Chương I - Multithreading (đa tuyến)

Multithreading cho phép hai phần của cùng một chương trình chạy đồng thời. Article này thảo luận về cách làm thế nào để thực hiện điều này tốt nhất trong Java. Đây là một phần trích từ chương 10 của cuốn sách Java Dymistyfied, được viết bởi Jim Keogh.

Các vận động viên marathon thường đối mặt với tình trạng khó khăn khi cả hai cuộc đua chính rơi vào trong cùng một tuần bởi vì họ phải chọn một cuộc đua để chạy. Họ chắc chắn phải mong ước có một cách, một phần của họ có thể chạy một cuộc đua và một phần khác chạy một cuộc đua khác. Điều đó không thể xảy ra – đó là ngoại trừ, vận động viên chính là một chương trình Java, bởi vì hai phần của một chương trình Java có thể chạy đồng thời bằng việc sử dụng multithreading. Bạn sẽ học về multithreading và cách làm thế nào để chạy đồng thời các phần của chương trình của bạn trong chương này.

I- Multitasking (đa nhiệm)

Multitasking thực thi hai hay nhiều tác nhiệm cùng một lúc. Gần như tất cả các hệ điều hành đều có khả năng multitasking bởi việc sử dụng một trong hai kỹ thuật multitasking: multitasking dựa trên process (xử lý) và multitasking dựa trên thread (phân tuyến).

Multitasking dựa trên process chạy hai chương trình cùng một lúc. Các lập trình viên nói đến một chương trình là một process. Vì thế bạn có thể nói rằng, multitasking dựa trên process là multitasking dựa trên chương trình.

Multitasking dựa trên thread có một chương trình thực hiện hai tác nhiệm cùng một thời điểm. Ví dụ, một chương trình xử lý văn bản có thể kiểm tra lỗi chính tả trong một tài liệu trong khi bạn đang viết một tài liệu đó. Đó là multitasking dựa trên thread.

Cách khá tốt để nhớ được sự khác nhau giữa multitasking dựa trên process và multitasking dựa trên thread là hãy nghĩ dựa trên process là làm việc với nhiều chương trình và dựa trên thread là làm việc với nhiều phần của một chương trình.

Mục tiêu của multitasking là sử dụng thời gian nghỉ của CPU. Tưởng tượng rằng CPU là một động cơ trong xe hơi của bạn. Động cơ xe của bạn luôn chạy cho dù xe hơi có di chuyển hay không. Bạn muốn xe hơi di chuyển nhiều đến mức có thể, để bạn có thể chạy được nhiều dặm từ một gallon ga. Một động cơ để không lãng phí ga.

Cùng một khái niệm như thế áp dụng cho CPU của máy tính của bạn. Bạn muốn CPU của bạn xoay vòng để xử lý các lệnh và dữ liệu hơn là chờ một điều gì đó để xử lý. Một CPU xoay vòng là những gì tương tự với sự vận hành của động cơ của bạn.

Có thể khó khăn để tin tưởng, nhưng CPU nghỉ nhiều hơn là nó xử lý trong nhiều máy tính để bàn. Hãy nói rằng, bạn đang sử dụng một trình xử lý văn bản để viết một tài liệu. Trong hầu hết các phần việc, CPU không làm gì cả cho đến khi bạn nhập một ký tự từ bàn phím hoặc di chuyển chuột. Multitasking được thiết kế để sử

dụng phần nhỏ của một giây để xử lý các lệnh từ một chương trình khác hoặc từ một phần khác của cùng một chương trình.
Tạo nên sự sử dụng CPU một cách có hiệu quả không phải là quá quyết định đối với các chương trình chạy trên một máy tính để bàn bởi vì hầu hết chúng ta hiếm khi chạy những chương trình đồng thời hoặc chạy nhiều phần của cùng một chương trình vào một thời điểm. Tuy nhiên, những chương trình đó chạy trên môi trường mạng, cần trao đổi xử lý từ nhiều máy tính, thì cần phải tạo ra một thời gian nghỉ hiệu quả của CPU.

II - Multithreading trong Java – Overhead

Hệ điều hành phải làm những công việc phụ để quản lý multitasking, các lập trình viên gọi công việc phụ này là overhead bởi vì các tài nguyên trong máy tính của bạn được sử dụng để quản lý sự hoạt động của multitasking nhiều hơn là được sử dụng bởi các chương trình để xử lý các lệnh và dữ liệu. Multitasking dựa trên process có một overhead lớn hơn multitasking dựa trên thread. Trong multitasking dựa trên process, mỗi process yêu cầu không gian địa chỉ của chính nó trong vùng nhớ. Hệ điều hành yêu cầu một lượng thời gian CPU xác định để chuyển từ một xử lý này sang một xử lý khác. Các lập trình viên gọi đây là context switching, ở đó mỗi process (hay chương trình) là một context. Các tài nguyên được bổ sung cần cho mỗi process để giao tiếp với mỗi process khác.
Trong sự so sánh này, các thread trong multitasking dựa trên thread chia sẻ cùng một không gian địa chỉ trong vùng nhớ bởi vì chúng chia sẻ cùng một chương trình. Ở đây cũng có một tác động trong context switching, bởi vì chuyển đổi từ một phần của chương trình sang một phần khác xảy ra trong cùng một không gian địa chỉ của vùng nhớ. Ngược lại, việc giao tiếp giữa các phần của chương trình xảy ra trong cùng một vị trí vùng nhớ.

III - Thread

Một thread là một phần của chương trình đang chạy. Multitasking dựa trên thread có nhiều thread chạy cùng một thời điểm (nhiều phần của một chương trình chạy cùng một lúc). Mỗi thread là một cách thực thi khác nhau.

Hãy trở về ví dụ trình xử lý văn bản để xem các thread được sử dụng như thế nào. Hai phần của trình xử lý văn bản được quan tâm: đầu tiên là phần nhận các ký tự nhập từ bàn phím, lưu chúng vào bộ nhớ, và hiển thị chúng trên màn hình. Phần thứ hai là phần còn lại của chương trình nhằm kiểm tra chính tả. Mỗi phần là một thread thực thi độc lập với phần khác. Thậm chí dù chúng là cùng một chương trình. Trong khi một thread nhận và xử lý các ký tự được nhập từ bàn phím, thread khác ngủ. Đó là, thread khác dừng cho đến khi CPU nghỉ. CPU thường nghỉ giữa các lần gõ phím. Vào khoảng thời gian này, thread bộ kiểm tra chính tả đang ngủ thức dậy tiếp tục kiểm tra chính tả của tài liệu. Thread bộ kiểm tra chính tả dừng một lần nữa khi ký tự kế tiếp được nhập từ bàn phím.

Môi trường runtime Java quản lý các thread không giống như multitasking dựa

trên process mà ở đó hệ điều hành quản lý việc chuyển đổi giữa các chương trình. Các thread được xử lý đồng bộ. Điều đó có nghĩa là một thread có thể dừng trong khi một thread có thể tiếp tục xử lý.
Một thread có thể là một trong bốn trạng thái sau:
+Running: một thread đang được thực thi.
+Suspended: việc thực thi bị tạm dừng và có thể phục hồi tại thời điểm dừng
+Blocked: một tài nguyên không thể được truy cập bởi vì nó đang được sử dụng bởi một thread khác.
+Terminated: việc thực thi bị ngừng hẳn và không thể phục hồi.

Tất cả các thread không giống nhau. Một vài thread quan trọng hơn những thread khác và được cho quyền ưu tiên cao hơn đối với các tài nguyên như CPU. Mỗi thread được gán một quyền ưu tiên thread được sử dụng để xác định khi nào thì chuyển từ một thread đang thực thi sang một thread khác. Được gọi là context switching.

