

# ĐA TUYẾN

## Mục tiêu:

Sau khi kết thúc chương này, bạn có thể:

- Định nghĩa một luồng
- Mô tả đa tuyến
- Tạo và quản lý luồng
- Hiểu được vòng đời của luồng
- Mô tả một luồng hiem
- Giải thích tập hợp các luồng ưu tiên như thế nào
- Giải thích được sự cần thiết của sự đồng bộ
- Hiểu được cách thêm vào các từ khoá synchronized (đồng bộ) như thế nào
- Liệt kê những điều không thuận lợi của sự đồng bộ
- Giải thích vai trò của các phương thức wait() (đợi), notify() (thông báo) và notifyAll().
- Mô tả một điều kiện bế tắc (deadlock).

## 1. Giới thiệu

Một luồng là một thuộc tính duy nhất của Java. Nó là đơn vị nhỏ nhất của đoạn mã có thể thi hành được mà thực hiện một công việc riêng biệt. Ngôn ngữ Java và máy ảo Java cả hai là các hệ thống được phân luồng

## 2. Đa tuyến

Java hỗ trợ đa tuyến, mà có khả năng làm việc với nhiều luồng. Một ứng dụng có thể bao hàm nhiều luồng. Mỗi luồng được đăng ký một công việc riêng biệt, mà chúng được thực thi đồng thời với các luồng khác.

Đa tuyến giữ thời gian nhàn rỗi của hệ thống thành nhỏ nhất. Điều này cho phép bạn viết các chương trình có hiệu quả cao với sự tận dụng CPU là tối đa. Mỗi phần của chương trình được gọi một luồng, mỗi luồng định nghĩa một đường dẫn khác nhau của sự thực hiện. Đây là một thiết kế chuyên dùng của sự đa nhiệm.

Trong sự đa nhiệm, nhiều chương trình chạy đồng thời, mỗi chương trình có ít nhất một luồng trong nó. Một vi xử lý thực thi tất cả các chương trình. Cho dù nó có thể xuất hiện mà các chương trình đã được thực thi đồng thời, trên thực tế bộ vi xử lý nhảy qua lại giữa các tiến trình.

## 3. Tạo và quản lý luồng

Khi các chương trình Java được thực thi, luồng chính luôn luôn đang được thực hiện. Đây là 2 nguyên nhân quan trọng đối với luồng chính:

- Các luồng con sẽ được tạo ra từ nó.
- Nó là luồng cuối cùng kết thúc việc thực hiện. Trong chốc lát luồng chính ngừng thực thi, chương trình bị chấm dứt.

Cho dù luồng chính được tạo ra một cách tự động với chương trình thực thi, nó có thể được điều khiển thông qua một luồng đối tượng.

Các luồng có thể được tạo ra từ hai con đường:

- Trình bày lớp như là một lớp con của lớp luồng, nơi mà phương thức run() của lớp luồng cần được ghi đè. Lấy ví dụ:

Class Mydemo extends Thread

```
{
    //Class definition
    public void run()
    {
        //thực thi
    }
}
```

- Trình bày một lớp mà lớp này thực hiện lớp Runnable. Rồi thì định nghĩa phương thức run().

Class Mydemo implements Runnable

```
{
    //Class definition
    public void run()
    {
        //thực thi
    }
}
```

Chương trình 8.1 sẽ chỉ ra sự điều khiển luồng chính như thế nào

### **Chương trình 8.1**

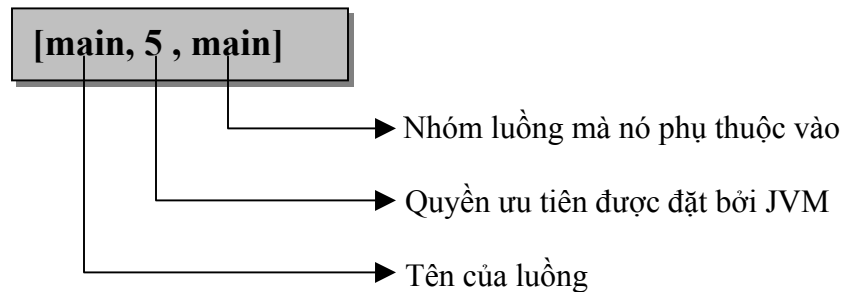
```
import java.io.*;
public class Mythread extends Thread{
/**
 * Mythread constructor comment.
 */
    public static void main(String args[]){
        Thread t = Thread.currentThread();
        System.out.println("The current Thread is :" + t);
        t.setName("MyJavaThread");
        System.out.println("The thread is now named: " + t);
        try{
            for(int i = 0; i <3;i++){
                System.out.println(i);
                Thread.sleep(1500);
            }
        }catch(InterruptedException e){
            System.out.println("Main thread interrupted");
        }
    }
}
```

Hình sau đây sẽ chỉ ra kết quả xuất ra màn hình của chương trình trên

```
Output
The current Thread is :Thread[main,5,main]
The thread is now named: Thread[MyJavaThread,5,main]
0
1
2
```

Hình 8.1 Luồng

Trong kết quả xuất ra ở trên



Mỗi luồng trong chương trình Java được đăng ký cho một quyền ưu tiên. Máy ảo Java không bao giờ thay đổi quyền ưu tiên của luồng. Quyền ưu tiên vẫn còn là hằng số cho đến khi luồng bị ngắt.

Mỗi luồng có một giá trị ưu tiên nằm trong khoảng của một Thread.MIN\_PRIORITY của 1, và một Thread.MAX\_PRIORITY của 10. Mỗi luồng phụ thuộc vào một nhóm luồng, và mỗi nhóm luồng có quyền ưu tiên của chính nó. Mỗi luồng được nhận một hằng số ưu tiên của phương thức Thread.PRIORITY là 5. Mỗi luồng mới thừa kế quyền ưu tiên của luồng mà tạo ra nó.

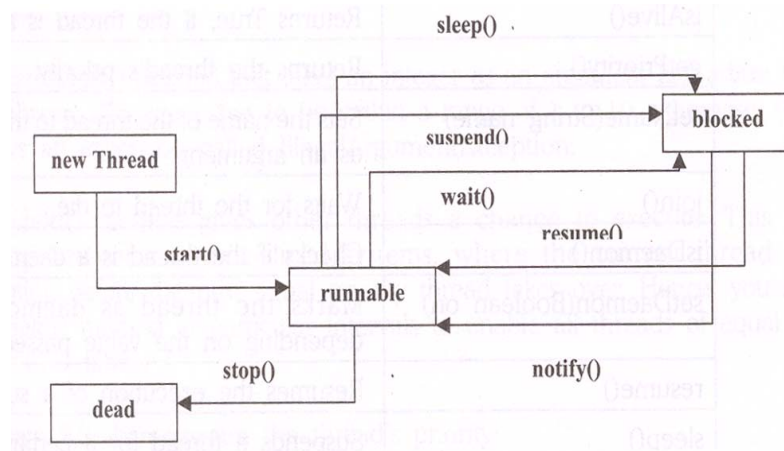
Lớp luồng có vài phương thức khởi dựng, hai trong số các phương thức khởi dựng được đề cập đến dưới đây:

- **public Thread(String threadname)**  
Cấu trúc một luồng với tên là “threadname”

- **public Thread()**  
Cấu trúc một luồng với tên “Thread, được ràng buộc với một số; lấy ví dụ, Thread-1, Thread-2, v.v...

