Giới thiệu

Thread là gì? Multi-thread là gì?

Thread (luồng) về cơ bản là một tiến trình con (sub-process). Một đơn vị xử lý nhỏ nhất của máy tính có thể thực hiện một công việc riêng biệt. Trong Java, các luồng được quản lý bởi máy ảo Java (JVM).

Multi-thread (đa luồng) là một tiến trình thực hiện nhiều luồng đồng thời. Một ứng dụng Java ngoài luồng chính có thể có các luồng khác thực thi đồng thời làm ứng dụng chạy nhanh và hiệu quả hơn.

VD: Trình duyệt web hay các chương trình chơi nhạc là 1 ví dụ điển hình về đa luồng.

+ Khi duyệt 1 trang web, có rất nhiều hình ảnh, CSS, javascript… được tải đồng thời bởi các luồng khác nhau.

+ Khi play nhạc, chúng ta vẫn có thể tương tác được với nút điều khiển như: Play, pause, next, back … vì luồng phát nhạc là luồng riêng biệt với luồng tiếp nhận tương tác của người dùng.

Đa nhiệm (multitasking)

Multitasking: Là khả năng chạy đồng thời một hoặc nhiều chương trình cùng một lúc trên một hệ điều hành. Hệ điều hành quản lý việc này và sắp xếp lịch phù hợp cho các chương trình đó. Ví dụ, trên hệ điều hành Windows chúng ta có làm việc đồng thời với các chương trình khác nhau như: Microsoft Word, Excel, Media Player, …

Chúng ta sử dụng đa nhiệm để tận dụng tính năng của CPU.

Đa nhiệm có thể đạt được bằng hai cách:

1. Đa nhiệm dựa trên đơn tiến trình (Process) – Đa tiến trình (Multiprocessing).
   * Mỗi tiến trình có địa chỉ riêng trong bộ nhớ, tức là mỗi tiến trình phân bổ vùng nhớ riêng biệt.
   * Tiến trình là nặng.
   * Sự giao tiếp giữa các tiến trình có chi phí cao.
   * Chuyển đổi từ tiến trình này sang tiến trình khác đòi hỏi thời gian để đăng ký việc lưu và tải các bản đồ bộ nhớ, các danh sách cập nhật, …
2. Đa nhiệm dựa trên luồng (Thread) – Đa luồng (MultiThreading).
   * Các luồng chia sẻ không gian địa chỉ ô nhớ giống nhau.
   * Luồng là nhẹ.
   * Sự giao tiếp giữa các luồng có chi phí thấp.

Đa tiến trình (multiprocessing) và đa luồng (multithreading) cả hai được sử dụng để tạo ra hệ thống đa nhiệm (multitasking). Nhưng chúng ta sử dụng đa luồng nhiều hơn đa tiến trình bởi vì các luồng chia sẻ một vùng bộ nhớ chung. Chúng không phân bổ vùng bộ nhớ riêng biệt để tiết kiệm bộ nhớ, và chuyển đổi ngữ cảnh giữa các luồng mất ít thời gian hơn tiến trình.

Ưu điểm của đa luồng

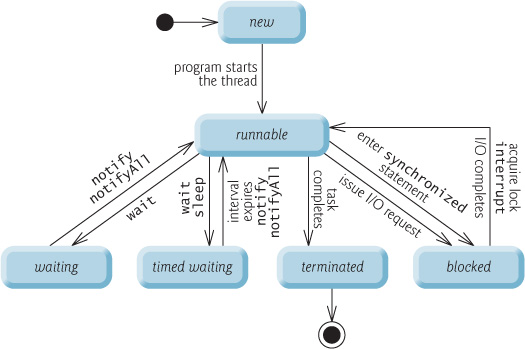
* Nó không chặn người sử dụng vì các luồng là độc lập và bạn có thể thực hiện nhiều công việc cùng một lúc.
* Mỗi luồng có thể dùng chung và chia sẻ nguồn tài nguyên trong quá trình chạy, nhưng có thể thực hiện một cách độc lập.
* Luồng là độc lập vì vậy nó không ảnh hưởng đến luồng khác nếu ngoại lệ xảy ra trong một luồng duy nhất.
* Có thể thực hiện nhiều hoạt động với nhau để tiết kiệm thời gian. Ví dụ một ứng dụng có thể được tách thành : luồng chính chạy giao diện người dùng và các luồng phụ nhiệm gửi kết quả xử lý đến luồng chính.

