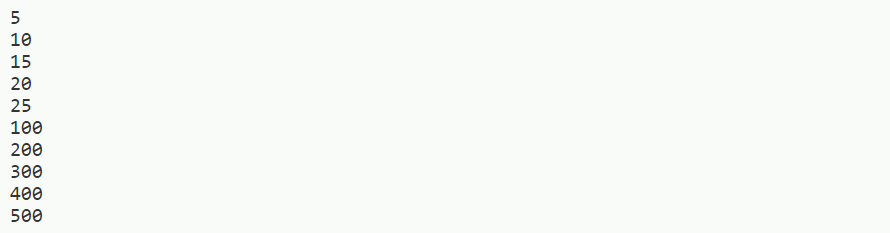


### **Ví dụ về synchronized method sử dụng lớp nặc danh**

Trong chương trình sau đây, chúng ta đã tạo ra hai thread bằng lớp ẩn danh, do đó phải viết ít code hơn.



Output:



# **Khối đồng bộ trong java**

**Khối đồng bộ trong java** (Synchronized block) có thể được sử dụng để thực hiện đồng bộ hóa trên bất kỳ tài nguyên cụ thể nào của phương thức.

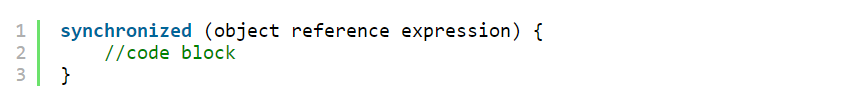
Giả sử có 50 dòng code trong phương thức, nhưng bạn chỉ muốn đồng bộ hóa 5 dòng, bạn có thể sử dụng khối đồng bộ.

Nếu bạn đặt tất cả các dòng code của phương thức trong khối đồng bộ, nó sẽ hoạt động giống như phương thức đồng bộ.

**Các điểm cần nhớ về khối đồng bộ**

* Khối đồng bộ được sử dụng để khóa một đối tượng cho bất kỳ tài nguyên dùng chung nào.
* Phạm vi của khối đồng bộ nhỏ hơn phương thức.

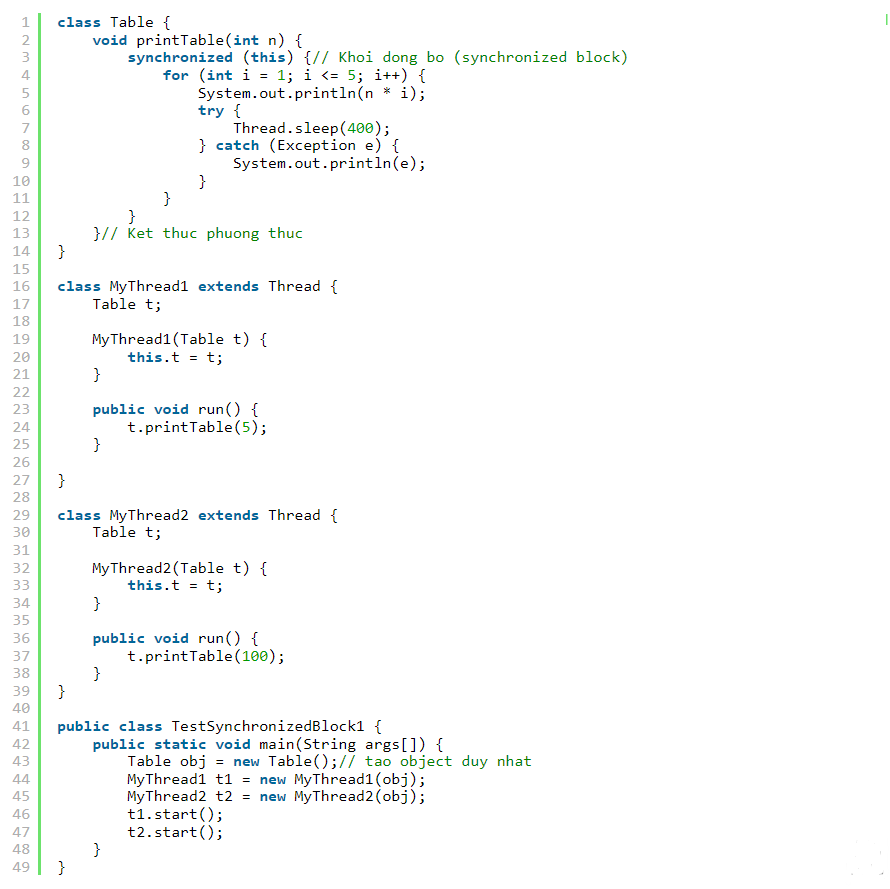
**Khai báo để sử dụng khối đồng bộ:**



## Ví dụ về khối đồng bộ

Dưới đây là ví dụ đơn giản về khối đồng bộ trong java.

File: TestSynchronizedBlock1.java



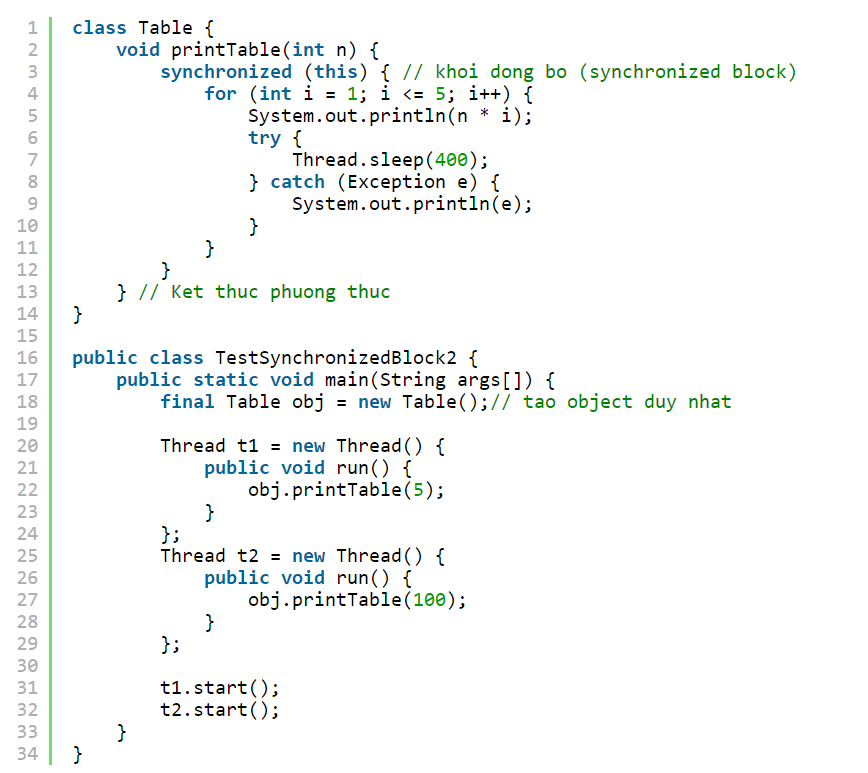
Output:



## Ví dụ tương tự về về khối đồng bộ bởi việc sử dụng lớp nặc danh

Dưới đây là ví dụ đơn giản về khối đồng bộ trong java sử dụng lớp nặc danh.