Chương trình bắt đầu thực thi luồng với việc gọi phương thức start(), mà phương thức này phụ thuộc vào lớp luồng. Phương thức này, lần lượt, viện dẫn phương thức run(), nơi mà phương thức định nghĩa tác vụ được thực thi. Phương thức này có thể viết đè lên lớp con của lớp luồng, hoặc với một đối tượng Runnable.

#### 4. Vòng đời của Luồng



**Hình 8.3 Vòng đời của luồng**

### 5. Phạm vi của luồng và các phương thức của lớp luồng

Một luồng đã được tạo mới gần đây là trong phạm vi “sinh”. Luồng không bắt đầu chạy ngay lập tức sau khi nó được tạo ra. Nó đợi phương thức `start()` của chính nó được gọi. Cho đến khi, nó là trong phạm vi “sẵn sàng để chạy”. Luồng đi vào phạm vi “đang chạy” khi hệ thống định rõ vị trí luồng trong bộ vi xử lý.

Bạn có thể sử dụng phương thức `sleep()` để tạm thời treo sự thực thi của luồng. Luồng trở thành sẵn sàng sau khi phương thức `sleep` kết thúc thời gian. Luồng Sleeping không sử dụng bộ vi xử lý. luồng đi vào phạm vi “waiting” (đợi) khi một luồng đang chạy gọi phương thức `wait()` (đợi).

Khi các luồng khác liên kết với các đối tượng, gọi phương thức `notify()`, luồng đi vào trở lại phạm vi “ready” (sẵn sàng) Luồng đi vào phạm vi “blocked” (khỏi) khi nó đang thực thi các phép toán vào/ra (Input/output). Nó đi vào phạm vi “ready” (sẵn sàng) khi các phương thức vào/ra nó đang đợi cho đến khi được hoàn thành. Luồng đi vào phạm vi “dead” (chết) sau khi phương thức `run()` đã được thực thi hoàn toàn, hoặc khi phương thức `stop()` (dừng) của nó được gọi.

Thêm vào các phương thức đã được đề cập trên, Lớp luồng cũng có các phương thức sau:

Phương thức	Mục đích
<code>Enumerate(Thread t)</code>	Sao chép tất cả các luồng hiện hành vào mảng được chỉ định từ nhóm của các luồng, và các nhóm con của nó.
<code>getName()</code>	Trả về tên của luồng
<code>isAlive()</code>	Trả về Đúng, nếu luồng là vẫn còn tồn tại (sống)
<code>getPriority()</code>	Trả về quyền ưu tiên của luồng
<code>setName(String name)</code>	Đặt tên của luồng là tên mà luồng được truyền như là một tham số.
<code>join()</code>	Đợi cho đến khi luồng chết.
<code>isDaemon(Boolean on)</code>	Kiểm tra nếu luồng là luồng một luồng hiểm.
<code>resume()</code>	Đánh dấu luồng như là luồng hiểm hoặc luồng người sử dụng phụ thuộc vào giá trị được truyền vào.
<code>sleep()</code>	Hoãn luồng một khoảng thời gian chính xác.
<code>start()</code>	Gọi phương thức <code>run()</code> để bắt đầu một luồng.

### ***Bảng 8.1 Các phương thức của một lớp luồng***

Bảng kế hoạch Round-robin (bảng kiến nghị ký tên vòng tròn) liên quan đến các luồng với cùng quyền ưu tiên được chiếm hữu quyền ưu tiên của mỗi luồng khác. Chúng chia nhỏ thời gian một cách tự động trong theo kiểu kế hoạch xoay vòng này.

Phiên bản mới nhất của Java không hỗ trợ các phương thức Thread.suspend() (trì hoãn), Thread.resume() (phục hồi) và Thread.stop() (dừng), như là các phương thức resume() (phục hồi) và suspend() (trì hoãn) được thiên về sự đình trệ (deadlock), trong khi phương thức stop() không an toàn.

## **6. Thời gian biểu luồng**

Hầu hết các chương trình Java làm việc với nhiều luồng. CPU chứa đựng cho việc chạy chương trình chỉ một luồng tại một khoảng thời gian. Hai luồng có cùng quyền ưu tiên trong một chương trình hoàn thành trong một thời gian CPU. Lập trình viên, hoặc máy ảo Java, hoặc hệ điều hành chắc chắn rằng CPU được chia sẻ giữa các luồng. Điều này được biết như là bảng thời gian biểu luồng.

Không có máy ảo Java nào thực thi rành mạch cho bảng thời gian biểu luồng. Một số nền Java hỗ trợ việc chia nhỏ thời gian. Ở đây, mỗi luồng nhận một phần nhỏ của thời gian bộ vi xử lý, được gọi là định lượng. Luồng có thể thực thi tác vụ của chính nó trong suốt khoảng thời gian định lượng đấy. Sau khoảng thời gian này được vượt qua, luồng không được nhận nhiều thời gian để tiếp tục thực hiện, ngay cả nếu nó không được hoàn thành việc thực hiện của nó. Luồng kế tiếp của luồng có quyền ưu tiên bằng nhau này sẽ lấy khoảng thời gian thay đổi của bộ vi xử lý. Java là người lập thời gian biểu chia nhỏ tất cả các luồng có cùng quyền ưu tiên cao.

Phương thức setPriority() lấy một số nguyên (integer) như là một tham số có thể hiệu chỉnh quyền ưu tiên của một luồng. Đây là giá trị có phạm vi thay đổi từ 1 đến 10, mặc khác, phương thức đưa ra một ngoại lệ (bẫy lỗi) được gọi là IllegalArgumentException (Chấp nhận tham số trái luật)

Phương thức yield() (lợi nhuận) đưa ra các luồng khác một khả năng để thực thi. Phương thức này được thích hợp cho các hệ thống không chia nhỏ thời gian (non-time-sliced), nơi mà các luồng hiện thời hoàn thành việc thực hiện trước khi các luồng có quyền ưu tiên ngang nhau kế tiếp tiếp quản. Ở đây, bạn sẽ gọi phương thức yield() tại những khoản thời gian riêng biệt để có thể tất cả các luồng có quyền ưu tiên ngang nhau chia sẻ thời gian thực thi CPU.

Chương trình 8.2 chứng minh quyền ưu tiên của luồng:

### **Chương trình 8.2**

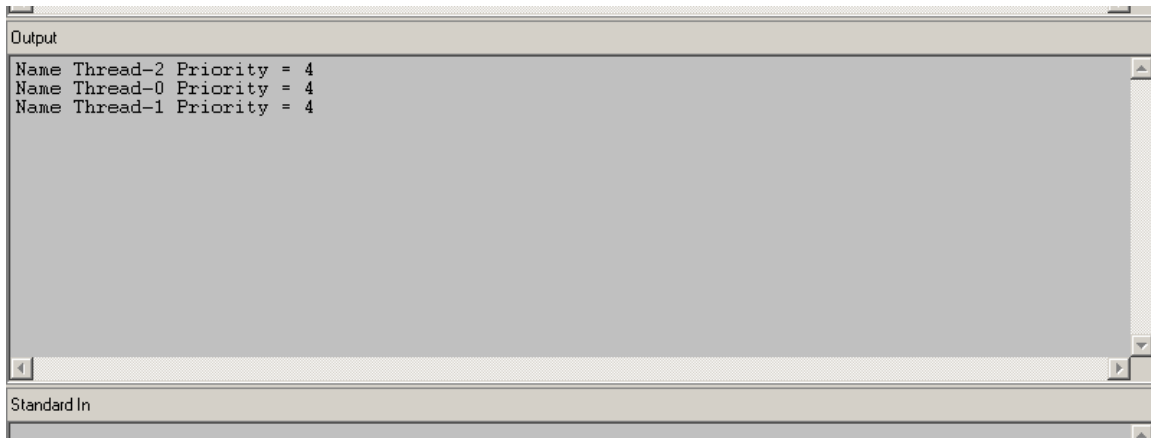
```
class PriorityDemo {
    Priority t1,t2,t3;
    public PriorityDemo(){
        t1 = new Priority();
        t1.start();
        t2 = new Priority();
        t2.start();
        t3 = new Priority();
        t3.start();
    }
    public static void main(String args[]){
```

```

        new PriorityDemo();
    }
    class Priority extends Thread implements Runnable{
        int sleep;
        int prio = 3;
        public Priority(){
            sleep += 100;
            prio++;
            setPriority(prio);
        }
        public void run(){
            try{
                Thread.sleep(sleep);
                System.out.println("Name "+ getName()+" Priority
= "+ getPriority());
            } catch (InterruptedException e){
                System.out.println(e.getMessage());
            }
        }
    }
}

```

Kết quả hiển thị như hình 8.4



**Hình 8.4 Quyền ưu tiên luồng**

## 7. Luồng hiểm

Một chương trình Java bị ngắt chỉ sau khi tất cả các luồng bị chết. Có hai kiểu luồng trong một chương trình Java:

- Các luồng người sử dụng
- Luồng hiểm

Người sử dụng tạo ra các luồng người sử dụng, trong khi các luồng được chỉ định như là luồng “background” (nền). Luồng hiểm cung cấp các dịch vụ cho các luồng khác. Máy ảo Java thực hiện tiến trình thoát, khi và chỉ khi luồng hiểm vẫn còn sống. Máy ảo

Java có ít nhất một luồng hiểm được biết đến như là luồng “garbage collection” (thu lượm những dữ liệu vô nghĩa - dọn rác). Luồng dọn rác thực thi chỉ khi hệ thống không có tác vụ nào. Nó là một luồng có quyền ưu tiên thấp. Lớp luồng có hai phương thức để thỏa thuận với các luồng hiểm:

- public void setDaemon(boolean on)
- public boolean isDaemon()

## 8. Đa tuyến với Applets

Trong khi đa tuyến là rất hữu dụng trong các chương trình ứng dụng độc lập, nó cũng đáng được quan tâm với các ứng dụng trên Web. Đa tuyến được sử dụng trên web, cho ví dụ, trong các trò chơi đa phương tiện, các bức ảnh đầy sinh khí, hiển thị các dòng chữ chạy qua lại trên biểu ngữ, hiển thị đồng hồ thời gian như là một phần của trang Web v.vv... Các chức năng này cấu thành các trang web làm quyến rũ và bắt mắt.

Chương trình Java dựa trên Applet thường sử dụng nhiều hơn một luồng. Trong đa tuyến với Applet, lớp java.applet.Applet là lớp con được tạo ra bởi người sử dụng định nghĩa applet. Từ đó, Java không hỗ trợ nhiều kế thừa với các lớp, nó không thể thực hiện được trực tiếp lớp con của lớp luồng trong các applet. Tuy nhiên, chúng ta sử dụng một đối tượng của luồng người sử dụng đã định nghĩa, mà các luồng này, lần lượt, dẫn xuất từ lớp luồng. Một luồng đơn giản xuất hiện sẽ được thực thi tại giao diện (Interface) Runnable

Chương trình 8.3 chỉ ra điều này thực thi như thế nào:

### Chương trình 8.3

```
import java.awt.*;
import java.applet.*;
public class MyApplet extends Applet implements Runnable {
    int i;
    Thread t;
    /**
     * MyApplet constructor comment.
     */
    public void init(){
        t = new Thread(this);
        t.start();
    }
    public void paint(Graphics g){
        g.drawString(" i = "+i,30,30);
    }
    public void run(){
        for(i = 1;i<=20;i++){
            try{
                repaint();
                Thread.sleep(500);
            } catch (InterruptedException e){
                System.out.println(e.getMessage());
            }
        }
    }
}
```

```

    }
}
}

```

Trong chương trình này, chúng ta tạo ra một Applet được gọi là Myapplet, và thực thi giao diện Runnable để cung cấp khả năng đa tuyến cho applet. Sau đó, chúng ta tạo ra một thể nghiệm (instance) cho lớp luồng, với thể nghiệm applet hiện thời như là một tham số để thiết lập (khởi dựng). Rồi thì chúng ta viện dẫn phương thức start() của luồng thể nghiệm này. Lần lượt, rồi sẽ viện dẫn phương thức run(), mà phương thức này thực sự là điểm bắt đầu cho phương thức này. Chúng ta in số từ 1 đến 20 với thời gian kéo trễ là 500 miligiây giữa mỗi số. Phương thức sleep() được gọi để hoàn thành thời gian kéo trễ này. Đây là một phương thức tĩnh được định nghĩa trong lớp luồng. Nó cho phép luồng nằm yên (ngủ) trong khoản thời gian hạn chế.

Xuất ra ngoài có dạng như sau:



**Hình 8.5 Đa tuyến với Applet**

## 9. Nhóm luồng

Một lớp nhóm luồng (ThreadGroup) nắm bắt một nhóm của các luồng. Lấy ví dụ, một nhóm luồng trong một trình duyệt có thể quản lý tất cả các luồng phụ thuộc vào một đơn thể applet. Tất cả các luồng trong máy ảo Java phụ thuộc vào các nhóm luồng mặc định. Mỗi nhóm luồng có một nhóm nguồn cha. Vì thế, các nhóm từ một cấu trúc dạng cây. Nhóm luồng “hệ thống” là gốc của tất cả các nhóm luồng. Một nhóm luồng có thể là thành phần của cả các luồng, và các nhóm luồng.

Hai kiểu nhóm luồng thiết lập (khởi dựng) là:

➤ **public ThreadGroup(String str)**

Ở đây, “str” là tên của nhóm luồng mới nhất được tạo ra.

➤ **public ThreadGroup(ThreadGroup tgroup, String str)**

Ở đây, “tgroup” chỉ ra luồng đang chạy hiện thời như là luồng cha, “str” là tên của nhóm luồng đang được tạo ra.

Một số các phương thức trong nhóm luồng (ThreadGroup) được cho như sau:

➤ **public synchronized int activeCount()**



Trả về số lượng các luồng kích hoạt hiện hành trong nhóm luồng

➤ **public synchronized int activeGroupCount()**

Trả về số lượng các nhóm hoạt động trong nhóm luồng

➤ **public final String getName()**

Trả về tên của nhóm luồng

➤ **public final ThreadGroup getParent()**

Trả về cha của nhóm luồng

## 10. Sự đồng bộ luồng

Trong khi đang làm việc với nhiều luồng, nhiều hơn một luồng có thể muốn thâm nhập cùng biến tại cùng thời điểm. Lấy ví dụ, một luồng có thể cố gắng đọc dữ liệu, trong khi luồng khác cố gắng thay đổi dữ liệu. Trong trường hợp này, dữ liệu có thể bị sai lệch.

