BÀI 10. LUỒNG DỮ LIỆU

NỘI DUNG

- Khái niệm về luồng dữ liệu
- Đồng bộ hoá đa luồng trong Java.

Luồng (Thread) và đa luồng (Multi Thread)

❖Một luồng (thread) là gì?

- **Một dòng các lệnh mà CPU phải thực thi.**

❖Các hệ điều hành mới cho phép nhiều luồng được thực thi đồng thời.

Chúng ta đã quen với việc mở nhiều ứng dụng trong 1 lần làm việc với máy tính → Nhiều ứng dụng được nạp.

❖**Như vậy**

- Một luồng là một chuỗi các lệnh nằm trong bộ nhớ
- 1 application thông thường khi thực thi là 1 luồng.
- Trong 1 application có thể có nhiều luồng. Thí dụ chuyển động của 10 đối tượng hiện hành trong 1 trò chơi là 10 luồng.

Kỹ thuật đa luồng

- Với máy có m CPU chạy m luồng→Mỗi CPU chạy 1 luồng→Hiệu quả.
- Với máy có m CPU chạy n luồng với n> m→ Mỗi CPU chạy n/m luồng.
- Với 1 CPU chạy đồng thời k luồng với k>1. Các luồng được quản lý bằng 1 hàng đợi, mỗi luồng được cấp phát thời gian mà CPU thực thi là t_i (cơ chế time-slicing – phân chia tài nguyên thời gian). Luồng ở đỉnh hàng đợi được lấy ra để thực thi trước, sau t_i thời gian của mình, luồng này được đưa vào cuối hàng đợi và CPU lấy ra luồng kế tiếp.
- Với máy chỉ có 1 CPU mà lại chạy k luồng → Hiệu suất mỗi chương trình sẽ kém.