Quyền ưu tiên của một thread có quan hệ với quyền ưu tiên của các thread khác. Quyền ưu tiên của một thread là không thích hợp khi chỉ có một mình thread đó đang chạy. Thread có quyền ưu tiên thấp hơn cũng chạy nhanh như thread có quyền ưu tiên cao hơn nếu như không có thread nào khác chạy cùng một lúc.
Các quyền ưu tiên của thread được sử dụng khi các quy luật của context switching được sử dụng. Dưới đây là những quy luật này:
Một thread có thể tự động sản sinh ra một thread khác. Để làm được điều này, điều khiển được trả về cho thread có quyền ưu tiên cao nhất.

Một thread có quyền ưu tiên cao hơn có thể giành quyền sử dụng CPU từ một thread có quyền ưu tiên thấp hơn. Thread có quyền ưu tiên thấp hơn bị tạm dừng bất chấp nó đang làm gì để trả về theo cách của thread có quyền ưu tiên cao hơn. Các lập trình viên gọi đây là preemptive multitasking.

Các thread có quyền ưu tiên bằng nhau được xử lý dựa trên các quy luật của hệ điều hành đang được sử dụng để chạy chương trình. Ví dụ, Windows sử dụng time slicing, bao gồm việc cho mỗi thread có quyền ưu tiên cao một vài mili giây của vòng CPU và giữ việc xoay vòng giữa các thread có quyền ưu tiên cao. Trong Solaris, thread có quyền ưu tiên cao nhất phải tự động sản sinh ra các thread có quyền ưu tiên cao khác. Nếu không, thread có quyền ưu tiên cao thứ nhì phải chờ thread có quyền ưu tiên cao nhất kết thúc.

IV - Sự đồng bộ hoá

Multithreading xảy ra không đồng bộ, có nghĩa là một thread thực thi độc lập với các thread khác. Theo đó, mỗi thread không phụ thuộc vào sự thực thi của các thread khác. Để bắt buộc, các xử lý chạy đồng bộ hóa phụ thuộc vào các xử lý khác. Đó là một xử lý chờ cho đến khi một xử lý khác kết thúc trước khi nó có thể thực thi.

Thỉnh thoảng, việc thực thi của một thread có thể phụ thuộc vào việc thực thi của

một thread khác. Giả sử bạn có hai thread – một tập hợp các thông tin đăng nhập và một cái khác kiểm tra mật khẩu và ID của người dùng. Thread login phải chờ thread validation hoàn tất xử lý trước khi nó có thể nói cho người dùng việc đăng nhập có thành công hay không. Vì thế cả hai thread phải được thực thi đồng bộ, không được không đồng bộ.

Java cho phép các thread đồng bộ hóa được định nghĩa bởi một method đồng bộ hoá. Một thread nằm trong một method đồng bộ hóa ngăn bất kỳ thread nào khác từ một phương thức đồng bộ hoá khác gọi trong cùng một đối tượng. Bạn sẽ học chúng trong phần sau của chương này.

V -Interface (giao tiếp) Runnable và các lớp Thread

Bạn khởi dựng các thread bằng việc sử dụng interface Runnable và class Thread. Điều này có nghĩa là thread của bạn phải kế thừa từ class Thread hoặc bổ sung interface Runnable. Class Thread định nghĩa bên trong các method bạn sử dụng để quản lý thread. Dưới đây là các method thông dụng được định nghĩa trong class Thread.

Method	Mô tả
`getName()`	Trả về tên của thread
`getPriority()`	Trả về quyền ưu tiên của thread.
`isAlive()`	Xác định thread nào đang chạy
`join()`	Tạm dừng cho đến khi thread kết thúc
`run()`	Danh mục cần thực hiện bên trong thread
`sleep()`	Suspends một thread. Method này cho phép bạn xác định khoảng thời gian mà thread được cho tạm dừng
`start()`	Bắt đầu thread.

VI - Main thread

Mỗi chương trình Java có một thread, thậm chí nếu bạn không tạo ra bất kỳ thread nào. Thread này được gọi là main thread bởi vì nó là thread thực thi khi bạn bắt đầu chương trình của bạn. Main thread sinh ra các thread mà bạn tạo ra. Những thread đó gọi là child thread. Main thread luôn luôn là thread cuối cùng kết thúc việc thực thi bởi vì thông thường main thread cần giải phóng tài nguyên được sử dụng bởi chương trình chẳng hạn như các kết nối mạng.

Các lập trình viên có thể điều khiển main thread bằng cách đầu tiên tạo ra một đối tượng thread và sau đó sử dụng các method của đối tượng thread để điều khiển main thread. Bạn tạo đối tượng thread bằng cách gọi method currentThread(). Method currentThread() trả về một reference (tham chiếu) đến thread, sau đó bạn sử dụng reference này để điều khiển main thread như bất kỳ thread nào khác. Để tạo một reference đến main thread và đổi tên của thread từ main thành Demo Thread. Chương trình dưới đây chỉ ra làm thế nào để làm được điều này. Màn hình hiển thị khi chương trình chạy

Current thread: Thread[main, 5,main]

Renamed Thread: Thread[Demo Thread, 5,main]

Đoạn mã dưới đây sẽ thể hiện điều đó.

```
class Demo {
public static void main (String args[] ) {
Thread t = Thread.currentThread();
System.out.println("Current thread: " + t);
t.setName("Demo Thread");
System.out.println("Renamed Thread: " + t);
}
}
```

Như bạn đã học trong chương này, một thread tự động được tạo ra khi bạn thực thi một chương trình. Mục đích của ví dụ này là công bố một reference đến một thread và gán nó cho main thread. Điều này được thực hiện trong dòng đầu tiên của method main(). Chúng ta công bố reference bằng việc xác định tên của lớp và tên cho reference. Điều này được thực hiện nhờ dòng mã

Thread t = Thread.currentThread()

Chúng ta có được một reference đến main thread bằng việc gọi method currentThread() của class Thread. Reference trả về bởi method currentThread() sau đó được gán cho reference được công bố trước đó trong phần mở đầu phát biểu.
Thread[main, 5,main]

Thông tin bên trong cặp ngoặc vuông nói cho chúng ta biết vài thông tin về thread. Từ main xuất hiện đầu tiên là tên của thread. Số 5 là quyền ưu tiên của thread, là quyền ưu tiên thông thường. Quyền ưu tiên nằm trong phạm vi từ 1 đến 10, trong đó 1 là thấp nhất và 10 là cao nhất. Từ main nằm cuối cùng là tên của nhóm thread mà thread đó thuộc về. Một nhóm các thead là một cấu trúc dữ liệu được sử dụng để điều khiển trạng thái của một tập hợp các thread. Bạn không cần quan tâm đến nhóm thread bởi vì môi trường Java run-time xử lý điều này.
Method setName() được gọi chỉ ra làm thế nào để bạn điều khiển main thread của chương trình của bạn.Method setName() là một method thành viên của class Thread và được sử dụng để thay đổi tên của một thread. Ví dụ này sử dụng setName() để thay đổi tên của main thread từ main thành Demo Thread. Thread hiển thị một lần nữa trên màn hình với tên đã được thay đổi.

VII - Tạo các thread của riêng bạn

Nhớ rằng chương trình của bạn là main thread, và các phần khác của chương trình của bạn có thể cũng là một thread. Bạn có thể thiết kế một phần của chương trình của bạn là thread bằng việc tạo ra thread của riêng bạn. Cách dễ dàng nhất để làm điều này là bổ sung interface Runnable. Việc bổ sung interface Runnable là một lựa chọn để các lớp của bạn kế thừa class Thread.
Một interface mô tả một hay nhiều method thành viên mà bạn phải định nghĩa trong class của bạn theo quy định tuân theo interface. Những method này được mô

tả với tên method, danh sách đối số và giá trị trả về.
Interface Runnable mô tả các method của lớp cần tạo và tương tác với một thread. Theo quy định, để sử dụng interface trong class của bạn, bạn phải định nghĩa các method được mô tả trong interface Runnable. Khá thuận lợi, bạn chỉ cần định nghĩa một method được mô tả bởi interface Runnable – method run(). Method run() phải là một method public, và nó không yêu cầu danh sách đối số cũng như giá trị trả về.