Nhược điểm của đa luồng

* Càng nhiều luồng thì xử lý càng phức tạp.
* Xử lý vấn đề về tranh chấp bộ nhớ, đồng bộ dữ liệu khá phức tạp.
* Cần phát hiện tránh các luồng chết (dead lock), luồng chạy mà không làm gì trong ứng dụng cả.

Deadlock (Khoá chết) là gì? Deadlock xảy ra khi 2 tiến trình đợi nhau hoàn thành, trước khi chạy. Kết quả của quá trình là cả 2 tiến trình không bao giờ kết thúc.

## Vòng đời (các trạng thái) của một Thread trong java

[](https://gpcoder.com/wp-content/uploads/2018/02/multithread-lifecylce.jpg)

Vòng đời của thread trong java được kiểm soát bởi JVM. Java định nghĩa các trạng thái của luồng trong các thuộc tính static của lớp **Thread.State**:

* **NEW**: Đây là trạng thái khi luồng vừa được khởi tạo bằng phương thức khởi tạo của lớp Thread nhưng chưa được start(). Ở trạng thái này, luồng được tạo ra nhưng chưa được cấp phát tài nguyên và cũng chưa chạy. Nếu luồng đang ở trạng thái này mà ta gọi các phương thức ép buộc stop,resume,suspend … sẽ là nguyên nhân sảy ra ngoại lệ IllegalThreadStateException .
* **RUNNABLE**: Sau khi gọi phương thức start() thì luồng test đã được cấp phát tài nguyên và các lịch điều phối CPU cho luồng test cũng bắt đầu có hiệu lực. Ở đây, chúng ta dùng trạng thái là Runnable chứ không phải Running, vì luồng không thực sự luôn chạy mà tùy vào hệ thống mà có sự điều phối CPU khác nhau.
* **WAITING** : Thread chờ không giới hạn cho đến khi một luồng khác đánh thức nó.
* **TIMED\_WAITING** : Thread chờ trong một thời gian nhất định, hoặc là có một luồng khác đánh thức nó.
* **BLOCKED**: Đây là 1 dạng của trạng thái “Not Runnable”, là trạng thái khi Thread vẫn còn sống, nhưng hiện tại không được chọn để chạy. Thread chờ một monitor để unlock một đối tượng mà nó cần.
* **TERMINATED**: Một thread ở trong trạng thái terminated hoặc dead khi phương thức run() của nó bị thoát.

## Cách tạo luồng trong Java

Trong java ta có thể tạo ra một luồng bằng một trong hai cách sau: tạo 1 đối tượng của lớp được **extend** từ **class Thread** hoặc **implements** từ **interface Runnable**.

### Tạo luồng bằng cách extend từ lớp Thread

Để tạo luồng bằng cách tạo lớp kế thừa từ lớp Thread, ta phải làm các công việc sau :

1. Khai báo 1 lớp mới kế thừa từ lớp Thread
2. Override lại phương thức run ở lớp này, những gì trong phương thức run sẽ được thực thi khi luồng bắt đầu chạy. Sau khi luồng chạy xong tất cả các câu lệnh trong phương thức run thì luồng cũng tự hủy.
3. Tạo 1 thể hiện (hay 1 đối tượng) của lớp ta vừa khai báo.
4. Sau đó gọi phương thức start() của đối tượng này để bắt đầu thực thi luồng.

|  |  |
| --- | --- |
|  |  |

**Lưu ý :**

* Tuy ta khai báo những công việc cần làm của luồng trong phương thức run() nhưng khi thực thi luồng ta phải gọi phương thức start(). Vì đây là phương thức đặc biệt mà java xây dựng sẵn trong lớp Thread, phương thức này sẽ cấp phát tài nguyên cho luồng mới rồi chạy phương thức run() ở luồng này. Vì vậy, nếu ta gọi phương thức run() mà không gọi start() thì cũng tương đương với việc gọi 1 phương thức của 1 đối tượng bình thường và phương thức vẫn chạy trên luồng mà gọi phương thức chứ không chạy ở luồng mới tạo ra, nên vẫn chỉ có 1 luồng chính làm việc chứ ứng dụng vẫn không phải là đa luồng.
* Sau khi start một thread, nó không bao giờ có thể được start lại. Nếu bạn làm như vậy, một ngoại lệ **IllegalThreadStateException** sẽ xảy ra.