Lợi ích của đa luồng

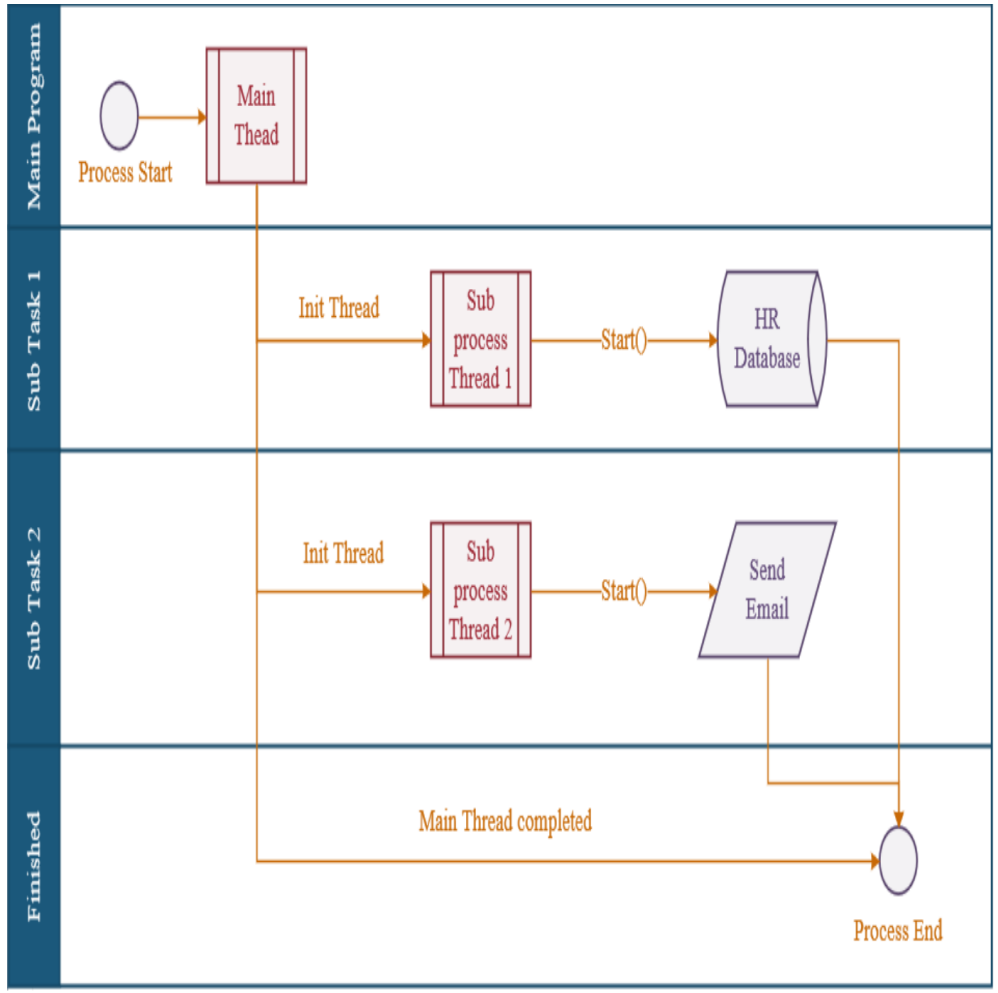
❖**Tăng hiệu suất sử dụng CPU**

❖**Tạo được sự đồng bộ giữa các đối tượng**

➔ Ứng dụng để quản lý được thời gian trong các ứng dụng như thi online, thời gian chơi một trò chơi.