Nội dung của method run() là một phần của chương trình bạn sẽ trở thành một thread mới. Các phát biểu bên ngoài method run() là thuộc về main thread. Các phát biểu bên trong method run() thuộc về thread mới. Cả main thread và thread mới chạy cùng một lúc khi bạn bắt đầu thread mới. Bạn sẽ học điều này trong ví dụ kế tiếp. Thread mới kết thúc khi method run() kết thúc. Điều khiển sau đó trả về cho phát biểu gọi method run().

Khi bạn bổ sung interface Runnable, bạn sẽ cần gọi khởi dựng dưới đây của class Thread. Khởi dựng này yêu cầu hai đối số. Đối số đầu tiên là thực thể của lớp để bổ sung interface Runnable và nói cho khởi dựng biết thread được thực thi từ đâu. Đối số thứ hai là tên của thread mới. Đây là định dạng của khởi dựng:
Thread(Runnable class, String name)

Khởi dựng tạo ra một thread mới nhưng nó không bắt đầu thread. Bạn bắt đầu thread bằng cách gọi method start(). Method start() gọi method run() bạn định nghĩa trong chương trình của bạn. Method start() không có danh sách đối số và không có giá trị trả về. Ví dụ dưới đây chỉ ra làm thế nào để tạo ra và bắt đầu một thread mới. Đây là những gì hiển thị khi chương trình chạy.
Main thread started
Child thread started
Child thread terminated
Main thread terminated

```
class MyThread implements Runnable {
Thread t;
MyThread() {
t = new Thread(this,"My thread");
t.start();
}

public void run() {
System.out.println("Child thread started");
System.out.println("Child thread terminated");
}
}
```

```
class Demo {
public static void main(String args[]) {
new MyThread();
System.out.println("Main thread started");
System.out.println("Main thread terminated");
}
}
```

Ví dụ này bắt đầu bằng việc định nghĩa một class gọi là MyThread, bổ sung interface Runnable. Vì thế chúng ta sử dụng từ khóa implements để bổ sung interface Runnable. Kế tiếp, một reference đến thread được công bố. Tiếp đó là định nghĩa khởi dựng cho class. Khởi dựng này gọi khởi dựng của class Thread. Bởi vì chúng ta bổ sung interface Runnable, chúng ta cần đặt khởi dựng reference đến thực thể của class sẽ thực thi thread mới và tên của thread mới. Chú ý rằng, từ khóa this là một reference đến thực thể hiện hành của class.
Khởi dựng trả về một reference cho thread mới, và được gán cho reference được công bố ở phát biểu đầu tiên trong class MyThread. Chúng ta sử dụng reference này để gọi method start(). Nhớ rằng method start() gọi method run()
Kế đến chúng ta định nghĩa method run(). Những phát biểu bên trong method run() trở thành một phần của chương trình thực thi khi thread thực thi. Chỉ có hai phát biểu thể hiện trong method run().
Kế tiếp, chúng ta định nghĩa class chương trình. Class chương trình thực thi thread mới bằng việc gọi thực thể của class MyThread. Thực hiện điều này bằng cách gọi toán tử new và khởi dựng của class MyThread.
Cuối cùng chương trình kết thúc bằng việc hiển thị hai dòng trên màn hình.

Chương II - Multithreading trong Java

(Phần này tiếp tục trình bày cách tạo thread bằng việc sử dụng từ khóa extends và cách tạo quyền ưu tiên cho thread)

I - Tạo một Thread sử dụng extends

Bạn có thể kế thừa class Thread như một cách khác để tạo thread trong chương trình của bạn. Bằng việc sử dụng từ khóa extends khi định nghĩa class để kế thừa class khác, khi bạn công bố một thực thể của class bạn cũng được truy cập đến những thành viên của class Thread.
Bất cứ khi nào class của bạn kế thừa class Thread, bạn phải override (cài chồng) method run(), là một phần trong thread mới. Ví dụ dưới đây chỉ ra làm thế nào để kế thừa class Thread và override method run().
Ví dụ này định nghĩa class MyThread kế thừa class Thread. Khởi dựng của class MyThread gọi khởi dựng của class Thread bằng việc sử dụng từ khóa super và đặt vào đó tên của thread mới, chính là MyThread. Sau đó nó gọi method start() để

kích hoạt thread mới. Method start() gọi method run() của class MyThread. Bạn sẽ chú ý rằng trong ví dụ này, method run() được override bằng việc hiển thị hai dòng trên màn hình chỉ ra rằng child thread bắt đầu và kết thúc. Nhớ rằng các phát biểu bên trong method run() tạo nên phần chương trình chạy như thread. Vì thế chương trình của bạn sẽ có nhiều phát biểu hơn trong method run() hơn là trong ví dụ này. Thread mới được công bố bên trong method main() của class Demo, chính là class chương trình của ứng dụng. Sau khi thread bắt đầu, hai thông điệp hiển thị chỉ ra trạng thái của main thread

```
class MyThread extends Thread {
   MyThread(){
      super("My thread");
      start();
   }
   public void run() {
      System.out.println("Child thread started");
      System.out.println("Child thread terminated");
   }
}
class Demo {
   public static void main (String args[]){
      new MyThread();
      System.out.println("Main thread started");
      System.out.println("Main thread terminated");
   }
}
```

Chú ý rằng, bạn nên bổ sung interface Runnable nếu như chỉ có method run() là method của class Thread mà bạn cần override. Bạn nên kế thừa class Thread nếu như bạn cần override những method khác được định nghĩa trong class Thread.

II - Sử dụng nhiều thread trong một chương trình.

Không phải không thường cần phải chạy nhiều thực thể của một thread, chẳng hạn như chương trình của bạn in ra nhiều tài liệu cùng một lúc. Các lập trình viên gọi đây là spawning một thread. Bạn có thể sinh ra bất kỳ số lượng thread nào mà bạn cần bằng class của riêng bạn đã được định nghĩa đầu tiên có bổ sung interface Runnable hoặc kế thừa class Thread và sau đó công bố các thực thể của class. Mỗi thực thể là một thread mới.

Hãy xem điều này được thực hiện thế nào. Ví dụ kế tiếp định nghĩa một class gọi là MyThread bổ sung interface Runnable. Khởi dựng của MyThread chấp nhận một đối số, đó là một chuỗi được sử dụng như tên của thread mới. Chúng ta tạo ra thread mới bên trong khởi dựng bằng cách gọi khởi dựng của class Thread và truyền vào một reference đến đối tượng đang định nghĩa thread và tên của thread.