### Tạo luồng bằng cách implement từ Interface Runnable

Để tạo luồng bằng cách hiện thực từ Interface Runnable, ta phải làm các công việc sau :

1. Khai báo 1 lớp mới implements từ Interface Runnable
2. Hiện thực phương thức run() ở lớp này, những gì trong phương thức run() sẽ được thực thi khi luồng bắt đầu chạy. Sau khi luồng chạy xong tất cả các câu lệnh trong phương thức run thì luồng cũng tự hủy.
3. Tạo 1 thể hiện (hay 1 đối tượng) của lớp ta vừa khai báo. (VD : Tên đối tượng là r1)
4. Tạo 1 thể hiện của lớp Thread bằng phương thức khởi tạo : Thread(Runnable target)
   * Runnable target: Là 1 đối tượng thuốc lớp được implements từ giao diện Runnable.
   * Ví dụ: Thread t1 = new Thread(r1);
5. Gọi phương thức start() của đối tượng t1.

|  |  |
| --- | --- |
|  |  |

### Khi nào implements từ interface Runnable?

+ Cách hay được sử dụng và được yêu thích là dùng **interface Runnable**, bởi vì nó không yêu cầu phải tạo một lớp kế thừa từ lớp Thread. Trong trường hợp ứng dụng thiết kế yêu cầu sử dụng đa kế thừa, chỉ có interface mới có thể giúp giải quyết vấn đề. Ngoài ra, **Thread Pool** rất hiểu quả và có thể được cài đặt, sử dụng rất hơn giản.  
+ Trong trường hợp còn lại ta có thể kế thừa từ lớp Thread.

### Ví dụ minh họa sử dụng đa luồng

#### Ví dụ Tạo luồng bằng cách extend từ class Thread

Tạo luồng extend từ class Thead

|  |  |
| --- | --- |
| package com.gpcoder.flow;    public class ThreadDemo extends Thread {      private Thread t;      private String threadName;        ThreadDemo(String name) {          threadName = name;          System.out.println(&quot;Creating &quot; + threadName);      }        @Override      public void run() {          System.out.println(&quot;Running &quot; + threadName);          try {              for (int i = 4; i &gt; 0; i--) {                  System.out.println(&quot;Thread: &quot; + threadName + &quot;, &quot; + i);                  // Let the thread sleep for a while.                  Thread.sleep(50);              }          } catch (InterruptedException e) {              System.out.println(&quot;Thread &quot; + threadName + &quot; interrupted.&quot;);          }          System.out.println(&quot;Thread &quot; + threadName + &quot; exiting.&quot;);      }        public void start() {          System.out.println(&quot;Starting &quot; + threadName);          if (t == null) {              t = new Thread(this, threadName);              t.start();          }      }    } |  |

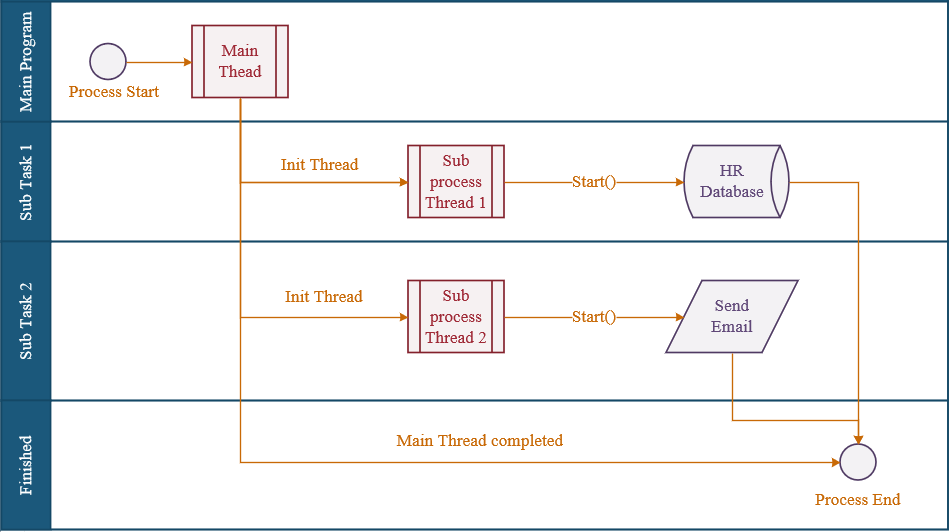
Chương trình sử dụng đa luồng:

|  |  |
| --- | --- |
| package com.gpcoder.flow;    public class ThreadDemoTest {      public static void main(String args[]) {          System.out.println(&quot;Main thread running... &quot;);            ThreadDemo T1 = new ThreadDemo(&quot;Thread-1-HR-Database&quot;);          T1.start();            ThreadDemo T2 = new ThreadDemo(&quot;Thread-2-Send-Email&quot;);          T2.start();            System.out.println(&quot;==&gt; Main thread stopped!!! &quot;);      }  } |  |

Kết quả thực thi chương trình trên:

|  |  |
| --- | --- |
|  |  |

Kết quả chương trình trên được giải thích thông qua hình bên dưới:

[](https://gpcoder.com/wp-content/uploads/2018/02/Multi-Thread-Flow.png)

#### Ví dụ Tạo luồng bằng cách implement từ Interface Runnable

Tạo luồng implement từ Interface Runnable

package com.gpcoder.flow;

class RunnableDemo implements Runnable {

    private Thread t;

    private String threadName;

    RunnableDemo(String name) {

        threadName = name;

        System.out.println(&quot;Creating &quot; + threadName);

    }

    @Override

    public void run() {

        System.out.println(&quot;Running &quot; + threadName);

        try {

            for (int i = 4; i &gt; 0; i--) {

                System.out.println(&quot;Thread: &quot; + threadName + &quot;, &quot; + i);

                // Let the thread sleep for a while.

                Thread.sleep(50);

            }

        } catch (InterruptedException e) {

            System.out.println(&quot;Thread &quot; + threadName + &quot; interrupted.&quot;);

        }

        System.out.println(&quot;Thread &quot; + threadName + &quot; exiting.&quot;);

    }

    public void start() {

        System.out.println(&quot;Starting &quot; + threadName);

        if (t == null) {

            t = new Thread(this, threadName);

            t.start();

        }

    }

}

Chương trình sử dụng đa luồng:

package com.gpcoder.flow;

public class RunnableDemoTest {

    public static void main(String args[]) {

        System.out.println(&quot;Main thread running... &quot;);

        RunnableDemo R1 = new RunnableDemo(&quot;Thread-1-HR-Database&quot;);

        R1.start();

        RunnableDemo R2 = new RunnableDemo(&quot;Thread-2-Send-Email&quot;);

        R2.start();

        System.out.println(&quot;==&gt; Main thread stopped!!! &quot;);

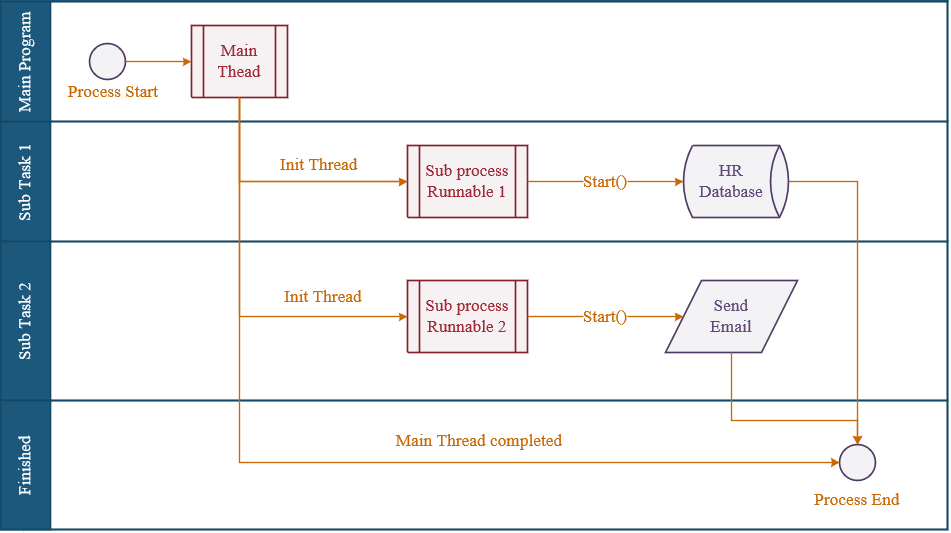
    }

}

Kết quả thực thi chương trình trên:

|  |  |
| --- | --- |
|  |  |

Kết quả chương trình trên được giải thích thông qua hình bên dưới:

[](https://gpcoder.com/wp-content/uploads/2018/02/Multi-Thread-Flow-Runnable.png)

## Các phương thức của lớp Thread thường hay sử dụng

* **suspend()** : Đây là phương thức làm tạm dừng hoạt động của 1 luồng nào đó bằng các ngưng cung cấp CPU cho luồng này. Để cung cấp lại CPU cho luồng ta sử dụng phương thức resume(). Cần lưu ý 1 điều là ta không thể dừng ngay hoạt động của luồng bằng phương thức này. Phương thức suspend() không dừng ngay tức thì hoạt động của luồng mà sau khi luồng này trả CPU về cho hệ điều hành thì không cấp CPU cho luồng nữa.
* **resume()** : Đây là phương thức làm cho luồng chạy lại khi luồng bị dừng do phương thức suspend() bên trên. Phương thức này sẽ đưa luồng vào lại lịch điều phối CPU để luồng được cấp CPU chạy lại bình thường.
* **stop()** : Luồng này sẽ kết thúc phương thức run() bằng cách ném ra 1 ngoại lệ ThreadDeath, điều này cũng sẽ làm luồng kết thúc 1 cách ép buộc. Nếu giả sử, trước khi gọi stop() mà luồng đang nắm giữa 1 đối tượng nào đó hoặc 1 tài nguyên nào đó mà luồng khác đang chờ thì có thể dẫn tới việc sảy ra deadlock.
* **destroy()** : dừng hẳn luồng.
* **isAlive()** : Phương thức này kiểm tra xem luồng còn active hay không. Phương thức sẽ trả về true nếu luồng đã được start() và chưa rơi vào trạng thái dead. Nếu phương thức trả về false thì luồng đang ở trạng thái “New Thread” hoặc là đang ở trạng thái “Dead”
* **yeild()** : Hệ điều hành đa nhiệm sẽ phân phối CPU cho các tiến trình, các luồng theo vòng xoay. Mỗi luồng sẽ được cấp CPU trong 1 khoảng thời gian nhất định, sau đó trả lại CPU cho hệ điều hành (HĐH), HĐH sẽ cấp CPU cho luồng khác. Các luồng sẽ nằm chờ trong hàng đợi Ready để nhận CPU theo thứ tự. Java có cung cấp cho chúng ta 1 phương thức khá đặc biệt là **yeild()**, khi gọi phương thức này luồng sẽ bị ngừng cấp CPU và nhường cho luồng tiếp theo trong hàng chờ Ready. Luồng không phải ngưng cấp CPU như suspend mà chỉ ngưng cấp trong lần nhận CPU đó mà thôi.
* **sleep(long)** : tạm dừng luồng trong một khoảng thời gian millisecond.
* **join()**: thông báo rằng hãy chờ thread này hoàn thành rồi thread cha mới được tiếp tục chạy.
* **join(long)** : Thread cha cần phải đợi millisecond mới được tiếp tục chạy, kể từ lúc gọi join(long). Nếu tham số millis = 0 nghĩa là đợi cho tới khi luồng này kết thúc.
* **getName()** : Trả về tên của thread.
* **setName(String name)** : Thay đổi tên của thread.
* **getId()** : Trả về id của thread.
* **getState():**trả về trạng thái của thread.
* **currentThread()** : Trả về tham chiếu của thread đang được thi hành.
* **getPriority()** : Trả về mức độ ưu tiên của thread.
* **setPriority(int)** : Thay đổi mức độ ưu tiên của thread.
* **isDaemon()** : Kiểm tra nếu thread là một luồng Daemon.
* **setDaemon(boolean)**: xác định thread là một luồng Daemon hay không.
* **interrupt()** : làm gián đoạn một luồng trong java. Nếu thread nằm trong trạng thái sleep hoặc wait, nghĩa là sleep() hoặc wait() được gọi ra. Việc gọi phương thức interrupt() trên thread đó sẽ phá vỡ trạng thái sleep hoặc wait và ném ra ngoại lệ InterruptedException. Nếu thread không ở trong trạng thái sleep hoặc wait, việc gọi phương thức interrupt() thực hiện hành vi bình thường và không làm gián đoạn thread nhưng đặt cờ interrupt thành true.
* **isInterrupted()** : kiểm tra nếu thread đã bị ngắt.
* **interrupted()** : kiểm tra nếu thread hiện tại đã bị ngắt.