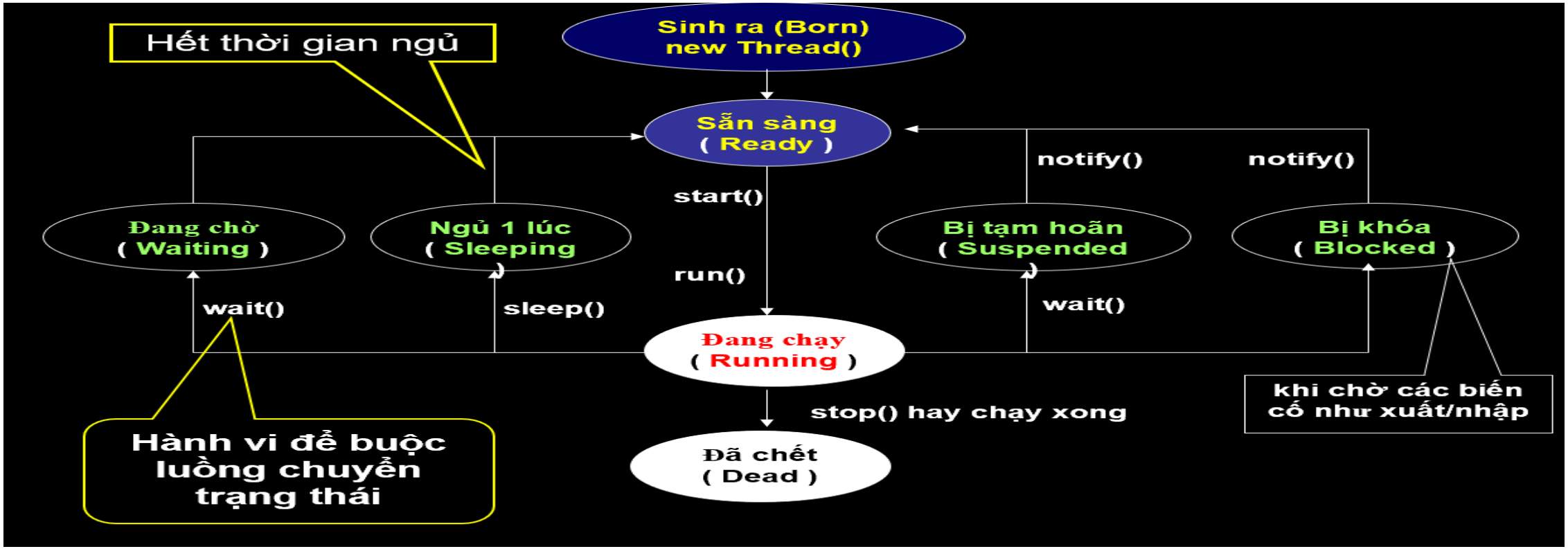
Luồng trong Java

- **Main thread - luồng chính**: là luồng chứa các luồng khác. Đây chính là luồng cho Java. Application hiện hành (mức toàn application).
- **Child thread - luồng con**: là luồng được tạo ra từ luồng khác.
- Khi 1 application thực thi, main thread được chạy, khi gặp các phát biểu phát sinh luồng con, các luồng con được khởi tạo. Vào thời điểm luồng chính kết thúc, application kết thúc.
- Luồng có thể được tạo ra bằng 2 cách:
  - Tạo lớp dẫn xuất từ lớp Thread
  - Tạo lớp hiện thực giao tiếp Runnable.
- Java cung cấp lớp Thread trong gói java.lang



Vòng đời của một luồng (Thread)

❖Một luồng có các trạng thái khác nhau và chuyển qua từng trạng thái trong các giai đoạn khác nhau như được mô tả như hình:



Vòng đời của một luồng

- Một luồng sau khi sinh ra (born/new) không được chạy ngay mà chỉ là sẵn sàng (ready) chạy. Chỉ khi nào phương thức start() được gọi thì luồng mới thực thi (chạy code phương thức run()).
- Luồng đang thực thi có thể bị tạm ngưng bằng phương thức sleep() một thời khoảng và sẽ lại ready sau khi đáo hạn thời gian. Luồng đang ngủ không sử dụng tài nguyên CPU.
- Khi nhiều luồng cùng được thực thi, nếu có 1 luồng giữ tài nguyên mà không nhả ra sẽ làm cho các luồng khác không dùng được tài nguyên này (đòi tài nguyên). Để tránh tình huống này, Java cung cấp cơ chế Wait-Notify (đợi-nhận biết). Phương thức wait() giúp đưa 1 luồng vào trạng thái chờ.