Nhớ rằng, từ khóa this là một reference đến đối tượng hiện tại. Method start() sau khi được gọi sẽ gọi method run().
Method run() được override trong class MyThread. Có hai điều xảy ra khi method run() thực thi. Đầu tiên tên của method hiển thị trên màn hình. Thứ hai, thread tạm dừng 2 giây khi method sleep() được gọi. Method sleep() được định nghĩa trong class Thread có thể chấp nhận một hoặc hai tham số. Tham số đầu tiên là số mili giây mà thread tạm dừng. Tham số thứ hai là số micro giây mà thread tạm dừng. Trong ví dụ này, chúng ta chỉ quan tâm đến mili giây vì thế chúng ta không cần tham số thứ hai (2000 nano giây là 2 giây). Sau khi thread tạm dừng, các phát biểu khác hiển thị trên màn hình bắt đầu là thread đang kết thúc.
Method main() của class Demo công bố bốn thực thể của cùng một thread bằng việc gọi khởi dựng của class MyThread và truyền vào đó tên của thread. Mỗi thread này được coi như một thread nhỏ. Main thread sau đó tạm dừng 10 giây bằng việc gọi method sleep(). Trong suốt thời gian này, các thread tiếp tục thực thi. Khi main thread quay lại, nó hiển thị thông điệp rằng main thread đang kết thúc.
Đây là những gì trên màn hình hiển thị khi ví dụ chạy

```
Thread: 1
Thread: 2
Thread: 3
Thread: 4
Terminating thread: 1
Terminating thread: 2
Terminating thread: 3
Terminating thread: 4
Terminating thread: main thread.
```

Mã nguồn của ví dụ:

```
class MyThread implements Runnable {
   String tName;
   Thread t;
   MyThread (String threadName) {
      tName = threadName;
      t = new Thread (this, tName);
      t.start();
   }
   public void run() {
      try {
         System.out.println("Thread: " + tName );
         Thread.sleep(2000);
      } catch (InterruptedException e ) {
         System.out.println("Exception: Thread "
```

```
            + tName + " interrupted");
        }
        System.out.println("Terminating thread: " + tName );
    }
}
class Demo {
    public static void main (String args []) {
        new MyThread ("1");
        new MyThread ("2");
        new MyThread ("3");
        new MyThread ("4");
        try {
            Thread.sleep (10000);
        } catch (InterruptedException e) {
            System.out.println("Exception: Thread main interrupted.");
        }
        System.out.println("Terminating thread: main thread.");
    }
}
```

III - Sử dụng isAlive() và join()

Thông thường, main thread là thread cuối cùng kết thúc trong một chương trình. Tuy nhiên, không phải là bảo đảm không có những trường hợp, main thread sẽ kết thúc trước khi một child thread kết thúc. Trong ví dụ trước, chúng tôi đã nói method main ngủ cho đến khi các child thread kết thúc. Tuy nhiên chúng tôi đã ước lượng khoảng thời gian nó thực hiện để các child thread hoàn tất xử lý. Nếu khoảng thời gian ước lượng quá ngắn, một child thread có thể kết thúc sau khi main thread đã kết thúc. Vì thế, kỹ thuật sleep không phải là cách tốt nhất để bảo đảm rằng main thread kết thúc cuối cùng.

Các lập trình viên sử dụng hai cách khác để xác định rằng main thread là thread cuối cùng kết thúc. Những kỹ thuật này bao gồm việc gọi method isAlive() và method join(). Cả hai method này đều được định nghĩa trong class Thread. Method isAlive() xác định còn method nào đang chạy hay không. Nếu còn, isAlive() trả về giá trị Boolean true. Ngược lại, Boolean false được trả về. Bạn sử dụng method isAlive() để xác định còn child method nào tiếp tục chạy hay không. Method join() chờ cho đến khi child thread kết thúc và "kết nối" main thread. Ngoài ra, bạn có thể sử dụng method join() để xác định lượng thời gian mà bạn muốn chờ một child thread kết thúc.

Ví dụ dưới đây chỉ ra sử dụng method isAlive() và method join() trong chương trình của bạn như thế nào. Ví dụ này gần giống ví dụ trước. Sự khác biết nằm ở method main() của class Demo.

Sau khi các thread được công bố sử dụng khởi dựng của class MyThread. Method

isAlive() được gọi cho mỗi thread. Giá trị trả về của method isAlive() được hiển thị trên màn hình. Kế tiếp method join() được gọi cho mỗi thread. Method join() làm cho main thread chờ tất cả các child thread hoàn tất xử lý trước khi main thread kết thúc. Đây là những gì hiển thị trên màn hình khi chương trình chạy.

```
Thread Status: Alive
Thread 1: true
Thread 2: true
Thread 3: true
Thread 4: true
Threads Joining.
Thread: 1
Thread: 2
Thread: 3
Thread: 4
Terminating thread: 1
Terminating thread: 2
Terminating thread: 3
Terminating thread: 4
Thread Status: Alive
Thread 1: false
Thread 2: false
Thread 3: false
Thread 4: false
Terminating thread: main thread.
```

Mã nguồn

```
class MyThread implements Runnable {
   String tName;
   Thread t;
   MyThread (String threadName) {
      tName = threadName;
      t = new Thread (this, tName);
      t.start();
   }
   public void run() {
      try {
         System.out.println("Thread: " + tName );
         Thread.sleep(2000);
      } catch (InterruptedException e ) {
         System.out.println("Exception: Thread "
            + tName + " interrupted");
```

```
        }
        System.out.println("Terminating thread: " + tName );
    }
}
class Demo {
    public static void main (String args []) {
        MyThread thread1 = new MyThread ("1");
        MyThread thread2 = new MyThread ("2");
        MyThread thread3 = new MyThread ("3");
        MyThread thread4 = new MyThread ("4");
        System.out.println("Thread Status: Alive");
        System.out.println("Thread 1: " + thread1.t.isAlive());
        System.out.println("Thread 2: " + thread2.t.isAlive());
        System.out.println("Thread 3: " + thread3.t.isAlive());
        System.out.println("Thread 4: " + thread4.t.isAlive());
        try {
            System.out.println("Threads Joining.");
            thread1.t.join();
            thread2.t.join();
            thread3.t.join();
            thread4.t.join();
        } catch (InterruptedException e) {
            System.out.println("Exception: Thread main interrupted.");
        }
        System.out.println("Thread Status: Alive");
        System.out.println("Thread 1: " + thread1.t.isAlive());
        System.out.println("Thread 2: " + thread2.t.isAlive());
        System.out.println("Thread 3: " + thread3.t.isAlive());
        System.out.println("Thread 4: " + thread4.t.isAlive());
        System.out.println("Terminating thread: main thread.");
    }
}
```

IV - Cài đặt quyền ưu tiên của Thread

Trong phần trước, bạn đã học được mỗi thread có một quyền ưu tiên được gán được sử dụng những thread quan trọng hơn sử dụng tài nguyên. Quyền ưu tiên được sử dụng như một hướng dẫn cho hệ điều hành xác định được thread nào được truy cập đến một tài nguyên chẳng hạn như CPU. Trong thực tế, hệ điều hành đặt các trình quản lý vào trong sự quản lý của nó. Thông thường các lập trình viên có một ít hoặc không có quyền điều khiển đối với những trình quản lý khác. Vì thế, chúng thành lập quyền ưu tiên cho các thread của chúng mà không quan tâm xa hơn đến những trình quản lý khác.