Trong những trường hợp này, bạn cần cho phép một luồng hoàn thành trọn vẹn tác vụ của nó (thay đổi giá trị), và rồi thì cho phép các luồng kế tiếp thực thi. Khi hai hoặc nhiều hơn các luồng cần thâm nhập đến một tài nguyên được chia sẻ, bạn cần chắc chắn rằng tài nguyên đó sẽ được sử dụng chỉ bởi một luồng tại một thời điểm. Tiến trình này được gọi là “sự đồng bộ” (synchronization) được sử dụng để lưu trữ cho vấn đề này, Java cung cấp duy nhất, ngôn ngữ cấp cao hỗ trợ cho sự đồng bộ này. Phương thức “đồng bộ” (synchronized) báo cho hệ thống đặt một khóa vòng một tài nguyên riêng biệt.

Mấu chốt của sự đồng bộ hóa là khái niệm “monitor” (sự quan sát, giám sát), cũng được biết như là một bảng mã “semaphore” (bảng mã). Một “monitor” là một đối tượng mà được sử dụng như là một khóa qua lại duy nhất, hoặc “mutex”. Chỉ một luồng có thể có riêng nó một sự quan sát (monitor) tại mỗi thời điểm được đưa ra. Tất cả các luồng khác cố gắng thâm nhập vào monitor bị khóa sẽ bị trì hoãn, cho đến khi luồng đầu tiên thoát khỏi monitor. Các luồng khác được báo chờ đợi monitor. Một luồng mà monitor của riêng nó có thể thâm nhập trở lại cùng monitor.

### 1. Mã đồng bộ

Tất cả các đối tượng trong Java được liên kết với các monitor (sự giám sát) của riêng nó. Để đăng nhập vào monitor của một đối tượng, lập trình viên sử dụng từ khóa synchronized (đồng bộ) để gọi một phương thức hiệu chỉnh (modified). Khi một luồng đang được thực thi trong phạm vi một phương thức đồng bộ (synchronized), bất kỳ luồng khác hoặc phương thức đồng bộ khác mà cố gắng gọi nó trong cùng thể nghiệm sẽ phải đợi.

Chương trình 8.4 chứng minh sự làm việc của từ khóa synchronized (sự đồng bộ). Ở đây, lớp “Target” (mục tiêu) có một phương thức “display()” (hiển thị) mà phương thức này lấy một tham số kiểu số nguyên (int). Số này được hiển thị trong phạm vi các cặp ký tự “< > # số # < >”. Phương thức “Thread.sleep(1000) tạm dừng luồng hiện tại sau khi phương thức “display()” được gọi.

Thiết lập (khởi dựng) của lớp “Source” lấy một tham chiếu đến một đối tượng “t” của lớp “Target”, và một biến số nguyên (integer). Ở đây, một luồng mới cũng được tạo ra. Luồng này gọi phương thức run() của đối tượng “t”. Lớp chính “Synch” thể nghiệm lớp “Target” như là “target (mục tiêu), và tạo ra 3 đối tượng của lớp “Source” (nguồn). Cùng đối tượng “target” được truyền cho mỗi đối tượng “Source”. Phương thức “join()” (gia nhập) làm luồng được gọi đợi cho đến khi việc gọi luồng bị ngắt.

## Chương trình 8.4

```
class Target {

/**
 * Target constructor comment.
 */
    synchronized void display(int num) {
        System.out.print("<> "+num);
        try{
            Thread.sleep(1000);
        } catch (InterruptedException e){
            System.out.println("Interrupted");
        }
        System.out.println(" <>");
    }
}

class Source implements Runnable{
    int number;
    Target target;
    Thread t;

/**
 * Source constructor comment.
 */
    public Source(Target targ,int n){
        target = targ;
        number = n;
        t = new Thread(this);
        t.start();
    }
    public void run(){
        synchronized(target) {
            target.display(number);
        }
    }
}

class Sync {

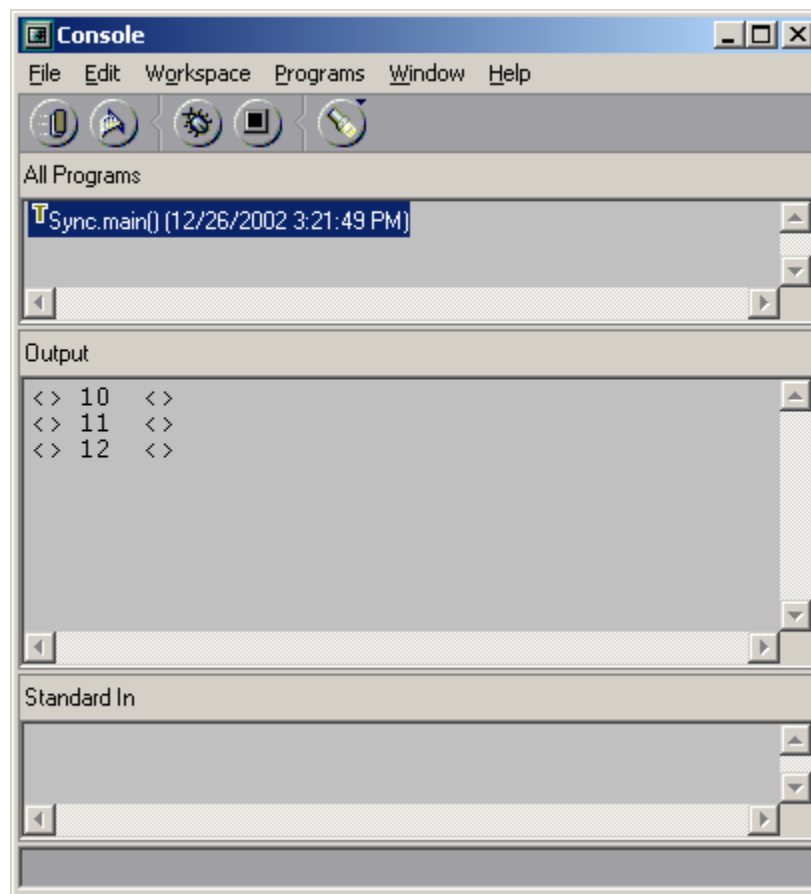
/**
 * Sync constructor comment.
 */
    public static void main(String args[]){
        Target target = new Target();
        int digit = 10;
        Source s1 = new Source(target,digit++);
    }
}
```

```

Source s2 = new Source(target,digit++);
Source s3 = new Source(target,digit++);
try{
    s1.t.join();
    s2.t.join();
    s3.t.join();
} catch (InterruptedException e){
    System.out.println("Interrupted");
}
}
}

```

Kết quả hiện thị như hình cho dưới đây:



**Hình 8.6 Kết quả hiện thị của chương trình 8.4**

Trong chương trình trên, có một “dãy số” đang nhập được hiển thị “display()”. Điều này có nghĩa là việc thêm nhập bị hạn chế một luồng tại mỗi thời điểm. Nếu từ khóa synchronized đặt trước bị bỏ quên trong phương thức “display()” của lớp “Target”, tất cả luồng trên có thể cùng lúc gọi cùng phương thức, trên cùng đối tượng. Điều kiện này được biết đến như là “loại điều kiện” (race condition). Trong trường hợp này, việc xuất ra ngoài sẽ được chỉ ra như hình 8.7



**Hình 8.7** Kết quả hiển thị của chương trình 8.7 không có sự đồng bộ

## 2. Sử dụng khối đồng bộ (Synchronized Block)

Tạo ra các phương thức synchronized (đồng bộ) trong phạm vi các lớp là một con đường dễ dàng và có hiệu quả của việc thực hiện sự đồng bộ. Tuy nhiên, điều này không làm việc trong tất cả các trường hợp.