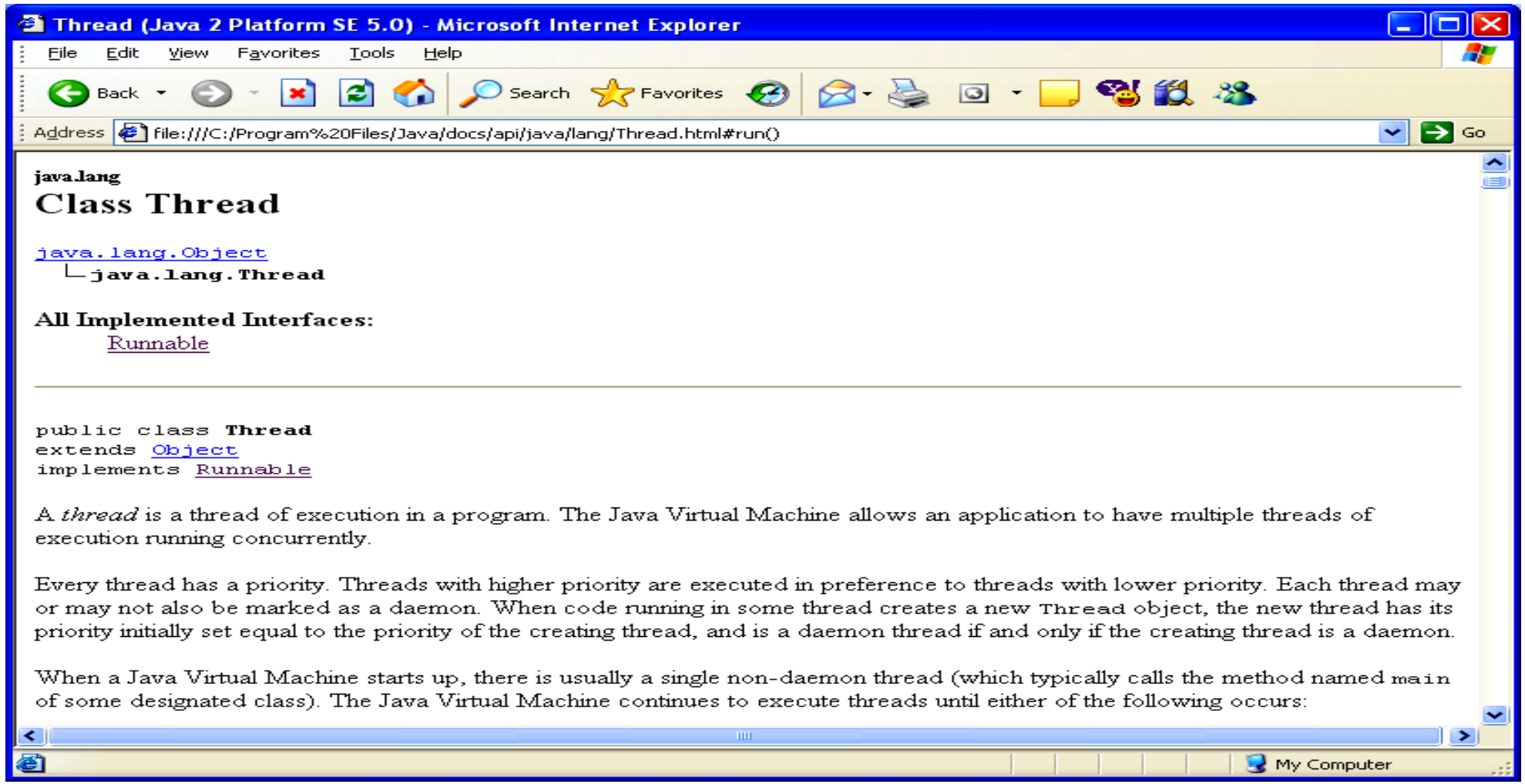
Vòng đời của một luồng

- Khi một luồng bị tạm ngưng hay bị treo, luồng rơi vào trạng thái tạm hoãn (suspended). Phương thức ***suspend()- version cũ/ wait() trong Java 2*** dùng cho mục đích này.
- Khi 1 suspended thread được mang ra thực thi tiếp, trạng thái của luồng là resumed. Phương thức ***resume() – version cũ/ notify() trong Java 2 được dùng cho mục đích này***.
- Luồng rơi vào trạng thái *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.
- Khi 1 luồng thực thi xong phương thức ***run()*** hay gặp phương thức ***stop()***, ta nói luồng đã chết (dead).

Lập trình luồng trong Java

- Cách 1: Xây dựng 1 lớp con của lớp java.lang.**Thread**, override hành vi run() để phù hợp với mục đích bài toán.
- Cách 2: Xây dựng 1 lớp có hiện thực interface **Runnable**
  - Không cần import java.lang vì là gói cơ bản.
  - java.lang.Thread là lớp Java xây dựng sẵn đã hiện thực interface Runnable.
  - Interface java.lang.Runnable chỉ có 1 method run()
  - Tham khảo thêm trong gói java.lang

Tham khảo lớp Thread



Tạo luồng là lớp con của lớp Thread

- Trong Java có sẵn lớp Thread. Để tạo một luồng mới ta có thể tạo một lớp thừa kế (extends) lớp Thread và ghi đè phương thức run()
- Cú pháp:

```
class MyThread extends Thread
{  // dữ liệu + hành vi của lớp
    public void  run()
   { // hiện thực code phụ thuộc bài toán
    }
}
```

Tạo luồng với interface Runnable

- Bạn có thể tạo luồng bằng cách tạo lớp mới hiện thực giao tiếp Runnable và định nghĩa phương thức:
- Cú pháp:

```
class MyThread implements Runnable
{   // dữ liệu + hành vi của lớp
     public void  run()
    { // hiện thực code phụ thuộc bài toán
     }
}
```