Một quyền ưu tiên là một số nguyên chính xác từ 1 đến 10, trong đó 10 là quyền ưu tiên cao nhất được xem như quyền ưu tiên tối đa, và 1 là quyền ưu tiên thấp nhất được xem như quyền ưu tiên tối thiểu. Quyền ưu tiên thông thường là 5, là quyền ưu tiên mặc định cho mỗi thread.
Nói chung, một thread có quyền ưu tiên cao hơn giành việc sử dụng tài nguyên của thread có quyền ưu tiên thấp hơn. Thread có quyền ưu tiên thấp hơn tạm dừng cho đến khi thread có quyền ưu tiên cao hơn kết thúc việc sử dụng tài nguyên. Bất kỳ khi nào, có hai thread có quyền ưu tiên bằng nhau cần cùng tài nguyên cùng một thời điểm, thread nào truy cập tài nguyên đầu tiên giành được quyền sử dụng tài nguyên. Những gì xảy ra cho thread thứ hai phụ thuộc vào hệ điều hành mà bạn đang chạy chương trình. Một số hệ điều hành bắt thread thứ hai chờ cho đến khi thread thứ nhất kết thúc sử dụng tài nguyên. Hệ điều hành khác yêu cầu thread đầu tiên để thread thứ hai truy cập tài nguyên sau một khoảng thời gian xác định. Điều này để chắc chắn rằng một thread không chiếm dụng tài nguyên và ngăn các thread khác sử dụng nó.
Trong thế giới thực, thread đầu tiên thường tạm dừng trong khi sử dụng tài nguyên bởi vì tài nguyên khác nó cần không có sẵn. Trong suốt thời gian tạm dừng đó, hệ điều hành để thread đầu tiên giải phóng tài nguyên. Vấn đề là bạn không biết khi nào thời gian tạm dừng xảy ra. Do đó tốt nhất là luôn luôn để một thread tạm dừng sau một khoảng thời gian bất cứ khi nào thread đang sử dụng tài nguyên suốt một khoảng thời gian dài. Theo cách này, thread chia sẻ tài nguyên với những thread khác.
Bạn cần nhớ rằng có một mặt tiêu cực khi tạm dừng một thread trong một khoảng thời gian một cách tự động. Việc dừng một thread làm giảm đi khả năng thực thi của chương trình của bạn và có thể gây ra một lỗi backlog trong việc sử dụng tài nguyên. Do đó bạn cần quan sát việc thực thi chương trình của bạn một cách đều đặn để chắc chắn rằng bạn không phải chịu đựng sự ảnh hưởng tiêu cực của việc dừng một thread.
Bây giờ, hãy tập trung vào những gì bạn làm để điều khiển – ấn định quyền ưu tiên của một thread. Bạn ấn định quyền ưu tiên của một thread bằng cách gọi method setPriority(), được định nghĩa trong class Thread. Method setPriority() yêu cầu một tham số, đó là một số nguyên thay thế cho mức độ của quyền ưu tiên. Bạn có hai cách để thay thế quyền ưu tiên. Bạn có thể sử dụng một số nguyên từ 1 đến 10, hoặc bạn có thể sử dụng các biến final được định nghĩa trong class Thread. Các biến này là MAX_PRIORITY, MIN_PRIORITY và NORM_PRIORITY.
Bạn có thể xác định được mức độ ưu tiên của thread bằng method getPriority(), cũng được định nghĩa trong class Thread. Method getPriority() không yêu cầu tham số nào, và nó trả về một số nguyên thay thế cho mức độ của quyền ưu tiên của thread.
Ví dụ dưới đây chỉ ra làm thế nào để sử dụng method setPriority() và method getPriority(). Ví dụ này tạo ra hai child thread và ấn định quyền ưu tiên cho mỗi thread. Đầu tiên thread có quyền ưu tiên thấp bắt đầu, và sau đó là thread có quyền

ưu tiên cao. Dưới đây là những gì hiển thị khi chạy chương trình (chú ý rằng thread có quyền ưu tiên cao chạy trước thread có quyền ưu tiên thấp mặc dù thread có quyền ưu tiên thấp bắt đầu trước

low priority started
high priority started
high priority running.
low priority running.
low priority stopped.
high priority stopped.

Mã nguồn:

```
class MyThread implements Runnable {
   Thread t;
   private volatile boolean running = true;
   public MyThread (int p, String tName) {
      t = new Thread(this,tName);
      t.setPriority (p);
   }
   public void run() {
      System.out.println(t.getName() + " running.");
   }
   public void stop() {
      running = false;
      System.out.println(t.getName() + " stopped.");
   }
   public void start() {
      System.out.println(t.getName() + " started");
      t.start();
   }
}
class Demo {
   public static void main(String args[] ) {
      Thread.currentThread().setPriority(10);
      MyThread lowPriority = new MyThread (3, "low priority");
      MyThread highPriority = new MyThread (7, "high priority");
      lowPriority.start();
      highPriority.start();
      try {
         Thread .sleep(1000);
      } catch ( InterruptedException e) {
```

```
            System.out.println("Main thread interrupted.");
        }
        lowPriority.stop();
        highPriority.stop();
        try {
            highPriority.t.join();
            lowPriority.t.join();
        } catch (InterruptedException e) {
            System.out.println("InterruptedException caught");
    }
}
}
```

Chương III - Multithreading (đa tuyến) trong Java

(*Ảnh hưởng quan trọng khi hai hay nhiều hơn các thread chia sẻ cùng tài nguyên đó là chỉ có một trong số chúng có thể truy cập vào tài nguyên tại một thời điểm. Các lập trình viên xác định việc này bằng việc đồng bộ hoá các thread.)*

I - Đồng bộ hoá các Thread

Ảnh hưởng quan trọng khi hai hay nhiều hơn các thread chia sẻ cùng tài nguyên đó là chỉ có một trong số chúng có thể truy cập vào tài nguyên tại một thời điểm. Các lập trình viên xác định việc này bằng việc đồng bộ hoá các thread.

Các thread được đồng bộ hoá trong Java sử dụng thông qua một monitor. Hãy nghĩ rằng, một monitor là một object cho phép một thread truy cập vào một tài nguyên. Chỉ có một thread sử dụng một monitor vào bất kỳ một khoảng thời gian nào. Các lập trình viên nói rằng, các thread sở hữu monitor vào thời gian đó. Monitor cũng được gọi là một semaphore.

Một thread có thể sở hữu một monitor chỉ nếu như thread khác không sở hữu monitor. Nếu monitor có sẵn, một thread có thể sở hữu monitor và truy cập thẳng đến tài nguyên được tập hợp với monitor đó. Nếu monitor không có sẵn, thread sẽ bị suspend cho đến khi monitor trở thành có sẵn. Các lập trình viên nói rằng thread đang chờ monitor.

Cuối cùng, tác nhiệm của việc yêu cầu một monitor xảy ra đằng sau "màn chắn" trong Java. Java xử lý tất cả các chi tiết đó cho bạn. Bạn phải đồng bộ hoá các thread trong chương trình của bạn nếu như có nhiều hơn một thread sử dụng cùng một tài nguyên.

Bạn có hai cách để đồng bộ hoá các thread: bạn sử dụng method được đồng bộ hóa hoặc statement được đồng bộ hóa.

II - Sử dụng method được đồng bộ hóa

Tất cả các object trong Java có một monitor. Một thread có một monitor bất kỳ khi nào một method được bổ sung từ khóa synchronized được gọi. Thread lần đầu tiên gọi method được đồng bộ hóa được nói nằm bên trong method và sở hữu method đó cũng như tài nguyên sử dụng bởi method. Các thread khác gọi method được đồng bộ hoá chờ cho đến khi thread đầu tiên giải phóng method được đồng bộ hóa.

Nếu method được đồng bộ hoá là một instance method, method được đồng bộ hóa kích hoạt khóa đi kèm với instance được gọi là method được đồng bộ hóa đó chính là object được biết là this trong suốt quá trình thực thi toàn bộ method. Nếu method được đồng bộ hóa là static thì nó kích hoạt khóa đi kèm với class định nghĩa method được đồng bộ hoá.

Trước khi bạn học cách làm thế nào để định nghĩa một method đồng bộ hóa

trong chương trình của bạn, hãy xem những gì xảy ra nếu việc đồng bộ hoá không được sử dụng trong một chương trình. Đây là mục đích của ví dụ dưới đây, chương trình sẽ hiển thị hai tên bên trong dấu ngoặc đơn sử dụng hai thread. Đây là tiến trình ba bước, bao gồm mở ngoặc đơn, tên và đóng ngoặc đơn hiển thị trong từng bước.

Ví dụ định nghĩa ba lớp, class Parentheses, class MyThread và class Demo cũng là class chương trình. Class Parentheses định nghĩa một method display(), nhận vào một chuỗi trong danh sách đối số của nó và hiển thị chuỗi trong ngoặc đơn trên màn hình. Class MyThread định nghĩa một thread. Trong khi làm điều này, khởi dựng của MyThread yêu cầu hai đối số. Đối số đầu tiên là một reference đến một instance của class Parentheses. Đối số thứ hai à một chuỗi chứa tên sẽ hiển thị trên màn hình. Method run() sử dụng instance của class Parentheses để gọi method display của nó, truyền vào cho method display() tên để nó xuất hiện trên màn hình.