File: TestSynchronizedBlock2.java

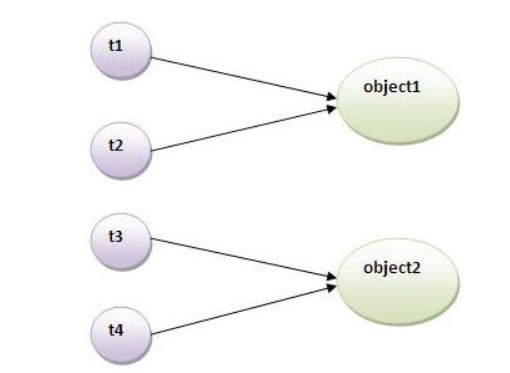


Output:



# **Phương thức đồng bộ static**

Nếu bạn tạo ra bất kỳ phương thức nào với từ khóa synchronized được gọi là **phương thức đồng bộ static**, khi đó khóa sẽ được sử dụng trên lớp không phải trên đối tượng.



**Vấn đề khi không sử dụng phương thức đồng bộ static**

Giả sử có hai đối tượng của một lớp được chia sẻ (ví dụ: Table) có tên object1 và object2. Trong trường hợp phương thức đồng bộ và khối đồng bộ không thể can thiệp giữa t1 và t2 hoặc t3 và t4 vì t1 và t2 đều tham chiếu đến một đối tượng chung có một khóa duy nhất. Nhưng có thể có sự va chạm giữa t1 và t3 hoặc t2 và t4 bởi vì t1 lấy được một khóa khác và t3 có được một khóa khác. Tôi không muốn sự xen vào giữa t1 và t3 hoặc t2 và t4. Phương thức đồng bộ hóa static được sử dụng để giải quyết vấn đề này.

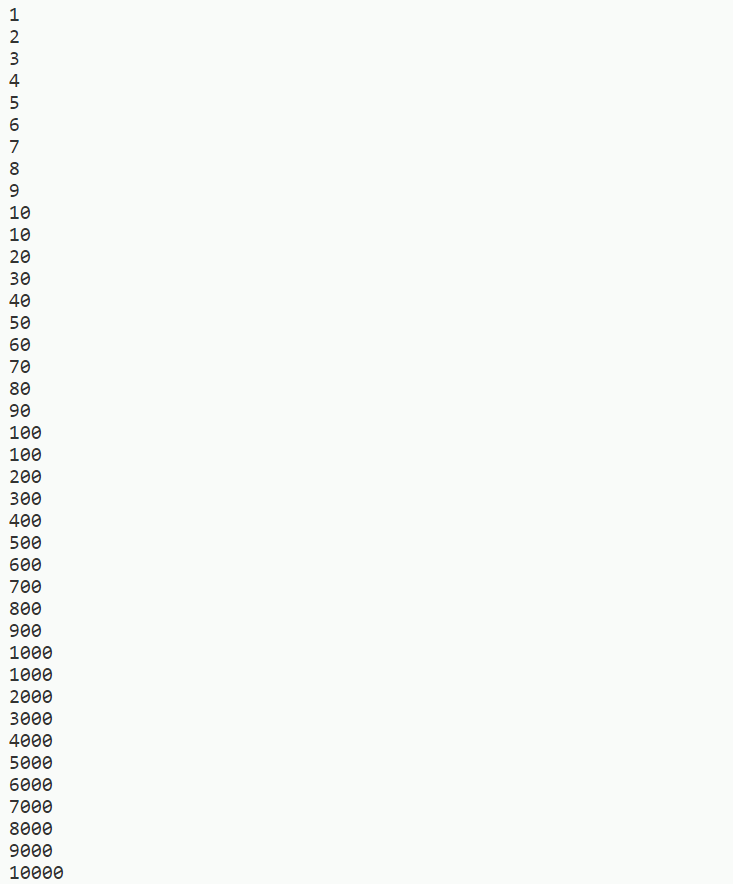
## Ví dụ về phương thức đồng bộ hóa static

Trong ví dụ này chúng ta áp dụng từ khóa synchronized trên phương pháp tĩnh để thực hiện đồng bộ hóa static.

File: TestSynchronization4.java



Output:



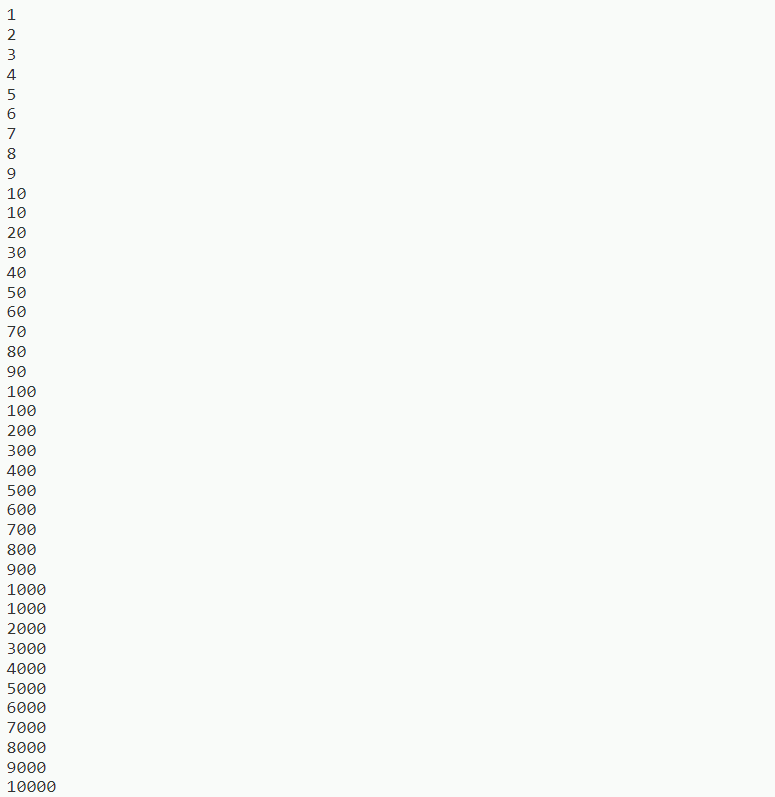
## Ví dụ về phương thức đồng bộ hóa static sử dụng lớp nặc danh

Trong ví dụ này, chúng ta sử dụng lớp nặc danh để tạo các thread.