Khởi tạo và thực thi 1 luồng

MyThread t = new MyThread(); // tạo 1 luồng
t.start(); // chỉ thị cho luồng thực thi

Hành vi start() sẽ tự động gọi hành vi run()

Hai loại luồng

- **Luồng Daemon**: luồng hệ thống, chạy ở mức nền (background-chạy ngầm), là những luồng cung cấp các dịch vụ cho các luồng khác. Các quá trình trong JVM chỉ tồn tại khi các luồng daemon tồn tại. JVM có ít nhất 1 luồng daemon là luồng “garbage collection”
- **Luồng do user tạo ra**.

Methods thông dụng của lớp Thread

Method	Mục đích
static int enumerate (Thread [] t)	Sao chép các luồng đang tích cực (active) vào 1 mảng từ các nhóm luồng và nhóm con của chúng.
final String getName()	Lấy tên của luồng
final boolean isAlive()	Kiểm tra luồng còn sống hay không?
final void setName(String NewName)	Đặt tên mới cho luồng
final void join () throws interruptedException	nó làm cho các thread đang chạy ngừng hoạt động cho đến khi luồng mà nó tham gia hoàn thành nhiệm vụ của nó.
public final boolean isDaemon()	Kiểm tra xem luồng này có phải là luồng daemon
void setDaemon(boolean on)	on=true : luồng là daemon on=false : luồng của user

Methods thông dụng của lớp Thread

Method	Mục đích
static void sleep (long milisec)	Trì hoãn luồng 1 thời gian
void start()	thực thi luồng
static int activeCount()	Đếm số luồng đang tích cực
static void yield()	Tạm dừng luồng hiện hành để các luồng khác tiếp tục thực thi

Minh họa tạo luồng với lớp Thread

```
// Thread1.java – Minh họa tạo luồng với lớp Thread

class Thread1 extends Thread
{
   public void Create() // tạo luồng con của luồng cha hiện hành
        { Thread t = new Thread (this);
          t.start();
        }
   public void run() // override hành vi run()
         { System.out.println("This is child thread.");
         }
   public static void main (String args[]){
           System.out.println("This is main thread.");
           Thread1 t= new Thread1();
            t.Create(); // tạo luồng con
        }
}
```

Kết qủa ???

Minh họa tạo luồng với lớp interface Runnable

```
class Thread2 implements Runnable
{ public void Create()
  { Thread t = new Thread(this);
    t.start();
  }
  public void run() // implement the run () method
   { System.out.println("This is child thread.");
    }
  public static void main (String args[]){
    System.out.println("This is main thread.");
    Thread2 t= new Thread2();
          t.Create();
  }
}
```

Khi xây dựng luồng bằng interface Runable, phải khai báo 1 đối tượng Thread và gọi hành vi start() để hành vi này gọi run()

Kết qủa ????

MInh họa một số methods của Thread

```
class Thread3 implements Runnable {
   Thread t1,t2;
    Thread3(){
        t1= new Thread(this);
   // t1 is an user-defined thread
         t1.start();
        t2= new Thread(this);
   // t2 is a daemon thread
         t2.setDaemon(true);
   }
    public void run(){
   // Đếm số luồng đang tích cực trong JVM
    int n= Thread.activeCount();
              System.out.println("Number of active
   threads:" + n);
   // lấy tên của 2 luồng
     String t1Name = t1.getName();
     String t2Name = t2.getName();
      System.out.println("Name of t1 thread:" +
     t1Name);
      System.out.println("Name of t2 thread:" +
     t2Name);
      System.out.println("Is t1 thread a daemon? :" +
     t1.isDaemon());
      System.out.println("Is t2 thread a daemon? :" +
     t2.isDaemon());
      System.out.println("Is t1 thread alive? :" +
     t1.isAlive());
      System.out.println("Is t2 thread alive? :" +
     t2.isAlive());
     }
    public static void main (String args[]){
      System.out.println("This is main thread.");
       Thread3 t= new Thread3();
    }
}
```