Phần còn lại hoạt động trong method main() của class Demo. Phát biểu đầu tiên công bố một instance của class Parentheses. Hai dòng kế tiếp tạo ra hai thread. Chú ý rằng cả hai thread này sử dụng cùng một instance của class Parentheses. Đây là những gì hiển thị khi bạn chạy chương trình. Có thể sẽ không như bạn mong đợi. Mỗi tên nên được bao trong cặp ngoặc đơn của chúng. Vấn đề là method display() không đồng bộ hoá.

```
class Parentheses {
   void display(String s) {
      System.out.print ("(" + s);
      try {
         Thread.sleep (1000);
      } catch (InterruptedException e) {
         System.out.println ("Interrupted");
      }
      System.out.println(")");
   }
}

class MyThread implements Runnable {
   String s1;
   Parentheses p1;
   Thread t;
   public MyThread (Parentheses p2, String s2) {
      p1= p2;
      s1= s2;
      t = new Thread(this);
      t.start();
   }
```

```
   public void run() {
      p1.display(s1);
   }
}

class Demo{
   public static void main (String args[]) {
      Parentheses p3 = new Parentheses();
      MyThread name1 = new MyThread(p3, "Bob");
      MyThread name2 = new MyThread(p3, "Mary");
      try {
         name1.t.join();
         name2.t.join();
      } catch (InterruptedException e ) {
         System.out.println( "Interrupted");
      }
   }
}
```

Vấn đề trong ví dụ trên là các thread cùng chia sẽ tài nguyên cùng một lúc. Tài nguyên được method display() định nghĩa trong class Parentheses. Theo quy định để một thread đặt điều khiển đối với method display() chúng ta phải đồng bộ hóa method display(). Điều này được thực hiện bằng cách sử dụng từ khóa synchronized đặt ở đầu method display(). Đây là những gì bạn muốn thấy trong ví dụ trước.

```
class Parentheses {
   synchronized void display(String s) {
      System.out.print ("(" + s);
      try {
         Thread.sleep (1000);
      } catch (InterruptedException e) {
         System.out.println ("Interrupted");
      }
      System.out.println(")");
   }
}

class MyThread implements Runnable {
   String s1;
   Parentheses p1;
   Thread t;
   public MyThread (Parentheses p2, String s2) {
```

```
        p1= p2;
        s1= s2;
        t = new Thread(this);
        t.start();
    }
    public void run() {
        p1.display(s1);
    }
}

class Demo{
    public static void main (String args[]) {
        Parentheses p3 = new Parentheses();
        MyThread name1 = new MyThread(p3, "Bob");
        MyThread name2 = new MyThread(p3, "Mary");
        try {
            name1.t.join();
            name2.t.join();
        } catch (InterruptedException e ) {
            System.out.println( "Interrupted");
        }
    }
}
```

Việc đồng bộ hóa một method là cách tốt nhất để hạn chế việc sử dụng một method tại một thời điểm. Tuy nhiên sẽ có những trường hợp mà bạn không thể đồng bộ hóa một method, chẳng hạn như khi bạn sử dụng một class được cung cấp bởi bên thứ ba. Trong những trường hợp như thế, bạn không được phép truy cập vào định nghĩa lớp, sẽ ngăn bạn sử dụng từ khóa synchronized.

III - Sử dụng phát biểu được đồng bộ hoá

Một cách khác để sử dụng từ khóa synchronized là sử dụng phát biểu được đồng bộ hóa. Một phát biểu được đồng bộ hóa chứa một block được đồng bộ hóa , mà bên trong đó đặt những đối tượng và những method được đồng bộ hóa. Gọi các method chứa block được đồng bộ hóa xảy ra khi một thread có được monitor của đối tượng.

Mặc dù bạn có thể gọi những method bên trong một block được đồng bộ hóa, việc công bố method phải được thực hiện bên ngoài một block được đồng bộ hóa. Ví dụ dưới đây chỉ cách làm thế nào để sử dụng một phát biểu được đồng bộ hóa. Ví dụ này cơ bản cũng giống ví dụ trước, tuy nhiên phát biểu được đồng bộ hóa được sử dụng thay vì từ khóa synchronized. Phát biểu này được đặt trong method run() của class MyThread. Phát biểu được đồng bộ hóa sẽ đồng bộ hóa instance của class Parentheses và vì thế ngăn hai thread sử dụng method display() cùng một lúc.

```
class Parentheses {
   void display(String s) {
      System.out.print ("(" + s);
      try {
         Thread.sleep (1000);
      } catch (InterruptedException e) {
         System.out.println ("Interrupted");
      }
      System.out.println(")");
   }
}

class MyThread implements Runnable {
   String s1;
   Parentheses p1;
   Thread t;
   public MyThread (Parentheses p2, String s2) {
      p1= p2;
      s1= s2;
      t = new Thread(this);
      t.start();
   }
   public void run() {
      synchronized(p1){
```

```
            p1.display(s1);
        }
    }
}

class Demo{
    public static void main (String args[]) {
        Parentheses p3 = new Parentheses();
        MyThread name1 = new MyThread(p3, "Bob");
        MyThread name2 = new MyThread(p3, "Mary");
        try {
            name1.t.join();
            name2.t.join();
        } catch (InterruptedException e ) {
            System.out.println( "Interrupted");
        }
    }
}
```

Ở đây, method display() không sử dụng từ khóa synchronized. Thay vào đó, phát biểu được đồng bộ hóa được sử dụng bên trong method run(). Điều này cho kết quả giống với ví dụ trước bởi vì một thread chờ một khoảng thời gian để thread còn lại kết thúc trước khi tiếp tục xử lý.

IV - Giao tiếp giữa các thread

Các thread mở ra cho các lập trình viên một khoảng không mới trong lập trình, nơi mà ở đó những phần của một chương trình thực thi không đồng bộ với nhau, mỗi một xử lý độc lập với những xử lý khác. Tuy nhiên mỗi thread thỉnh thoảng cần tính toán việc xử lý của chúng và vì thế cần có thể giao tiếp với những thread khác trong suốt quá trình xử lý. Các lập trình viên gọi đây là inter-process communication (giao tiếp trong xử lý).

Bạn có thể có các thread giao tiếp với các thread khác trong chương trình của bạn bằng cách sử dụng những method wait(), notify() và notifyAll(). Những method này được gọi từ bên trong một method được đồng bộ hóa. Method wait() nói cho một thread giải phóng monitor và đi vào trạng thái suspend. Có hai dạng method wait() khác nhau. Một dạng không yêu cầu đối số và vì thế một thread sẽ chờ cho đến khi nó được thông báo. Một dạng khác có đối số để bạn xác định khoảng thời gian chờ. Bạn xác định độ dài thời gian trong mili giây và đặt nó vào trong method wait().

Method notify() nói cho một thread đang suspend bởi method wait() và lấy lại điều khiển của monitor. Method notifyAll() đánh thức tất cả các thread đang chờ điều khiển của monitor. Những thread khác chờ trong trạng thái suspend cho đến khi monitor có sẵn trở lại.