Hãy xem một trường hợp nơi mà lập trình viên muốn sự đồng bộ được xâm nhập vào các đối tượng của lớp mà không được thiết kế cho thâm nhập đa tuyến. Tức là, lớp không sử dụng các phương thức đồng bộ. Hơn nữa, mã nguồn là không có giá trị. Vì thế từ khoá synchronized không thể được thêm vào các phương thức thích hợp trong phạm vi lớp.

Để đồng bộ thâm nhập một đối tượng của lớp này, tất cả chúng gọi các phương thức mà lớp này định nghĩa, được đặt bên trong một khối đồng bộ. Tất cả chúng sử dụng chung một câu lệnh đồng bộ được cho như sau:

```
synchronized(object)
{
    // các câu lệnh đồng bộ
}
```

Ở đây, “object” (đối tượng) là một tham chiếu đến đối tượng được đồng bộ. Dấu ngoặc móc không cần thiết khi chỉ một câu lệnh được đồng bộ. Một khối đồng bộ bảo đảm rằng nó gọi đến một phương thức (mà là thành phần của đối tượng) xuất hiện chỉ sau khi luồng hiện hành đã được tham nhập thành công vào monitor (sự quan sát) của đối tượng.

Chương trình 8.5 chỉ ra câu lệnh đồng bộ sử dụng như thế nào:

### **Chương trình 8.5**

```
class Target {

    /**
     * Target constructor comment.
     */

    synchronized void display(int num) {
        System.out.print("<> "+num);
        try{
            Thread.sleep(1000);
        } catch (InterruptedException e){
            System.out.println("Interrupted");
        }
    }
}
```

```

        System.out.println(" <>");
    }
}

class Source implements Runnable{
    int number;
    Target target;
    Thread t;
/**
 * Source constructor comment.
 */
    public Source(Target targ,int n){
        target = targ;
        number = n;
        t = new Thread(this);
        t.start();
    }
    // đồng bộ gọi phương thức display()
    public void run(){
        synchronized(target) {
            target.display(number);
        }
    }
}

class Synchblock {
/**
 * Synchblock constructor comment.
 */
    public static void main(String args[]){
        Target target = new Target();
        int digit = 10;
        Source s1 = new Source(target,digit++);
        Source s2 = new Source(target,digit++);
        Source s3 = new Source(target,digit++);
        try{
            s1.t.join();
            s2.t.join();
            s3.t.join();
        } catch (InterruptedException e){
            System.out.println("Interrupted");
        }
    }
}

```

Ở đây, từ khóa `synchronized` không hiệu chỉnh phương thức `display()`. Từ khóa này được sử dụng trong phương thức `run()` của lớp `“Target”` (mục tiêu). Kết quả xuất ra màn hình tương tự với kết quả chỉ ra ở hình số 8.6

### 3. Sự không thuận lợi của các phương thức đồng bộ

Người lập trình thường viết các chương trình trên các đơn thể luồng. Tất nhiên các trạng thái này chắc chắn không lợi ích cho đa tuyến. Lấy ví dụ, luồng không tận dụng việc thực thi của trình biên dịch. Trình biên dịch Java từ Sun không chứa nhiều phương thức đồng bộ.

Các phương thức đồng bộ không thực thi tốt như là các phương thức không đồng bộ. Các phương thức này chậm hơn từ ba đến bốn lần so với các phương thức tương ứng không đồng bộ. Trong trạng thái nơi mà việc thực thi là có giới hạn, các phương thức đồng bộ bị ngăn ngừa.

## 11. Kỹ thuật “wait-notify” (đợi – thông báo)

Luồng chia các tác vụ thành các đơn vị riêng biệt và logic (hợp lý). Điều này thay thế các trường hợp (sự kiện) chương trình lập. Các luồng loại trừ “polling” (kiểm soát vòng).

Một vòng lặp mà lặp lại việc một số điều kiện thường thực thi “polling” (kiểm soát vòng). Khi điều kiện nhận giá trị là `True` (đúng), các câu lệnh phức tạp được thực hiện. Đây là tiến trình thường bỏ phí thời gian của CPU. Lấy ví dụ, khi một luồng sinh ra một số dữ liệu, và các luồng khác đang chờ đợi nó, luồng sinh ra phải đợi cho đến khi các luồng sử dụng nó hoàn thành, trước khi phát sinh ra dữ liệu.

Để tránh trường hợp kiểm soát vòng, Java bao gồm một thiết kế tốt trong tiến trình kỹ thuật truyền thông sử dụng các phương thức `“wait()”` (đợi), `“notify()”` (thông báo) và `“notifyAll()”` (thông báo hết). Các phương thức này được thực hiện như là các phương thức cuối cùng trong lớp đối tượng (`Object`), để mà tất cả các lớp có thể thâm nhập chúng. Tất cả 3 phương thức này có thể được gọi chỉ từ trong phạm vi một phương thức đồng bộ (`synchronized`).

Các chức năng của các phương thức `“wait()”`, `“notify()”`, và `“notifyAll()”` là:

- Phương thức **`wait()`** nói cho việc gọi luồng trao cho monitor (sự giám sát), và nhập trạng thái “sleep” (chờ) cho đến khi một số luồng khác thâm nhập cùng monitor, và gọi phương thức `“notify()”`.
- Phương thức **`notify()`** đánh thức, hoặc thông báo cho luồng đầu tiên mà đã gọi phương thức `wait()` trên cùng đối tượng.
- Phương thức **`notifyAll()`** đánh thức, hoặc thông báo tất cả các luồng mà đã gọi phương thức `wait()` trên cùng đối tượng.
- Quyền ưu tiên cao nhất luồng chạy đầu tiên.

Cú pháp của 3 phương thức này như sau:

**`final void wait() throws IOException`**

**`final void notify()`**

**`final void notifyAll()`**

Các phương thức `wait()` và `notify()` cho phép một đối tượng được chia sẻ để tạm ngừng một luồng, khi đối tượng trở thành không còn giá trị cho luồng. Chúng cũng cho phép luồng tiếp tục khi thích hợp.

Các luồng bản thân nó không bao giờ kiểm tra trạng thái của đối tượng đã chia sẻ.

Một đối tượng mà điều khiển các luồng khách (client) của chính nó theo kiểu này được biết như là một monitor (sự giám sát). Trong các thuật ngữ chặt chẽ của Java, một monitor là bất kỳ đối tượng nào mà có một số mã đồng bộ. Các monitor được sử dụng cho các phương thức `wait()` và `notify()`. Cả hai phương thức này phải được gọi trong mã đồng bộ.

Một số điểm cần nhớ trong khi sử dụng phương thức **`wait()`**:

- Luồng đang gọi đưa vào CPU
- Luồng đang gọi đưa vào khóa
- Luồng đang gọi đi vào vùng đợi của monitor.