Kết qủa

Minh họa về trạng thái của luồng

```
class Thread4 extends Thread{
   Thread t;
  Thread4(){
     t= new Thread(this);
    System.out.println("t thread is born and
   ready.");
    t.start();
  }
  public void run(){
   try{
      System.out.println("t thread is running.");
        t.sleep(5000);
        System.out.println("t is awaked and
      running again after 5 secs.");
        }catch( InterruptedException e){
      System.out.println("thread is interrupted!");
        }
      }
      public static void main (String args[])
     {   new Thread4();
     }
  }
```

Độ ưu tiên của luồng

- Các luồng cùng chia sẻ thời gian của CPU → Luồng ở cuối hàng đợi sẽ lâu được CPU thực thi → Có nhu cầu thay đổi độ ưu tiên của luồng. Java cung cấp 3 hằng mô tả độ ưu tiên của 1 luồng (các độ ưu tiên khác dùng 1 số nguyên từ 1.. 10).
  - **NORM_PRIORITY : mang trị 5**
  - **MAX_PRIORITY : mang trị 10**
  - **MIN_PRIORITY : mang trị 1**
- Độ ưu tiên mặc định của 1 luồng là NORMAL_PRIORITY. Luồng con có cùng độ ưu tiên với luồng cha (do đặc điểm thừa kế).

Thao tác với độ ưu tiên của luồng

- **final void setPriority(int newPriority)**
- **final int getPriority()**

Như vậy, các điều kiện để 1 luồng không được thực thi:

- Luồng không có được độ ưu tiên cao nhất để dành lấy thời gian của CPU.
- Luồng bị cưỡng bức ngủ bằng hành vi sleep().
- Luồng bị chờ do hành vi wait().
- Luồng bị tường minh nhận hành vi yield().
- Luồng bị khóa vì đang chờ I/O

Minh họa về độ ưu tiên của luồng

```
class Thread5 extends Thread// Thread5.java
{
  public void run()
   { Thread Child = new Thread(this);
     Child.setName("Child thread");
     System.out.println("Name of current thread:" + Thread.currentThread().getName());
     System.out.println("Piority of current thread:" + Thread.currentThread().getPriority());
     System.out.println("Name of child:" + Child.getName());
     System.out.println("Priority of child:" + Child.getPriority());
  }
  public static void main (String args[])
  { Thread5 t = new Thread5();
    t.start();
    t.setName("Parent thread");
  }
}
```

Bài tập

- Bài 1. Tạo 2 luồng: luồng 1 hiển thị các số chẵn, luồng 2 hiển thị các số lẻ.
- Bài 2. Tạo 2 luồng: luồng 1 hiển thị các số nguyên tố, luồng 2 hiển thị các số hoàn thiện.

Đồng bộ các luồng

❖**Tình huống:** Có hai luồng t1, t2 cùng truy xuất 1 đối tượng dữ liệu là biến m. t1 muốn đọc biến m còn t2 muốn ghi biến m.

→ dữ liệu mà t1 đọc được có thể không nhất quán.

→ Nếu để cho t2 ghi m trước rồi t1 đọc sau thì t1 đọc được dữ liệu nhất quán tại thời điểm đó.

→ Cần có cơ chế để chỉ cho phép 1 luồng được truy xuất dữ liệu chung (shared data) tại 1 thời điểm.