Ví dụ dưới đây chỉ cho bạn làm thế nào để sử dụng những method này trong một ứng dụng. Mục đích của chương trình là có một class Pulishser cho một giá trị cho class Consumer thông qua sử dụng class Queue. Ví dụ này định nghĩa bốn class, class Pulisher, class Comsumer, class Queue và class Demo. Class Queue định nghĩa hai instance: exchangeValue và một biến cờ. exchangeValue đặt vào một giá trị trong queue bởi publisher. Biến cờ được sử dụng như một cách đánh dấu giá trị được đặt vào trong queue. Class Queue cũng định nghĩa một method get() và một method put(). Method put() sử dụng để đặt một giá trị vào queue (gán một giá trị cho exchangeValue), method get() sử dụng để nhận giá trị chứa trong queue (trả về giá trị của exchangeValue. Khi một giá trị được gán, method put() thay đổi giá trị của biến cờ từ false thành true xác định một giá trị được đặt vào trong queue. Chú ý giá trị của biến cờ được sử dụng như thế nào trong method get() và method put() để có thread gọi method chờ cho đến khi có một giá trị trong queue hoặc không có giá trị nào trong queue, phụ thuộc vào method nào đang được gọi.

Class Publisher công bố một instance của class Queue và sau đó gọi method put() đặt vào năm số nguyên integer trong queue. Mặc dù method put() được đặt trong một vòng lặp for, mỗi số nguyên integer được đặt vào trong queue, và sau đó có một khoảng tạm dừng cho đến khi số nguyên integer được nhận bởi class Consumer.

Class Consumer tương tự như thiết kế class Publisher, ngoại trừ class Consumer gọi method get() năm lần bên trong một vòng lặp for. Mỗi lần gọi, method get() tạm dừng cho đến khi class Publisher đặt một số nguyên integer vào trong queue.

Method main() của class Demo tạo ra các instance của class Publisher, class Consumer và class Queue. Chú ý rằng các khởi dựng của của class Publisher và class Consumer đều đặt vào một reference đến instance của class Queue. Chúng sử dụng instance của class Queue cho inter-process communication.

Dưới đây là những gì mà bạn sẽ thấy khi chạy chương trình và mã nguồn của ví dụ

```
Put: 0
Get: 0
Put: 1
Get: 1
Put: 2
Get: 2
Put: 3
Get: 3
Put: 4
Get: 4
```

```
class Queue {
   int exchangeValue;
   boolean busy = false;
   synchronized int get() {
```

```
        if (!busy)
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println(
                    "Get: InterruptedException");
            }
            System.out.println("Get: " + exchangeValue);
            notify();
            return exchangeValue;
    }
    synchronized void put (int exchangeValue) {
        if (busy)
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println(
                    "Put: InterruptedException");
            }
            this.exchangeValue = exchangeValue;
            busy = true;
            System.out.println("Put: " + exchangeValue);
            notify();
    }
}

class Publisher implements Runnable {
    Queue q;
    Publisher(Queue q) {
        this.q = q;
        new Thread (this, "Publisher").start();
    }
    public void run() {
        for (int i = 0; i < 5; i++){
            q.put(i);
        }
    }
}

class Consumer implements Runnable {
    Queue q;
    Consumer (Queue q) {
```

```
        this.q = q;
        new Thread (this, "Consumer").start();
    }
    public void run() {
        for (int i = 0; i < 5; i++){
            q.get();
        }
    }
}

class Demo {
    public static void main(String args []) {
        Queue q = new Queue ();
        new Publisher (q);
        new Consumer (q);
    }
}
```

Chương V - Multithread trong Java

(Có thể có nhiều lần bạn muốn dừng tạm thời một thread đang xử lý và sau đó phục hồi lại xử lý, chẳng hạn như bạn muốn một thread khác sử dụng tài nguyên hiện thời. Bạn có thể đạt được mục đích này bằng việc định nghĩa những method suspend và resume của chính bạn như được chỉ ra trong ví dụ dưới đây)

I - Suspending và Resuming Thread

Ví dụ này định nghĩa một class MyThread. Class MyThread định nghĩa ba method: method run(), method suspendThread(), và method resumeThread(). Thêm vào đó, class MyThread công bố biến instance suspended, mà giá trị được sử dụng để chỉ ra thread có suspend hay không.

Method run() chứa một vòng lặp for để hiển thị giá trị của biến counter. Mỗi lần biến counter được hiển thị, thread dừng một khoảng thời gian.Sau đó là một phát biểu synchronized để xác định giá trị của biến instance suspended là true hay không. Nếu có, method wait() được gọi, làm cho thread đi vào trạng thái suspend cho đến khi method notify() được gọi.

Method suspendThread() đơn giản chỉ gán giá trị true cho biến instance suspended. Method resumeThread() gán false cho biến instance suspended và sau đó gọi method notify(). Điều này sẽ làm cho thread đang ở trạng thái suspend phục hồi xử lý.

Method main() của class Demo công bố một instance của MyThread và sau đó tạm dừng khoảng 1 giây trước khi gọi method suspendThread() và hiển thị thông điệp tương ứng trên màn hình. Sau đó nó lại tạm dừng một khoảng thời gian trước khi gọi method resumeThread() và hiển thị thông điệp tương ứng trên màn hình một lần nữa.

Thread tiếp tục hiển thị giá trị của biến đếm counter cho đến khi thread đã đi vào trạng thái suspend. Thread tiếp tục hiển thị giá trị của biến counter một khi thread phục hồi xử lý. Đây là những gì hiển thị khi bạn chạy chương trình này và mã nguồn

```
Thread: 0
Thread: 1
Thread: 2
Thread: 3
Thread: 4
Thread: Suspended
Thread: Resume
Thread: 5
Thread: 6
Thread: 7
Thread: 8
```

Thread: 9
Thread exiting.

```
class MyThread implements Runnable {
   String name;
   Thread t;
   boolean suspended;
   MyThread() {
      t = new Thread(this, "Thread");
      suspended = false ;
      t.start();
   }

   public void run() {
      try {
         for (int i = 0; i < 10; i++) {
            System.out.println("Thread: " + i );
            Thread.sleep(200);
            synchronized (this) {
               while (suspended) {
                  wait();
               }
            }
         }
      } catch (InterruptedException e ) {
         System.out.println("Thread: interrupted.");
      }
      System.out.println("Thread exiting.");
   }

   void suspendThread() {
      suspended = true;
   }
   synchronized void resumeThread() {
      suspended = false;
      notify();
   }
}

class Demo {
   public static void main (String args [] ) {
      MyThread t1 = new MyThread();
      try{
```

```
            Thread.sleep(1000);
            t1.suspendThread();
            System.out.println("Thread: Suspended");
            Thread.sleep(1000);
            t1.resumeThread();
            System.out.println("Thread: Resume");
        } catch ( InterruptedException e) {
        }
        try {
            t1.t.join();
        } catch ( InterruptedException e) {
            System.out.println ("Main Thread: interrupted");
        }
    }
}
```

Chương VI - Sử dụng thread trong Swing như thế nào ?

(Phần này thảo luận về hai cách thể hiện để sử dụng thread. Đầu tiên, chúng ta thảo luận về các thread như cách chúng áp dụng cho mỗi ứng dụng Swing, tại sao tất cả mã với những thành phần khác nhau cần phải thực thi trên một thread cụ thể. Chúng ta cũng thảo luận về các method invokeLater và invokeAndWait, được sử dụng để chắc chắn mã thực thi trên thread cụ thể này. Phần còn lại áp dụng chỉ cho những ai muốn sử dụng thêm các thread để hoàn thiện khả năng thực thi trong ứng dụng của họ.)

I - Event-dispatching thread (Thread gởi đi sự kiện)

Xử lý sự kiện và thực thi mã Swing trong một thread, gọi là event-dispatching thread. Điều này chắc chắn rằng mỗi xử lý sự kiện kết thúc việc thực thi trước khi thực thi một sự kiện kế tiếp và vẽ lại component không bị gãy bởi các sự kiện. Để tránh khả năng tạo ra deadlock, bạn phải cực kỳ cẩn thận rằng các component Swing và các mô hình được tạo ra, được chỉnh sửa, được truy vấn chỉ từ event-dispatching thread.