Các điểm chính cần nhớ về phương thức **`notify()`**

- Một luồng đưa ra ngoài vùng đợi của monitor, và vào trạng thái sẵn sàng.
- Luồng mà đã được thông báo phải thu trở lại khóa của monitor trước khi nó có thể bắt đầu.
- Phương thức `notify()` là không chính xác, như là nó không thể chỉ ra được luồng mà phải được thông báo. Trong một trạng thái đã trộn lẫn, luồng có thể thay đổi trạng thái của monitor trong một con đường mà không mang lại kết quả tốt cho luồng đã được đưa thông báo. Trong một số trường hợp này, các phương thức của monitor đưa ra 2 sự đề phòng:
  - Trạng thái của monitor sẽ được kiểm tra trong một vòng lặp “while” tốt hơn là câu lệnh if
  - Sau khi thay đổi trạng thái của monitor, phương thức `notifyAll()` sẽ được sử dụng, tốt hơn phương thức `notify()`.

Chương trình 8.6 biểu thị cho việc sử dụng các phương thức `notify()` và `wait()`:

### Chương trình 8.6

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*<applet code = "mouseApplet" width = "100" height = "100"> </applet> */
public class mouseApplet extends Applet implements MouseListener{
    boolean click;
    int count;
    public void init() {
        super.init();
        add(new clickArea(this)); //doi tuong ve duoc tao ra va them vao
        add(new clickArea(this)); //doi tuong ve duoc tao ra va them vao
        addMouseListener(this);
    }
    public void mouseClicked(MouseEvent e) {

    }
    public void mouseEntered(MouseEvent e) {
```

```

    }
    public void mouseExited(MouseEvent e) {

    }
    public void mousePressed(MouseEvent e) {
        synchronized (this) {
            click = true;
            notify();
        }
        count++; //dem viec click
        Thread.currentThread().yield();
        click = false;

    }
    public void mouseReleased(MouseEvent e) {

    }
} //kết thúc Applet

class clickArea extends java.awt.Canvas implements Runnable{
    mouseApplet myapp;
    clickArea(mouseApplet mapp){
        this.myapp = mapp;
        setSize(40,40);
        new Thread(this).start();
    }
    public void paint(Graphics g){
        g.drawString(new Integer(myapp.count).toString(),15,20);

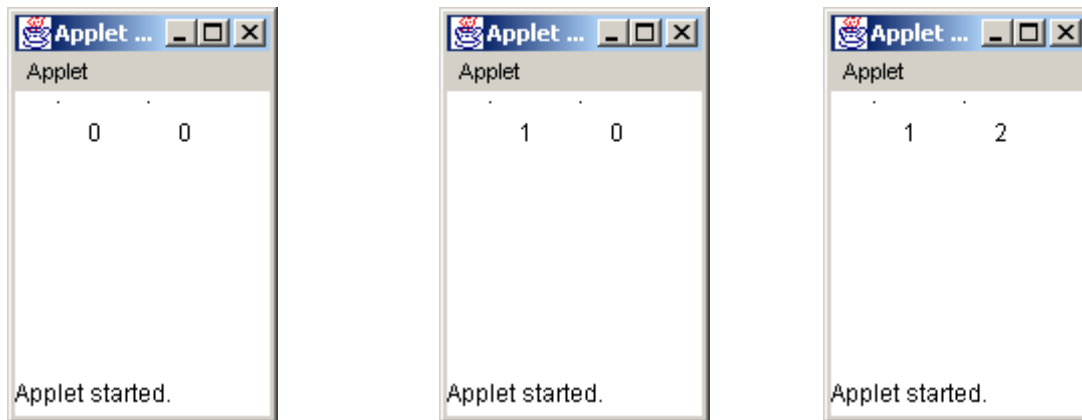
    }
    public void run(){
        while(true){
            synchronized (myapp) {
                while(!myapp.click){
                    try{
                        myapp.wait();
                    } catch (InterruptedException ie){
                    }
                }
            }
            repaint(250);
        }
    }
}

} //end run
}

```



Không cần các phương thức wait() và notify(), luồng bức vẽ (canvas) không thể biết khi nào cập nhật hiển thị. Kết quả xuất ra ngoài của chương trình được đưa ra như sau:



**Hình 8.8 Kết quả sau mỗi lần kích chuột**

## 12. Sự bế tắc (Deadlocks)

Một “deadlock” (sự bế tắc) xảy ra khi hai luồng có một phụ thuộc vòng quanh trên một cặp đối tượng đồng bộ; lấy ví dụ, khi một luồng thâm nhập vào monitor trên đối tượng “ObjA”, và một luồng khác thâm nhập vào monitor trên đối tượng “ObjB”. Nếu luồng trong “ObjA” cố gắng gọi phương thức đồng bộ trên “ObjB”, một bế tắc xảy ra.

Nó khó để gỡ lỗi một bế tắc bởi những nguyên nhân sau:

- Nó hiếm khi xảy ra, khi hai luồng chia nhỏ thời gian trong cùng một con đường
- Nó có thể bao hàm nhiều hơn hai luồng và hai đối tượng đồng bộ

Nếu một chương trình đa tuyến khóa kín thường xuyên, ngay lập tức kiểm tra lại điều kiện bế tắc.

Chương trình 8.7 tạo ra điều kiện bế tắc. Lớp chính (main) bắt đầu 2 luồng. Mỗi luồng gọi phương thức đồng bộ run(). Khi luồng “t1” đánh thức, nó gọi phương thức “synchIt()” của đối tượng deadlock “dlk1”. Từ đó luồng “t2” một mình giám sát cho “dlk2”, luồng “t1” bắt đầu đợi monitor. Khi luồng “t2” đánh thức, nó cố gắng gọi phương thức “synchIt()” của đối tượng Deadlock “dlk2”. Bây giờ, “t2” cũng phải đợi, bởi vì đây là trường hợp tương tự với luồng “t1”. Từ đó, cả hai luồng đang đợi lẫn nhau, cả hai sẽ đánh thức. Đây là điều kiện bế tắc.

### Chương trình 8.7

```
public class Deadlock implements Runnable{
    public static void main(String args[]){
        Deadlock dlk1= new Deadlock();
        Deadlock dlk2 = new Deadlock();
        Thread t1 = new Thread(dlk1);
        Thread t2 = new Thread(dlk2);

        dlk1.grabIt = dlk1;
        dlk2.grabIt = dlk2;
```

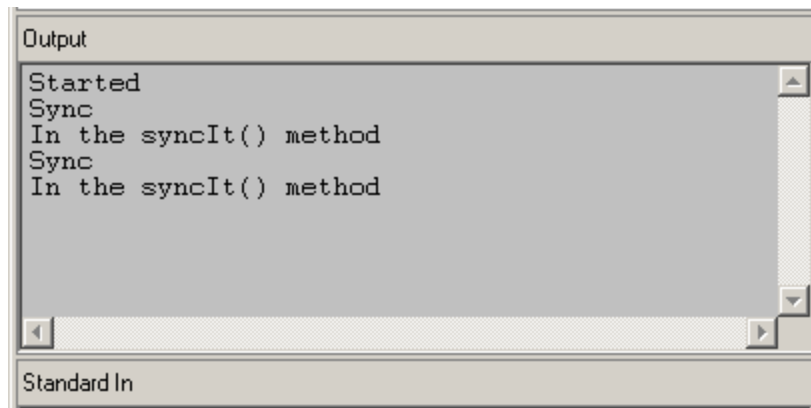
```

        t1.start();
        t2.start();
        System.out.println("Started");
        try{
            t1.join();
            t2.join();
        }catch(InterruptedException e){
            System.out.println("error occurred");
        }
        System.exit(0);
    }
    Deadlock grabIt;
    public synchronized void run() {
        try{
            Thread.sleep(1500);
        }catch(InterruptedException e){
            System.out.println("error occurred");
        }
        grabIt.syncIt();
    }
    public synchronized void syncIt() {
        try{
            Thread.sleep(1500);
            System.out.println("Sync");