→ Kỹ thuật này gọi là **"ĐỒNG BỘ HÓA – SYNCHRONIZATION"**

Kỹ thuật cơ bản về đồng bộ hóa

- Tạo ra 1 đối tượng quản lý sự đồng bộ của 1 thao tác dữ liệu của các luồng bằng cách thực thi hộ một tác vụ của các luồng mỗi lần chỉ cho 1 luồng bằng từ khóa **synchronized**
- → **Luồng đang được chiếu cố gọi là luồng đang có monitor**

Minh họa về đồng bộ các luồng bằng MONITOR

❖**Chương trình sau sẽ xuất 3 số 10, 11, 12 ra màn hình, mỗi số được 1 luồng thực thi.**

```
// Monitor1.java – Lớp làm nhiệm vụ xuất hộ 1 số num
class Monitor1{
    synchronized void Display (int num){
    System.out.println("Output " + num + " - done.");
        try{
         Thread.sleep(500); // current thread sleep 1/2 sec
        }
        catch (InterruptedException e){
         System.out.println ("Thread is interrupted!");
        }
     }
```

Từ khóa synchronized khai báo có quản lý việc đồng bộ các luồng

Minh họa về đồng bộ các luồng bằng MONITOR

```
class OutNum implements Runnable // luồng{
     Monitor1 monitor; // Luồng có dữ liệu là monitor
     int number;  // dữ liệu cần xuất
     Thread t;
    // hành vi xuất n với Monitor1 có tên moni
     OutNum(Monitor1 moni, int n ){
        monitor= moni;
        number = n;
        t = new Thread(this);
        t.start();
     }
   // khi luồng chạy, số number được xuất bởi monitor
     public void run() {
            monitor.Display(number);
     }
}
```

Minh họa về đồng bộ các luồng bằng MONITOR

```
class Synchro // lớp của chương trình chính
{
   public static void main (String args[])
   {  Monitor1 monitor = new Monitor1();
      int num = 10;
      OutNum Obj1 = new OutNum(monitor,num++);
      OutNum Obj2 = new OutNum(monitor,num++);
      OutNum Obj3 = new OutNum(monitor,num++);
      // wait for 3 threads to end
      try
      { Obj1.t.join();
        Obj2.t.join();
        Obj3.t.join();
      }
      catch(InterruptedException e)
      { System.out.println ("Thread was interrupted!");  }
    }
}
```

3 luồng có 3 trị khác nhau là 10,11, 12 nhưng có chung 1 monitor
Ba luồng cùng đơ

```
Output 10 - done.
Output 11 - done.
Output 12 - done.
```

Kỹ thuật đồng bộ luồng theo khối

- Đồng bộ một khối tác vụ.
- Người lập trình có thể không muốn dùng các synchronized method để đồng bộ truy xuất đến đối tượng.
- Các lớp được cung cấp bởi các thư viện (lớp đã xây dựng) - nên không thể thêm từ khóa **synchonized** vào được các method này.

❖Cú pháp đồng bộ khối

```
synchronized (Object){
<các lời gọi methods của Object cần phải được đồng bộ>
}
```

❖**Buộc phải có { } dù chỉ có 1 phát biểu**

Minh họa đồng bộ khối

❖**Chương trình sau viết lại chương trình trước, bỏ qua từ khóa synchronized trong lớp Monitor1 (ở đây gọi là lớp Monitor2)**

```
class Monitor2  // Monitor2.java
{
  void Display (int num){
        System.out.println("Output " + num + " - done.");
    try{
        Thread.sleep(500); // current thread sleap 1/2 sec
    }catch (InterruptedException e){
        System.out.println ("Thread is interrupted!");
    }
  }
}
```

Minh họa đồng bộ khối

```
class OutNum implements Runnable{
   Monitor2 monitor;
   int number;
   Thread t;
  OutNum(Monitor2 moni, int n ){
     monitor= moni;
    number = n;
    t = new Thread(this);
    t.start();
  }
  public void run(){
          synchronized (monitor) {
                  monitor.Display(number);
      }
  }
}
```

Minh họa đồng bộ khối

```
class Synchro
{  public static void main (String args[])
   {  Monitor2 monitor = new Monitor2();
      int num = 10;
      OutNum Obj1 = new OutNum(monitor,num++);
      OutNum Obj2 = new OutNum(monitor,num++);
      OutNum Obj3 = new OutNum(monitor,num++);
      // wait for 3 threads to end
      try
      { Obj1.t.join();
        Obj2.t.join();
        Obj3.t.join();
      }
      catch(InterruptedException e)
      { System.out.println ("Thread was interrupted!");
      }
   }
}
```