## Một số thông tin liên quan đến luồng

### Định danh của luồng (ThreadId)

ThreadId là định danh của luồng, nó dùng để phân biệt với các luồng khác cùng tiến trình hoặc cùng tập luồng. Đây là thông số mà máy ảo java tự tạo ra khi ta tạo luồng nên ta không thể sửa đổi cũng như áp đặt thông số này khi tạo luồng. Nhưng ta có thể lấy được nó thông qua phương thức getId() của lớp Thread

### Tên của luồng (ThreadName)

ThreadName là tên của luồng, đây là thuộc tính mà ta có thể đặt hoặc không đặt cho luồng. Nếu ta không đặt cho luồng thì máy ảo java sẽ tự đặt với quy tắc sau: “Thread-” + Thứ tự luồng được tạo ra, bắt đầu từ 0.

### Độ ưu tiên của luồng (Priority)

Như đã nói ở phần trước, mỗi luồng có 1 độ ưu tiên nhất định. Đây sẽ là thông số quyết định mức ưu tiên khi cấp phát CPU cho các luồng.

Trong java, đế đặt độ ưu tiên cho 1 luồng ta dùng phương thức: **void setPriority(int newPriority)**

* int newPriority : Là giá trị từ 1 đến 10.

Java có định nghĩa sẵn 3 mức ưu tiên chuẩn như sau:

* Thread.**MIN\_PRIORITY** (giá trị **01**)
* hread.**NORM\_PRIORITY** (giá trị **05**)
* Thread.**MAX\_PRIORITY** (giá trị **10**)

Để lấy độ ưu tiên của 1 luồng, ta dùng phương thức: **int getPriority()**

### Ví dụ minh họa

WorkingThread.java

package com.gpcoder.info;

public class WorkingThread extends Thread {

    public WorkingThread(String name) {

        super(name);

    }

    public void run() {

        for (int i = 0; i &amp;lt; 5; i++) {

            System.out.printf(&quot;Luồng: %s có độ ưu tiên là %d \n&quot;, getName(), getPriority());

        }

    }

}

ThreadInfoExample.java

|  |  |
| --- | --- |
|  | package com.gpcoder.info;    public class ThreadInfoExample {        public static void main(String[] args) {          Thread t1 = new WorkingThread(&quot;Luồng 1&quot;);          Thread t2 = new WorkingThread(&quot;Luồng 2&quot;);          Thread t3 = new WorkingThread(&quot;Luồng 3&quot;);            System.out.println(&quot;ID luồng 1: &quot; + t1.getId());          System.out.println(&quot;ID luồng 2: &quot; + t2.getId());          System.out.println(&quot;ID luồng 3: &quot; + t3.getId());            t1.setPriority(1);          t2.setPriority(5);          t3.setPriority(10);            t1.start();          t2.start();          t3.start();      }    } |

Kết quả thực thi chương trình trên:

|  |  |
| --- | --- |
|  |  |

## Sử dụng phương thức sleep()

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

Ví dụ chương trình in ra số từ 1 – 5, tạm ngừng 500 ms trước khi in chữ số tiếp theo.

package com.gpcoder.sleep;

public class SleepMethodExample extends Thread {

    public void run() {

        for (int i = 1; i &lt;= 5; i++) {

            System.out.println(i);

            try {

                Thread.sleep(500);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

    public static void main(String args[]) {

        SleepMethodExample t1 = new SleepMethodExample();

        t1.start();