Bạn cũng chú ý rằng demo sử dụng một phương thức chuẩn main để gọi method invokeLater của SwingUtilities để chắc chắn rằng GUI được tạo ra trên event-dispatching thread. Ở đây là một ví dụ của method main từ ví dụ FocusConceptsDemo. Chúng ta cũng có cả nguồn của method createAndShowGUI mà mỗi method gọi để việc tạo GUI được xử lý.

```
/*
* Tạo GUI và hiển thị nó. Để thread an toàn,
* phương thức này nên được gọi từ event-dispatching thread
*/
private static void createAndShowGUI() {
//Chắc chắn rằng chúng ta đã tạo một cửa sổ tốt
JFrame.setDefaultLookAndFeelDecorated(true);

//Tạo và cài đặt một cửa sổ
frame = new JFrame("FocusConceptsDemo");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//Tạo và cài đặt một content Pane
JComponent newContentPane = new FocusConceptsDemo();
newContentPane.setOpaque(true); //content panes must be opaque
frame.setContentPane(newContentPane);

//Hiển thị cửa sổ
```

```
frame.pack();
frame.setVisible(true);
}

public static void main(String[] args) {
//Tạo và hiển thị ứng dụng GUI
javax.swing.SwingUtilities.invokeLater(new Runnable() {
public void run() {
createAndShowGUI();
}
});
}
```

II - Sử dụng method invokeLater

Bạn có thể gọi invokeLater từ bất kỳ thread nào để yêu cầu event-dispatching thread chạy mã của bạn. Bạn phải đặt mã này trong method run của đối tượng Runnable và xác định đối tượng Runnable như một đối số của invokeLater. Method invokeLater trả về trực tiếp, mà không phải chờ đợi event-dispatching thread thực thi mã. Đây là ví dụ việc sử dụng invokeLater

```
Runnable updateAComponent = new Runnable() {
   public void run() { component.doSomething(); }
};

SwingUtilities.invokeLater(updateAComponent);
```

III - Sử dụng method invokeAndWait

Method invokeAndWait cũng tương tự như method invokeLater, ngoại trừ invokeAndWait không trả về cho đến khi event-dispatching thread đã thực thi mã cụ thể. Tốt hơn hết, bạn nên sử dụng method invokeLater thay vì method invokeAndWait – bởi rất dễ dàng gây ra một deadlock nếu sử dụng invokeAndWait. Nếu bạn sử dụng invokeAndWait, chắc chắn rằng thread gọi invokeAndWait không nắm giữ bất kỳ khóa nào để những thread khác có thể cần trong khi method gọi nó đang thực thi.

```
public void showHelloThereDialog() throws Exception {
   Runnable showModalDialog = new Runnable() {
      public void run() {
         JOptionPane.showMessageDialog(myMainFrame, "Hello There");
      }
```

```
    };
    SwingUtilities.invokeAndWait(showModalDialog);
}
```

Tương tụ, một thread cần truy cập đến trạng thái GUI, chẳng hạn như một cặp text fields, có thể có mã dưới đây

```
void printTextField() throws Exception {
    final String[] myStrings = new String[2];

    Runnable getTextFieldText = new Runnable() {
        public void run() {
            myStrings[0] = textField0.getText();
            myStrings[1] = textField1.getText();
        }
    };

    SwingUtilities.invokeAndWait(getTextFieldText);

    System.out.println(myStrings[0] + " " + myStrings[1]);
}
```

IV - Sử dụng thread để hoàn chỉnh khả năng thực thi

Khi được sử dụng đúng, thread có thể là một công cụ đầy sức mạnh. Tuy nhiên, bạn phải xử lý cẩn thận khi sử dụng thread trong một chương trình Swing. Mặc dù nguy hiểm, thread có thể là vô giá. Bạn có thể sử dụng chúng để hoàn thiện khả năng thực thi của chương trình của bạn. Và thỉnh thoảng, thread có thể đơn giản mã hoặc kiến trúc của một chương trình. Đây là một vài vị trí thông thường mà thread được sử dụng.

* Để di chuyển một tác nhiệm khởi tạo lãng phí thời gian lkhỏi thread main, để GUI có thể hiển thị nhanh hơn. Ví dụ thời gian lãng phí của các tác nhiệm như tính toán hay khóa mạng hoặc nhập xuất.
* Để gỡ bỏ một tác nhiệm khởi tạo lãng phí thời gian khỏi event-dispatching thread, để GUI trả lời nhanh hơn
* Để thực thi một hành động lặp lại, thường sử dụng nhiều khoảng thời gian giữa các hành động đó.
* Để chờ các thông điệp gởi từ các chương trình khác

Nếu bạn cần tạo ra thread, bạn có thể tránh một vài cái bẫy thông thương bằng việc bổ sung thread với một lớp tiện ích như SwingWorker hay một lớp Timer. Một đối tượng SwingWorker tạo ra một thread để thực thi hành động lãng phí thời gian.

Sau khi hành động này kết thúc, SwingWorker cho bạn lựa chọn thực thi các đoạn mã khác trong event-dispatching thread. Timer hữu dụng trong việc thực thi một tác nhiệm lặp lại hoặc sau một khoảng chờ xác định. Nếu bạn cần bổ sung thread của riêng bạn, bạn có thể tìm thêm thông tin để làm trong các sách thread khác. Bạn có thể sự dụng những kỹ thuật khác để tạo nên những chương trình Swing multi-thread làm việc tốt:

* Nếu bạn cần cập nhật một component nhưng mã của bạn không thực thi một event listener, sử dụng invokeLater (đề nghị) hoặc invokeAndWait
* Nếu bạn không chắc chắn rằng mã của bạn đang thực thi một event listener, bạn nên phân tích mã của chương trình và tài liệu để tạo thread mỗi method được gọi
* Nếu bạn cần cập nhật một component sau một khoảng thời gian, sự dụng một Timer để làm điều đó
* Nếu bạn cần cập nhật một component vào một thời điểm xác định, sử dụng một Timer

Một ví dụ:

Chúng tôi cung cấp cho các bạn một ví dụ cụ thể để so sánh việc tạo cảm quan nếu không dùng thread và nếu dùng thread khác nhau thế nào

Giả sử bạn có các menuItem mà khi click vào đó thì sẽ thay đổi cảm quan từ Windows sang Unix hay ngược lại.

```
private static final String macLaF =
"com.sun.java.swing.plaf.mac.MacLookAndFeel";
private static final String metalLaF =
"javax.swing.plaf.metal.MetalLookAndFeel";
private static final String motifLaF =
"com.sun.java.swing.plaf.motif.MotifLookAndFeel";
private static final String gtkLaF =
"com.sun.java.swing.plaf.gtk.GTKLookAndFeel";

/**
* A utility function that set look and feels
* @param lookAndFeel
*/
private void createLookAndFeel(String lookAndFeel) {
   try {
      UIManager.setLookAndFeel(lookAndFeel);
   } catch(Exception exception) {}
   javax.swing.SwingUtilities.updateComponentTreeUI(frame);
}
```

Nếu bạn không sử dụng thread thì sẽ

```
metalItem.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      createLookAndFeel(metalLaF);
   }
});
```

Còn nếu sử dụng thread

```
metalItem.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      javax.swing.SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            createLookAndFeel(metalLaF);
         }
      });
   }
});
```

Bạn sẽ thấy rõ rằng, nếu sử dụng thread, các component khi được vẽ lại không hề bị giật hay hiển thị không đồng bộ như lúc không dùng thread.

Kết thúc article này, bạn có thể thấy được lợi ích của thread trong Swing. Nó giúp tăng tốc độ cũng như khả năng thực thi mã của ứng dụng. Hy vọng là các bạn sẽ nắm bắt được điểm mạnh cũng như điểm yếu của thread trong Swing.