File: TestSynchronization5.java

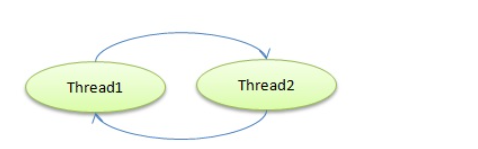


Output:

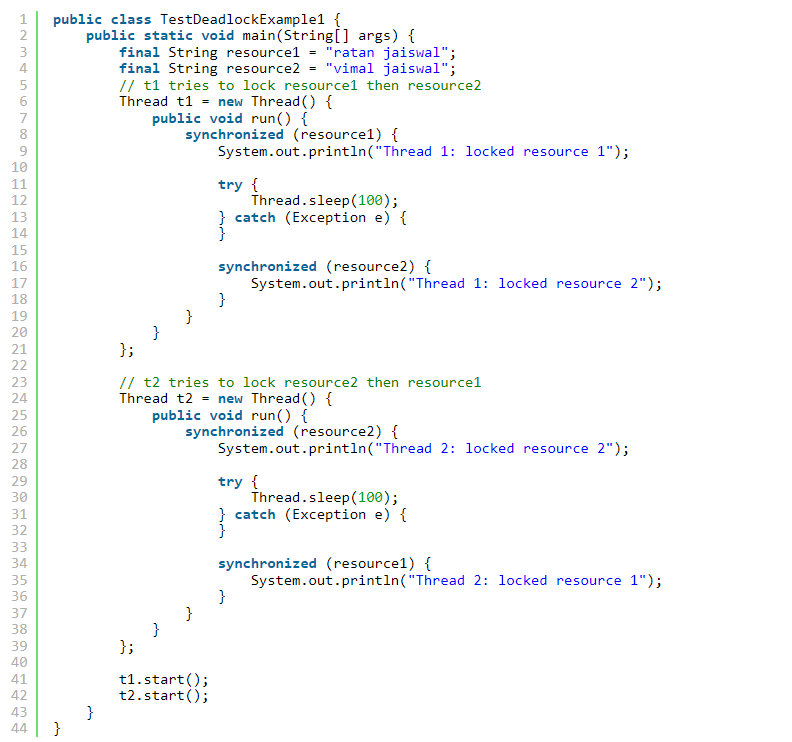


# **Deadlock trong java**

**Deadlock trong java** hay bế tắc trong java là một phần của đa luồng. Deadlock có thể xảy ra trong trường hợp một luồng đang đợi một khóa đối tượng (object lock), được sử dụng bởi một luồng khác và luồng thứ hai đang chờ đợi một khóa đối tượng (object lock) được sử dụng bởi luồng thứ nhất. Vì cả hai luồng đang chờ đợi nhau để giải phóng khóa, tình trạng này được gọi là bế tắc (deadlock).



## Ví dụ về deadlock trong java



Output:



# **Giao tiếp giữa các thread trong Java**

**Giao tiếp giữa các thread trong Java** (Inter-thread communication) là một kỹ thuật cho phép các luồng đồng bộ để giao tiếp với nhau.

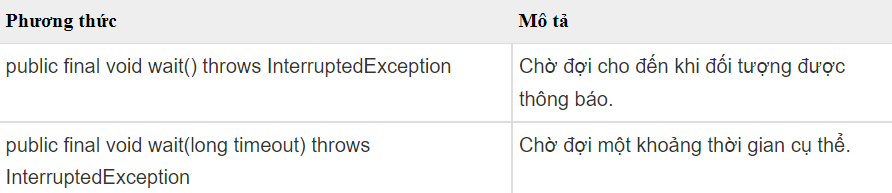
Giao tiếp giữa các thread là một cơ chế trong đó thread bị tạm dừng chạy trong session quan trọng của nó và một thread khác được phép xen vào (hoặc khóa) trong cùng một session quan trọng sẽ được thực hiện. Nó được thực hiện bằng các phương thức sau của lớp Object :

* wait()
* notify()
* notifyAll()

### **1. Phương thức wait()**

Làm cho luồng hiện tại giải phóng khóa và đợi cho đến khi một thread khác gọi phương thức notify() hoặc phương thức notifyAll() cho đối tượng này, hoặc một chờ đợi khoảng thời gian xác định.

Thread hiện tại phải sở hữu monitor của đối tượng này, vì vậy nó phải được gọi từ phương thức synchronized nếu không nó sẽ xảy ra ngoại lệ.



### **2. Phương thức notify()**

Đánh thức luồng duy nhất đang chờ trên monitor của đối tượng này. Nếu có nhiều luồng nào đó đang chờ đối tượng này, một trong số chúng sẽ được đánh thức. Sự lựa chọn là độc đoán và xảy ra theo quyết định của bản cài đặt. Cú pháp:

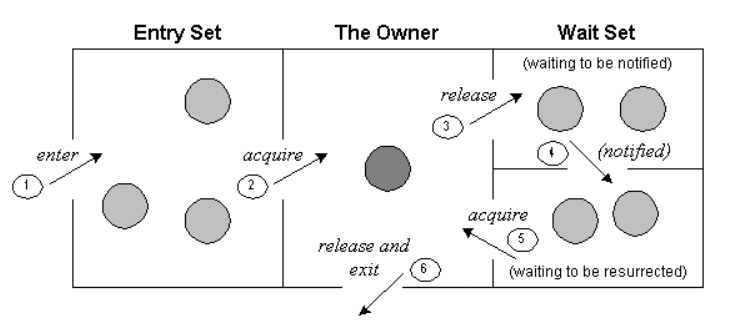


### **3. Phương thức notifyAll()**

Đánh thức tất cả các luồng đang chờ trên monitor của đối tượng này. Cú pháp:



### **Quá trình giao tiếp giữa các thread trong java**



Giải thích sơ đồ trên

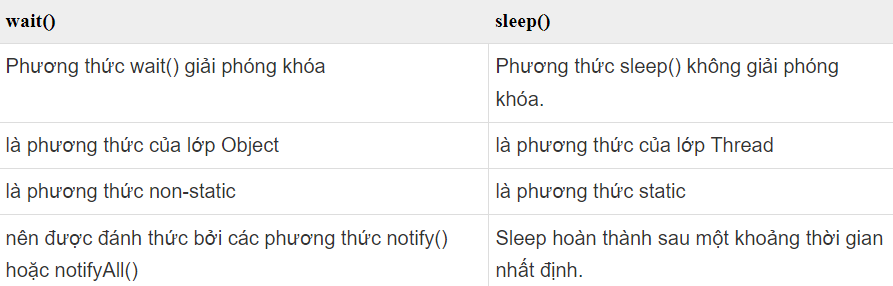
1. Luồng nhập vào để có được khóa.
2. Khóa được chiếm bởi một luồng.
3. Bây giờ luồng đi vào trạng thái chờ nếu bạn gọi phương thức wait() trên đối tượng. Nếu không, nó sẽ giải phóng khóa và thoát ra.
4. Nếu bạn gọi notify() hoặc notifyAll(), luồng sẽ chuyển đến trạng thái được thông báo (trạng thái chạy được)
5. Hiện luồng có sẵn để có được khóa.
6. Sau khi hoàn thành nhiệm vụ, luồng sẽ giải phóng khóa và thoát khỏi trạng thái monitor của đối tượng.

### **Tại sao các phương thức wait(), notify() và notifyAll() được định nghĩa trong lớp Object mà không phải lớp Thread?**

Đó là bởi vì chúng liên quan đến khóa và đối tượng có một khóa.

## Sự khác nhau giữa wait và sleep

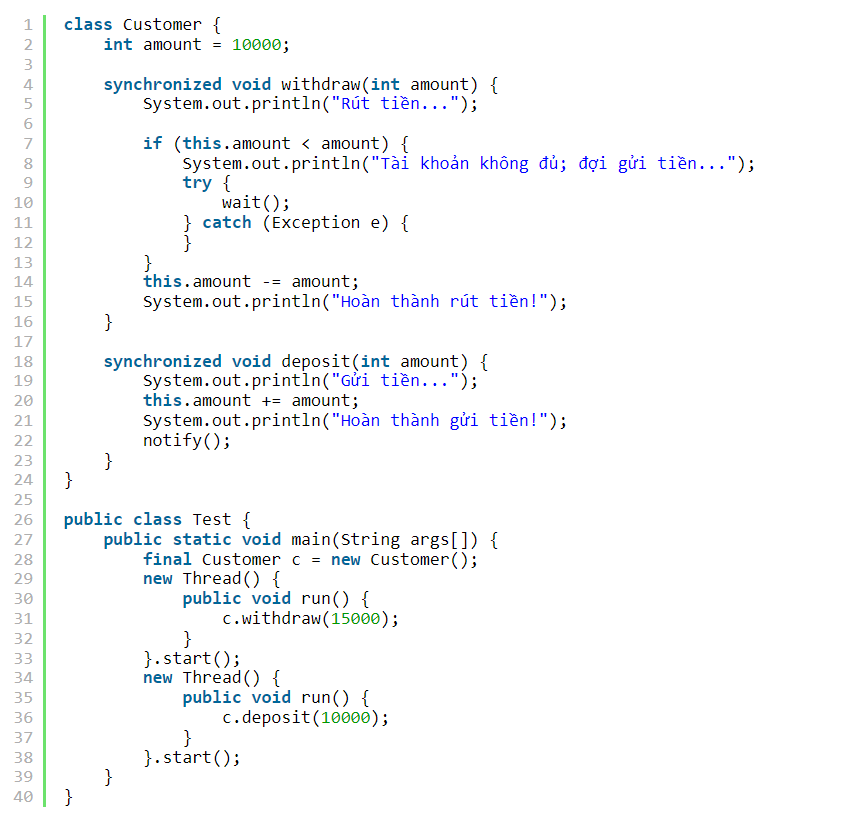
Hãy xem sự khác biệt quan trọng giữa phương thức wait và sleep.



## Ví dụ về giao tiếp giữa các thread trong java

Hãy xem ví dụ đơn giản về giao tiếp giữa các thread trong java như dưới đây.

File: Test.java



Output:



# **Interrupt một thread trong java**

**Interrupt một thread trong java** hay làm gián đoạn một luồng trong java. Nếu thread nằm trong trạng thái sleep hoặc wait (nghĩa là sleep() hoặc wait() được gọi ra), việc gọi phương thức interrupt() trên thread đó sẽ phá vỡ trạng thái sleep hoặc wait và ném ra ngoại lệ InterruptedException. Nếu thread không ở trong trạng thái sleep hoặc wait, việc gọi phương thức interrupt() thực hiện hành vi bình thường và không làm gián đoạn thread nhưng đặt cờ interrupt thành true. Trước hết chúng ta sẽ quan tâm đến các phương thức được cung cấp bởi lớp Thread để làm gián đoạn một luồng.

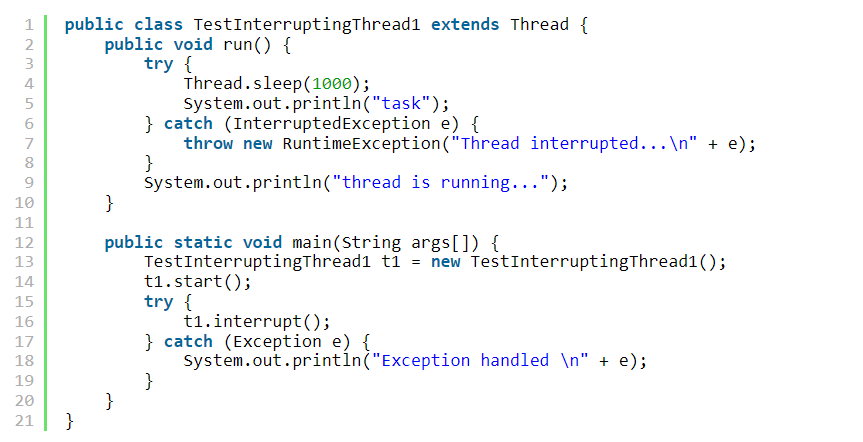
### **3 phương thức được cung cấp bởi lớp Thread để làm gián đoạn một luồng trong java**

* **public void interrupt()**
* **public static boolean interrupted()**
* **public boolean isInterrupted()**

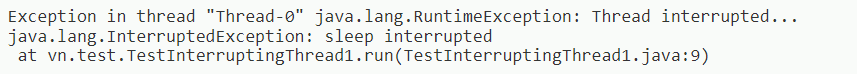
## Ví dụ về interrupt một thread trong java khiến thread ngừng hoạt động

Trong ví dụ này, sau khi gọi phương thức interrupt() để làm gián đoạn thread, sẽ xảy ra ngoại lệ InterruptedException, dẫn đến thread sẽ ngừng hoạt động. Nếu chúng ta không muốn dừng thread, chúng ta có thể xử lý tại nơi phương thức sleep() hoặc wait() được gọi ra.