    }

}

## Sử dụng join() và join(long)

**join()** : thông báo rằng hãy chờ thread này hoàn thành rồi thread cha mới được tiếp tục chạy.

Ví dụ:

package com.gpcoder.join;

public class UsingJoinMethod extends Thread {

    public UsingJoinMethod(String name) {

        super(name);

    }

    @Override

    public void run() {

        System.out.println(getName());

        for (int i = 1; i &lt;= 5; i++) {

            try {

                System.out.print(i + &quot; &quot;);

                Thread.sleep(300);

            } catch (InterruptedException ie) {

                System.out.println(ie.toString());

            }

        }

        System.out.println();

    }

    public static void main(String[] args) throws InterruptedException {

        UsingJoinMethod t1 = new UsingJoinMethod(&quot;Thread 1&quot;);

        UsingJoinMethod t2 = new UsingJoinMethod(&quot;Thread 2&quot;);

        t1.start();

        t1.join();

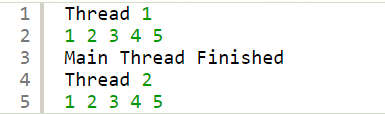
        t2.start();

        System.out.println(&quot;Main Thread Finished&quot;);

    }

}

Thực thi chương trình trên:



**join(long)** : Thread cha cần phải đợi millisecond mới được tiếp tục chạy, kể từ lúc gọi join(long). Nếu tham số millis = 0 nghĩa là đợi cho tới khi luồng này kết thúc.

package com.gpcoder.join;

public class UsingJoinMethod2 extends Thread {

public UsingJoinMethod2(String name) {

super(name);

}

@Override

public void run() {

System.out.println(getName());

for (int i = 1; i &lt;= 5; i++) {

try {

System.out.print(i + &quot; &quot;);

Thread.sleep(300);

} catch (InterruptedException ie) {

System.out.println(ie.toString());

}

}

System.out.println();

}

public static void main(String[] args) throws InterruptedException {

UsingJoinMethod2 t1 = new UsingJoinMethod2(&quot;Thread 1&quot;);

UsingJoinMethod2 t2 = new UsingJoinMethod2(&quot;Thread 2&quot;);

t1.start();

// Main Thread phải chờ 450ms mới được tiếp tục chạy.

// Không nhất thiết phải chờ Thread t1 kết thúc

t1.join(450);

t2.start();

System.out.println(&quot;Main Thread Finished&quot;);

}

}

Thực thi chương trình trên:

|  |  |
| --- | --- |
|  |  |

## Xử lý ngoại lệ cho Thread

Phương thức Thread.setDefaultUncaughtExceptionHandler() thiết lập mặc định xử lý khi luồng đột ngột chấm dứt do một ngoại lệ xảy ra mà không có xử lý khác đã được xác định cho luồng đó.

Ví dụ:

package com.gpcoder.exception;

import java.util.Random;

public class WorkingThread implements Runnable {

    @Override

    public void run() {

        while (true) {

            processSomething();

        }

    }

    private void processSomething() {

        try {

            System.out.println(&quot;Processing working thread&quot;);

            Thread.sleep(500);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

        Random r = new Random();

        int i = r.nextInt(100);

        if (i &gt; 70) {

            throw new RuntimeException(&quot;Simulate an exception was not handled in the thread&quot;);

        }

    }

}

Chương trình minh họa xử lý Thread Exception

package com.gpcoder.exception;

public class ThreadExceptionDemo {

    public static void main(String[] args) {

        System.out.println(&quot;==&gt; Main thread running...&quot;);

        Thread thread = new Thread(new WorkingThread());

        Thread.setDefaultUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {

            @Override

            public void uncaughtException(Thread t, Throwable e) {

                System.out.println(&quot;#Thread: &quot; + t);

                System.out.println(&quot;#Thread exception message: &quot; + e.getMessage());

            }

        });

        thread.start();

        System.out.println(&quot;==&gt; Main thread end!!!&quot;);

    }

}

Thực thi chương trình trên:

|  |  |
| --- | --- |
|  |  |