        }catch(InterruptedException e){
            System.out.println("error occurred");
        }
        System.out.println("In the syncIt() method");
    }
}

```

Kết quả của chương trình này được hiển thị như sau:



**Hình 8.9 Sự bế tắc**

### 13. Thu dọn “rác” (Garbage collection)

Thu dọn “rác” (Garbage collection) cải tạo hoặc làm trống bộ nhớ đã định vị cho các đối tượng mà các đối tượng này không sử dụng trong thời gian dài. Trong ngôn ngữ lập trình hướng đối tượng khác như C++, lập trình viên phải làm cho bộ nhớ trống mà đã không được yêu cầu trong thời gian dài. Tình trạng không hoạt động để bộ nhớ trống có thể là kết quả trong một số vấn đề. Java tự động tiến hành thu dọn rác để cung cấp giải pháp duy nhất cho vấn đề này. Một đối tượng trở nên thích hợp cho sự dọn rác nếu không có tham chiếu đến nó, hoặc nếu nó đã đăng ký rỗng.

Sự dọn rác thực thi như là một luồng riêng biệt có quyền ưu tiên thấp. Bạn có thể viện dẫn một phương thức gc() của thể nghiệm để viện dẫn sự dọn rác. Tuy nhiên, bạn không thể dự đoán hoặc bảo đảm rằng sự dọn rác sẽ thực thi một cách trọn vẹn sau đó.

Sử dụng câu lệnh sau để tắt đi sự dọn rác trong ứng dụng:

**Java –noasyncgc ....**

Nếu chúng ta tắt đi sự dọn rác, chương trình hầu như chắc chắn rằng bị treo do bởi việc đó.

#### 1. Phương thức finalize() (hoàn thành)

Java cung cấp một con đường để làm sạch một tiến trình trước khi điều khiển trở lại hệ điều hành. Điều này tương tự như phương thức phân hủy của C++

Phương thức finalize(), nếu hiện diện, sẽ được thực thi trên mỗi đối tượng, trước khi sự dọn rác.

Câu lệnh của phương thức finalize() như sau:

**protected void finalize() throws Throwable**

Tham chiếu không phải là sự dọn rác; chỉ các đối tượng mới được dọn rác

Lấy thể nghiệm:

**Object a = new Object();**

**Object b = a;**

**a = null;**

Ở đây, nó sẽ sai khi nói rằng “b” là một đối tượng. Nó chỉ là một đối tượng tham chiếu. Hơn nữa, trong đoạn mã trích trên mặc dù “a” được đặt là rỗng, nó không thể được dọn rác, bởi vì nó vẫn còn có một tham chiếu (b) đến nó. Vì thế “a” vẫn còn với đến được, thật vậy, nó vẫn còn có phạm vi sử dụng trong phạm vi chương trình. Ở đây, nó sẽ không được dọn rác.

Tuy nhiên, trong ví dụ cho dưới đây, giả định rằng không có tham chiếu đến “a” tồn tại, đối tượng “a” trở nên thích hợp cho garbage collection.

**Object a = new Object();**

**...**

**...**

**...**

**a = null;**

Một ví dụ khác:

**Object m = new Object();**

**Object m = null;**

Đối tượng được tạo ra trong sự bắt đầu có hiệu lực cho garbage collection

**Object m = new Object();**

**M = new Object();**

Bây giờ, đối tượng căn nguyên có hiệu lực cho garbage collection, và một đối tượng mới tham chiếu bởi “m” đang tồn tại.

Bạn có thể chạy phương thức garbage collection, nhưng không có bảo đảm rằng nó sẽ xảy ra.

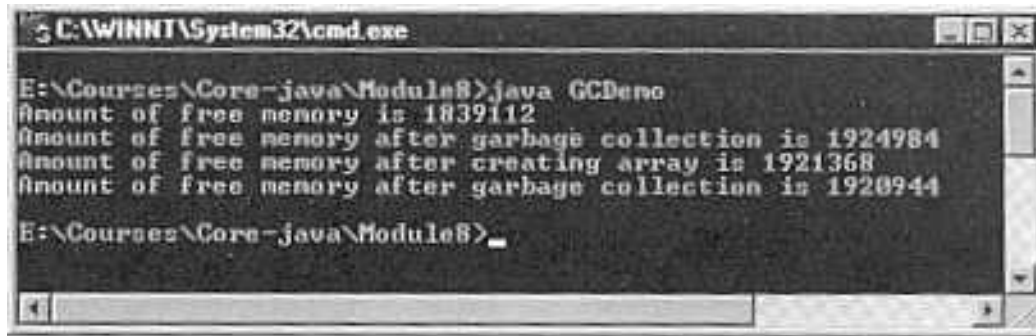
Chương trình 8.8 diễn hình cho garbage collection.

### **Chương trình 8.8**

```
class GCDemo
{
    public static void main(String args[])
    {
        int i;
        long a; ,
        Runtime r=Runtime.getRuntimeO;
        Long valuesD =new Long[200];
        System.out.print In ("Amount of free memory is" +
r.freeMemoryO);
        r.gcO;
        System.out.println("Amount of free memory after garbage
collection is " + r.freeMemoryO);
        for (a=100000.i=0;i<200;a++.i++)
        {
            values[i] =new Long(a);
        }
        System.out.println("Amount of free memory after creating the array
" + r.freeMemoryO);
        for (i=0;i<200;i++)
        {
            values[i] =null;
        }
        System.out.println("Arnount of free memory after garbage collection is
" + r.freeMemoryO);
    }
}
```

Chúng ta khai một mảng gồm 200 phần tử, trong đó kiểu dữ liệu là kiểu Long. Trước khi mảng được tạo ra, chúng ta phải xác định rõ số lượng bộ nhớ trống, và hiển thị nó. Rồi thì chúng ta viện dẫn phương thức gc() của thể nghiệm Runtime (thời gian thực thi) hiện thời. Điều này có thể hoặc không thể thực thi garbage collection. Rồi thì chúng ta tạo ra mảng, và đang ký giá trị cho các phần tử của mảng. Điều này sẽ giảm bớt số lượng bộ nhớ trống. Để làm các mảng phần tử thích hợp cho garbage collection, chúng ta đặt chúng rỗng. Cuối cùng, chúng ta sử dụng phương thức gc() để viện dẫn garbage collection lần nữa.

Kết quả xuất ra màn hình của chương trình trên như sau:



```
C:\WINNT\System32\cmd.exe
E:\Courses\Core-java\Module8>java GCDemo
Amount of free memory is 1839112
Amount of free memory after garbage collection is 1924984
Amount of free memory after creating array is 1921368
Amount of free memory after garbage collection is 1920944
E:\Courses\Core-java\Module8>
```

**Hình 8.10 Garbage collection**

### Tổng kết

- Một luồng là đơn vị nhỏ nhất của đoạn mã thực thi được mà một tác vụ riêng biệt.
- Đa tuyến giữ cho thời gian rồi là nhỏ nhất. Điều này cho phép bạn viết các chương trình có khả năng sử dụng tối đa CPU.
- Luồng bắt đầu thực thi sau khi phương thức start() được gọi
- Lập trình viên, máy ảo Java, hoặc hệ điều hành bảo đảm rằng CPU được chia sẻ giữa các luồng.
- Có hai loại luồng trong một chương trình Java:
  - Luồng người dùng
  - Luồng hiểm.
- Một nhóm luồng là một lớp mà nắm bắt một nhóm các luồng.
- Đồng bộ cho phép chỉ một luồng thâm nhập một tài nguyên được chia sẻ tại một thời điểm.
- Để tránh kiểm soát vòng, Java bao gồm một thiết kế tốt trong tiến trình kỹ thuật truyền thông sử dụng các phương thức “wait()” (đợi), “notify()” (thông báo) và “notifyAll()” (thông báo hết).
- Một “bế tắc” xảy ra khi hai luồng có một phụ thuộc xoay vòng trên một phần của các đối tượng đồng bộ
- Garbage collection là một tiến trình nhờ đó bộ nhớ được định vị để các đối tượng mà không sử dụng trong thời gian dài, có thể cải tạo hoặc làm rảnh bộ nhớ.

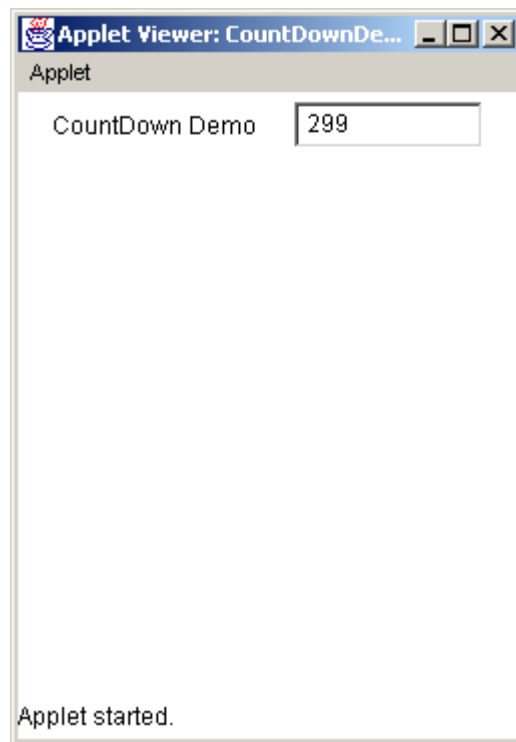
### Kiểm tra lại sự hiểu biết của bạn

- |   |                 |
|---|-----------------|
| 1. Một ứng dụng có thể chứa đựng nhiều luồng  | <b>Đúng/Sai</b> |
| 2. Các luồng con được tạo ra từ luồng chính   | <b>Đúng/Sai</b> |
| 3. Mỗi luồng trong một chương trình Java được đăng ký một quyền ưu tiên mà máy ảo Java có thể thay đổi. | <b>Đúng/Sai</b> |

4. Phương thức \_\_\_\_\_ có thể tạm thời ngừng việc thực thi luồng
5. Mặc định, một luồng có một quyền ưu tiên \_\_\_\_\_ một hằng số của \_\_\_\_\_
6. \_\_\_\_\_ luồng được dùng cho các luồng “nền”, cung cấp dịch vụ cho luồng khác.
7. Trong luồng đồng bộ, một \_\_\_\_\_ là một đối tượng mà được sử dụng như là một khóa riêng biệt lẫn nhau.
8. \_\_\_\_\_ thường thực thi bởi một vòng lặp mà được sử dụng để lặp lại việc kiểm tra một số điều kiện.

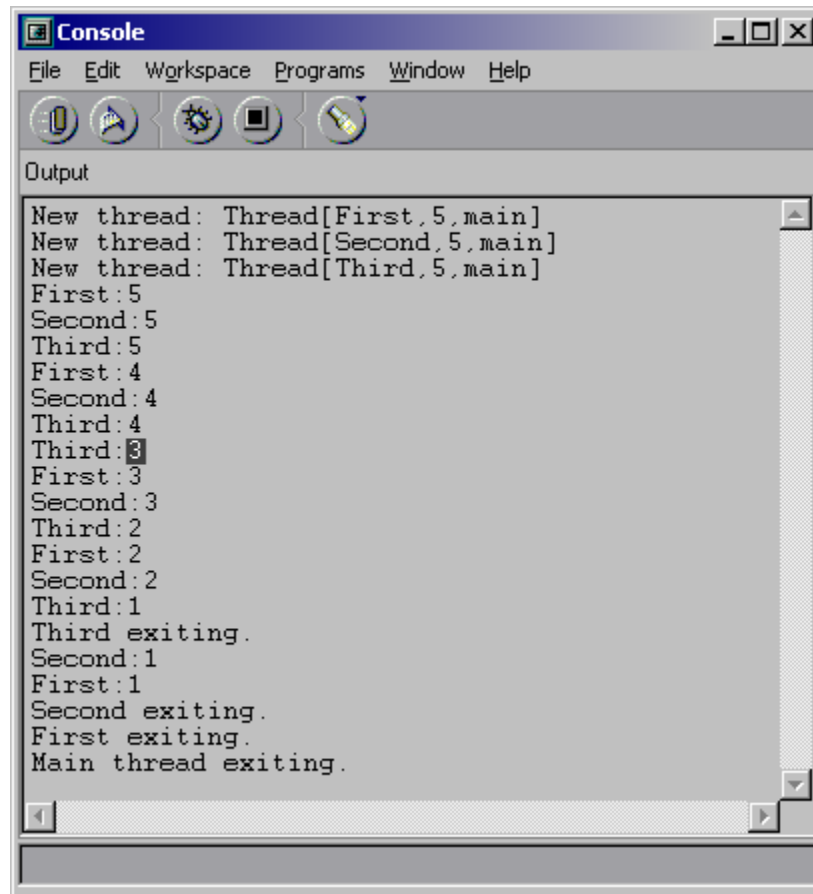
**Bài tập:**

1. Viết một chương trình mà hiển thị một sự đếm lùi từng giây cho đến không, như hình sau:



Ban đầu, số 300 sẽ được hiển thị. Giá trị sẽ được giảm dần cho đến 1 đến khi ngoài giá trị 0. Giá trị sẽ được trả lại 300 một lần nữa giảm đến trở thành 0.

2. Viết một chương trình mà hiển thị như hình dưới đây:



```
Console
File Edit Workspace Programs Window Help
Output
New thread: Thread[First,5,main]
New thread: Thread[Second,5,main]
New thread: Thread[Third,5,main]
First:5
Second:5
Third:5
First:4
Second:4
Third:4
Third:3
First:3
Second:3
Third:2
First:2
Second:2
Third:1
Third exiting.
Second:1
First:1
Second exiting.
First exiting.
Main thread exiting.
```

Tạo 3 luồng và một luồng chính trong “main”. Thực thi mỗi luồng như một chương trình thực thi. Khi chương trình kết thúc, các câu lệnh thoát cho mỗi luồng sẽ được hiển thị. Sử dụng kỹ thuật nắm bắt lỗi.