File: TestInterruptingThread1.java

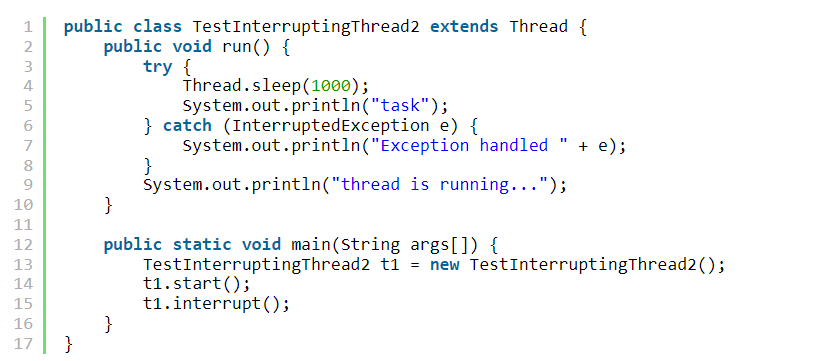


Output:



## Ví dụ về interrupt một thread trong java mà thread không ngừng hoạt động

Trong ví dụ này, sau khi làm gián đoạn thread, chúng ta xử lý ngoại lệ, vì vậy nó sẽ phá vỡ sleep nhưng thread sẽ không ngừng hoạt động.



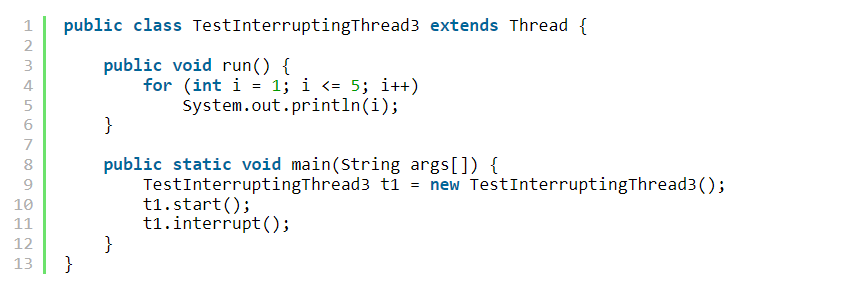
Output:



## Ví dụ về interrupt một thread trong java mà thread hoạt động bình thường

Nếu thread không ở trạng thái sleep hoặc wait, việc gọi phương thức interrupt() sẽ thiết lập cờ interrupted thành true mà về sau có thể được sử dụng để dừng thread bởi lập trình viên.

File: TestInterruptingThread3.java



Output:

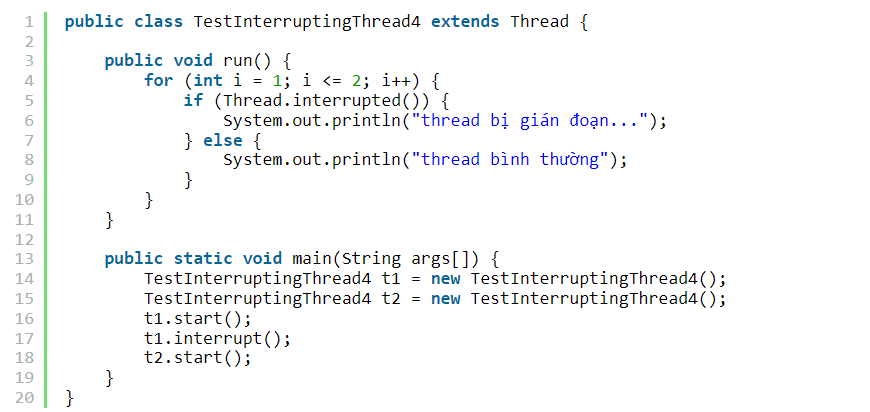
## Các phương thức isInterrupted và interrupted

Phương thức isInterrupted() trả về cờ interrupted là true hoặc false. Phương thức static interrupted() trả về cờ interrupted sau đó nó đặt cờ thành false nếu nó là true.

## Ví dụ về interrupt một thread trong java mà thread hoạt động bình thường

Nếu thread không ở trạng thái sleep hoặc wait, việc gọi phương thức interrupt() sẽ thiết lập cờ interrupted thành true mà về sau có thể được sử dụng để dừng thread bởi lập trình viên.

File: TestInterruptingThread3.java



Output:



# Garbage Collection trong Java

Trong Java, **rác** (garbage) có nghĩa là các đối tượng không còn được tham chiếu và **bộ thu gom rác** (garbage collection) được sử dụng để thực hiện quá trình tự động khôi phục lại bộ nhớ. Nói cách khác, đó là một cách để phá hủy các đối tượng không sử dụng nữa.

Vậy Garbage Collection trong Java có điểm gì đáng chú ý? Hãy cùng Quản Trị Mạng tìm hiểu về khái niệm của Garbage Collection, cách thức hoạt động và tại sao nó lại quan trọng trong bài viết này nhé!

## 1. Khái niệm về Garbage Collection

**Garbage Collection** (Bộ thu gom rác) trong Java được định nghĩa như một quá trình tự động thực thi nhiệm vụ quản lý bộ nhớ.

Code Java được dịch sang bytecode rồi chạy trên máy ảo Java hay viết tắt là JVM. Trong quá trình chạy chương trình, các đối tượng được tạo ở vùng nhớ heap, một phần bộ nhớ dành cho chương trình. Sau cùng, sẽ có một vài đối tượng mà chương trình không cần dùng đến. Các đối tượng này sẽ được garbage collector truy tìm và xóa bỏ để thu hồi lại dung lượng bộ nhớ.

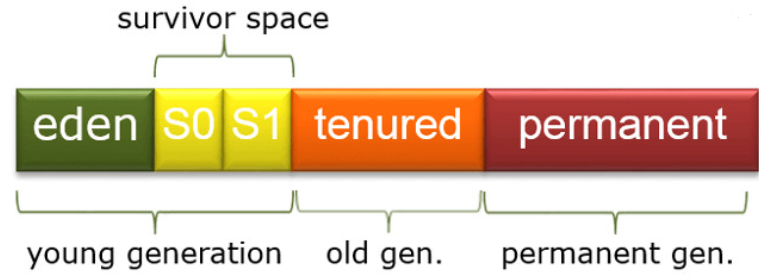
Không gian trống này sẽ được cấp phát cho những đối tượng mới. Với các ngôn ngữ lập trình như C, việc giải phóng bộ nhớ được thực hiện một cách thủ công (bằng những lệnh khởi tạo, giải phóng bộ nhớ). Với Java, việc giải phóng bộ nhớ được thực hiện một cách tự động.

## 2. Garbage collection trong Java hoạt động như thế nào?

Garbage Collection trong Java là một quá trình hoàn toàn tự động. Các lập trình viên không cần gọi các lệnh "dọn" bộ nhớ như trong ngôn ngữ lập trình C/ C++. Phần hệ thống xử lý của garbage collection nằm trong JVM. Mỗi JVM lại có một cách cài đặt bộ dọn rác khác nhau, phù hợp với những đặc tính của JVM đó. Trong số các đại diện JVM, Oracle HotSpot sẽ được chúng tôi lấy ra làm ví dụ vì nó đang được sử dụng phổ biến nhất và bộ dọn rác của HotSpot có nhiều tính năng mạnh mẽ hơn so với những người anh em khác.

HotSpot có nhiều bộ dọn rác nhưng tựu chung lại các bộ dọn rác này đều hoạt động theo mô hình chung. Đầu tiên, xác định và đánh dấu các đối tượng không có tham chiếu (unreferenced object) sẵn sàng để thu gom rác. Bước thứ hai, xóa các đối tượng đã được đánh dấu. Tùy chọn, bộ nhớ có thể được "nén" lại sau khi bộ garbage collector tiến hành xóa các đối tượng, điều này đồng nghĩa với việc các đối tượng đang hoạt động sẽ nằm ở các ô nhớ sát nhau tại phần bắt đầu của heap. Quá trình "nén" này giúp cấp phát bộ nhớ cho các đối tượng mới dễ dàng hơn sau khi khối bộ nhớ được phân bổ cho các đối tượng hiện có.

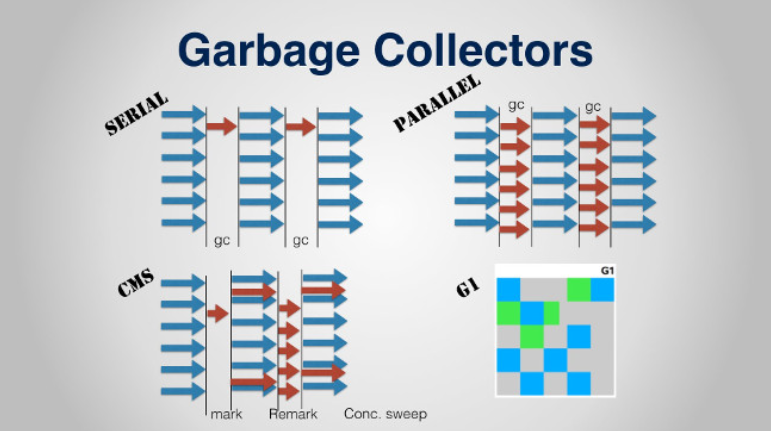
Tất cả các garbage collector của HotSpot thực hiện theo một chiến thuật chung là chia đối tượng theo "tuổi hoạt động". Lý do đằng sau việc chia bộ thu dọn rác theo tuổi hoạt động là vì hầu hết các vật thể đều tồn tại trong thời gian ngắn và sẽ sẵn sàng thu gom ngay sau khi tạo ra.



Theo biểu đồ bên trên, ta thấy tuổi hoạt động của các đối tượng được chia thành 3 nhóm tuổi: Young generation - Thế hệ trẻ, Old generation - Thế hệ già và Permanent generation - Thế hệ bất tử.

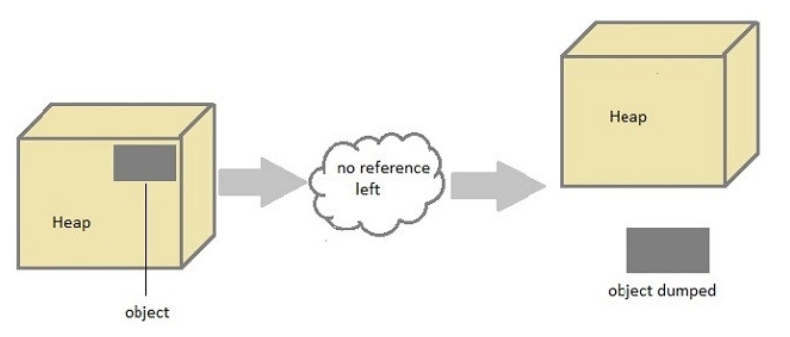
* **Young generation - Thế hệ trẻ**: Nhóm này được chia thành 2 nhóm con là Eden (Khởi thủy) và Survivor (Sống sót). Trong nhóm Survivor lại được chia thành 2 nhóm nhỏ hơn là S0 và S1. Ban đầu, các đối tượng mới được khởi tạo sẽ nằm trong nhóm Eden. Sau một chu kỳ hoạt động của garbage collector, đối tượng nào "sống sót" sẽ được chuyển sang nhóm Survivor. Sự kiện các đối tượng ở nhóm Young generation được thu hồi bởi Garbage collector được xem là Minor event. Sau "nhiều" chu kỳ quét mà đối tượng vẫn còn được sử dụng thì chúng mới được chuyển sang vùng nhớ Old generation.
* **Old generation - Thế hệ già**: Nhóm này chứa các đối tượng chuyển từ Young generation (tất nhiên với thời gian hoạt động đủ lâu, mỗi bộ garbage collector sẽ định nghĩa bao nhiêu được coi là "lâu"). Sự kiện các đối tượng ở nhóm Old generation được thu hồi bởi garbage collector được xem là Major event.
* **Permanent generation - Thế hệ bất tử**: Nhóm này gồm metadata (ví dụ: các class, method,...). Do đó, khi phải "dọn" các class, method không cần thiết, garbage collector sẽ tìm kiếm trong nhóm này.

Trong một sự kiện thu gom rác, các vật dụng không sử dụng trong tất cả các thế hệ là rác thải thu được. HotSpot chứa 4 bộ Garbage collector khác nhau:



1. **Serial**: Các event thu gom rác được xử lý tuần tự trên một thread. Quá trình dồn bộ nhớ (compaction) được thực thi sau mỗi event thu gom rác. Serial là bộ garbage collector đơn giản nhất, được thiết kế cho môi trường single thread. Khi Serial hoạt động, ứng dụng buộc phải dừng lại. Do đó, nó không phù hợp với môi trường server.
2. **Parallel**: Parallel là garbage collector mặc định của Java. Minor event sẽ được xử lý trên nhiều thread. Major event và quá trình dồn bộ nhớ cho nhóm Old generation được xử lý trên một thread. Bên cạnh bộ Parallel, có một bộ khác tên là Parallel Old xử lý Major event và quá trình dồn bộ nhớ (cho nhóm Old generation) trên nhiều thread. Tuy nhiên, khi hoạt động Parallel, nó sẽ dừng thread chạy chương trình.
3. **Concurrent Mark Sweep (CMS)**: CMS là sự kết hợp và cải tiến của Parallel và Parallel Old. Nó xử lý minor event trên nhiều thread (giống với Parallel) và xử lý major event trên nhiều thread (giống với Parallel Old). Bên cạnh đó, CMS chạy song song với ứng dụng và đảm bảo quá trình dọn rác không làm ảnh hướng tới quá trình thực thi ứng dụng. CMS không tiến hành dồn bộ nhớ. CMS tiêu tốn nhiều tài nguyên CPU hơn nhưng lại không làm ảnh hưởng tới quá trình thực thi ứng dụng (hay còn gọi là trạng thái Stop The World - STW). Đối với các server hoặc các ứng dụng gặp bất lợi khi phải STW thì sử dụng CMS là lựa chọn phù hợp.
4. **Garbage First (G1)**: Đây là bộ garbage collector mới nhất, ra đời cùng Java 7 với mục tiêu thay thế cho CMS trong việc quản lý các vùng heap > 4GB. G1 sử dụng nhiều background thread để scan qua vùng heap (được chia thành các vùng có dung lượng từ 1 đến 32MB). G1 sẽ thu dọn ở các vùng nhớ có nhiều "rác" nhất. G1 có khả năng vừa tiến hành thu hồi vừa dồn bộ nhớ, còn CMS chỉ có thể dồn bộ nhớ ở trong trạng thái STW. Hơn nữa, G1 có khả năng nhận diện các xâu ký tự trùng nhau trong heap (được tham chiếu bởi các đối tượng khác nhau) và sửa tham chiếu nhằm tránh các bản copy thừa của các xâu ký tự, tiết kiệm dung lượng trống cho vùng nhớ heap.

## 3. Lợi ích của garbage collection trong Java



Lập trình C/ C++ chắc hẳn đã từng gặp khó khăn khi đương đầu với Memory Leaks. Tuy nhiên, với Java nói riêng và các ngôn ngữ sở hữu Garbage Collector, cực hình này đã được chấm dứt hoàn toàn.

Trong khi giới lập trình vẫn đang tranh cãi về khả năng hoạt động hiệu quả của Garbage Collector và phe bảo thủ một mực khẳng định rằng việc quản lý bộ nhớ bằng tay sẽ mang lại khả năng kiểm soát và hiệu năng tốt hơn thì Garbage Collector đang trở thành tính năng "chuẩn" của nhiều ngôn ngữ lập trình mới.

Đối với Java, khi quá trình hoạt động của garbage collector có nguy cơ ảnh hưởng tới performance của ứng dụng, lập trình viên có thể tùy chỉnh garbage collector theo nhiều cách khác nhau để đạt được hiệu năng mong muốn.

## 4. Làm việc với garbage collector sao cho hiệu quả?



Với các ứng dụng đơn giản, lập trình viên không cần quan tâm nhiều đến garbage collector. Tuy nhiên, nếu họ muốn nâng cao khả năng của bản thân thì việc hiểu cơ chế hoạt động và tùy chỉnh bộ garbage collector là một trong các kỹ năng phải học.

Bên cạnh cơ chế hoạt động của garbage collector, các lập trình viên cần lưu ý một điều, đó là việc thu gom rác ở Java không xác định được và không thể dự đoán garbage collector sẽ chạy ở thời điểm nào. Kể cả khi gọi nó một cách tường minh với System.gc() hay Runtime.gc(), ta vẫn không thể chắc chắn được garbage collector có chạy hay không.

Về việc tùy chỉnh garbage collector, cách tiếp cận tốt nhất là điều chỉnh các setting flag của JVM (chính là các tham số tương ứng với từng bộ garbage collector được liệt kể ở trên). Flag có thể điều chỉnh bộ thu gom rác (ví dụ: Serial, G1, v.v.), quy định kích thước khi cấp phát và kích thước tối đa của vùng nhớ heap mà chương trình sử dụng, hoặc điều chỉnh kích thước của từng nhóm "tuổi". Ví dụ: bộ garbage collector Parallel có hiệu quả nhưng sẽ gây ra sự cố "stop the world", làm cho nó phù hợp hơn cho việc xử lý phụ trợ, nơi được chấp nhận tạm dừng để thu gom rác.

Mặt khác, bộ garbage collector CMS được thiết kế để giảm thiểu thời gian tạm dừng, làm cho nó trở nên lý tưởng cho các ứng dụng GUI, nơi đáp ứng quan trọng. Điều chỉnh bổ sung có thể được thực hiện bằng cách thay đổi kích thước heap hoặc các phần của nó và đo hiệu quả thu gom rác bằng cách sử dụng một công cụ như jstat.

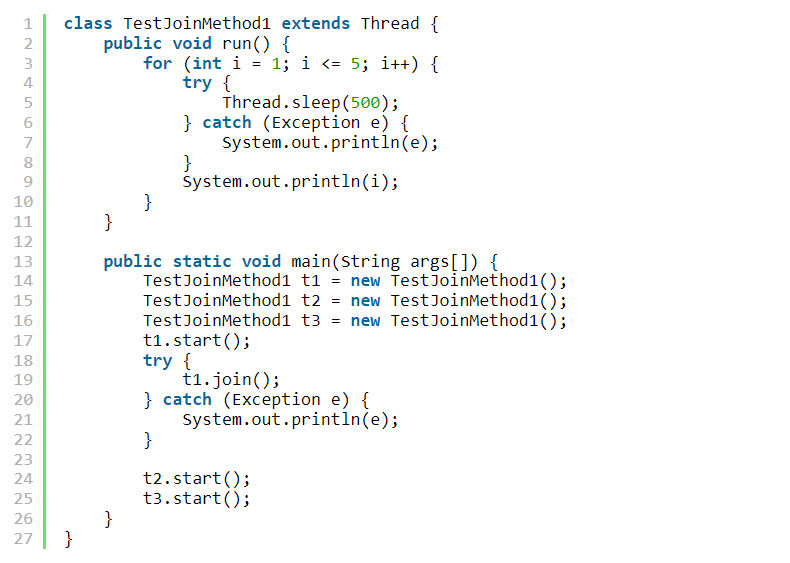
# **Phương thức join() - Java Thread**

Phương thức join() chờ một thread chết. Nói cách khác, nó làm cho các thread đang chạy ngừng hoạt động cho đến khi luồng mà nó tham gia hoàn thành nhiệm vụ của nó.

## Các dạng phương thức join()

* public void join()throws InterruptedException
* public void join(long milliseconds)throws InterruptedException

**Ví dụ về phương thức join()**

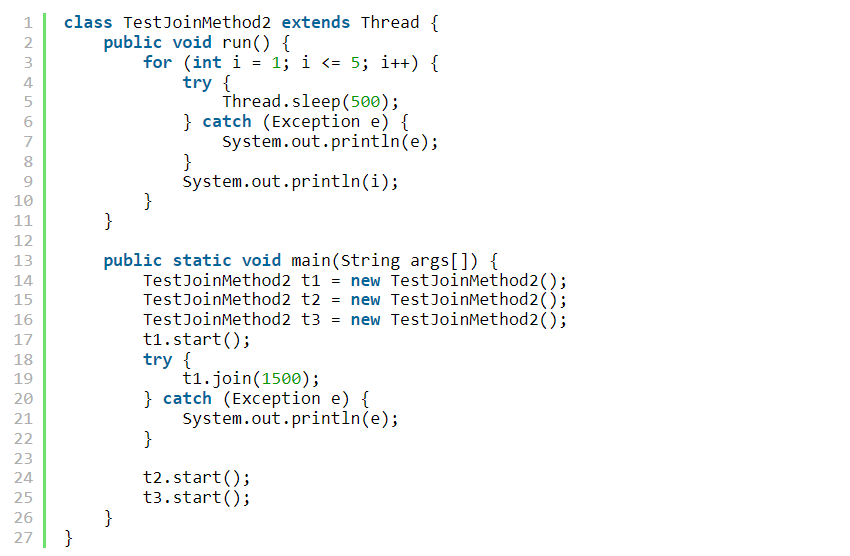


Output:



Như ta thấy trong ví dụ trên, khi t1 hoàn thành nhiệm vụ của nó thì t2 và t3 bắt đầu thực thi.

**Ví dụ về phương thức join(long miliseconds)**



Output:

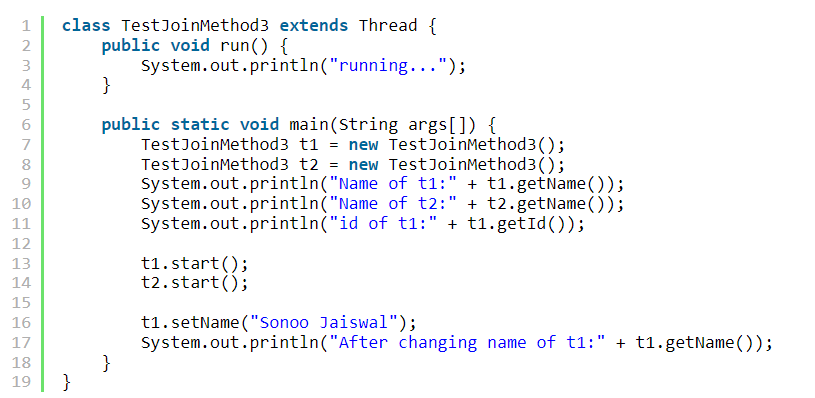


## Các phương thức getName(),setName(String) và getId()

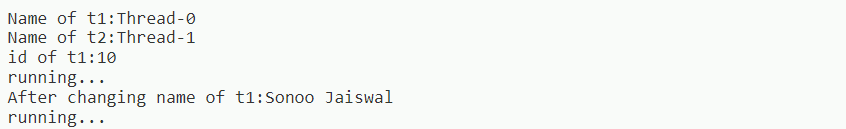
public String getName()

public void setName(String name)

public long getId()



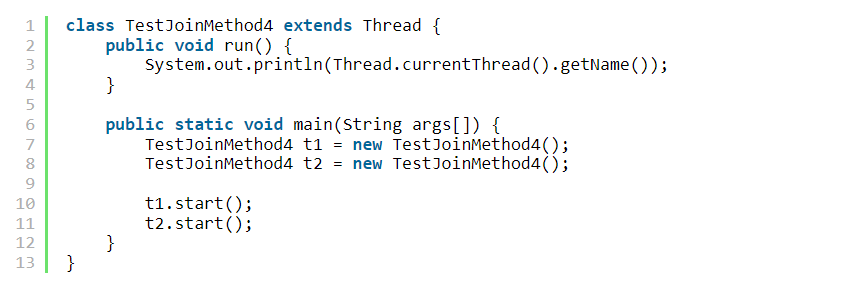
Output:



## Phương thức currentThread()

Phương thức currentThread() trả về một tham chiếu đến đối tượng thread hiện đang thực thi.

**Ví dụ về phương thức currentThread()**



Output:



[Garbage Collection trong Java có gì đáng chú ý? - Bộ thu gom rác trong Java (quantrimang.com)](https://quantrimang.com/garbage-collection-trong-java-co-gi-dang-chu-y-147845)

[Phương thức join() - học Java miễn phí hay nhất - VietTuts](https://viettuts.vn/java-thread/phuong-thuc-join-java-thread)

[Thread trong java - học Java miễn phí hay nhất - VietTuts](https://viettuts.vn/java-thread)