Deadlock

- Deadlock – tình huống bế tắc, đóng băng- xẩy ra khi các luồng chờ tài nguyên (monitor) của nhau hình thành một chu trình. Deadlock hiếm khi xẩy ra.
- Minh họa: DeadlockDemo.java

Ví dụ demo Deadlock DeadlockDemo class

```
public class DeadlockDemo {
public static void main(String[] args) {
final String value1 = "Lock1";
final String value2 = "Lock2";
Thread t1 = new Thread() {
public void run() {
synchronized(value1) {
System.out.println("Thread 1 locked value1");
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("Thread 1 waiting for value2
lock");
synchronized(value2) {
System.out.println("Thread 1 locked value2");
}}}};
Thread t2 = new Thread() {
public void run() {
synchronized(value2) {
System.out.println("Thread 2 locked value2");
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("Thread 2 waiting for value1
lock");
synchronized(value1) {
System.out.println("Thread 2 locked value1");
}}}};
t1.start();
t2.start();
}
}
```

Giải thích DeadlockDemo class

- 1 ứng dụng có 2 luồng: Luồng t1 trong đối tượng d1, luồng t2 trong đối tượng d2
- Monitor của t1 lại là d2 và monitor của t2 lại là d1 (chéo nhau).
- Cả 2 luồng cùng gọi hành vi **synchronized run()** và cùng ngủ 300 mili giây. Vì chia sẻ thời gian CPU nên t1 ngủ trước và t2 ngủ sau (xem phương thức run()).
- Khi t1 thức dậy (wake up), phương thức Synchro() của đối tượng monitor của d2 (chứa luồng t2) được gọi nhưng luồng t2 đang ngủ nên phương thức này chưa thể thực thi.
- Khi t2 thức dậy (wake up), phương thức Synchro() của đối tượng monitor của d1 (chứa luồng t1) được gọi nhưng luồng t1 cũng đang ngủ nên phương thức này chưa thể thực thi.
- Như vậy chương trình sẽ đóng băng (blocked) không làm gì được nữa.

CHỈNH CODE KHÔNG XẢ RA Deadlock

```
public class DeadlockDemo {
public static void main(String[] args) {
final String value1 = "Lock1";
final String value2 = "Lock2";
Thread t1 = new Thread() {
public void run() {
synchronized(value1) {
System.out.println("Thread 1 locked value1");
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("Thread 1 waiting for value2
lock");
synchronized(value2) {
System.out.println("Thread 1 locked value2");
}}}};
Thread t2 = new Thread() {
public void run() {
synchronized(value1) {
System.out.println("Thread 2 locked value1");
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("Thread 2 waiting for value2
lock");
synchronized(value2) {
System.out.println("Thread 2 locked value2");
}}}};
t1.start();
t2.start();
}
}
```

Cơ chế chờ - nhận biết

- Java cung cấp sẵn một cơ chế giao tiếp liên quá trình (inter-process mechanism) để các luồng có thể gọi nhau (**yêu cầu nhau**) sử dụng các final methods của lớp Object: *wait() , notify() , notifyAll()*. Như vậy mọi lớp đều có thể sử dụng chúng và các phương thức này *chỉ có thể được gọi trong các synchronized methods*.
- **Phương thức wait()** : Luồng nhả monitor để đi vào trạng thái ***sleep*** cho đến khi 1 luồng khác vào cùng monitor và gọi phương thức ***notify***.
- **Phương thức notify()** : Luồng thức dậy (wake up) và nhận biết (notify) rằng luồng thứ nhất đã gọi wait().
- **Phương thức notifyAll()** : Đánh thức tất cả các luồng đang ngủ để chúng biết rằng luồng hiện hành đã gọi phương thức wait(). Khi tất cả các luồng đang ngủ thức dậy, luồng có ưu tiên cao nhất sẽ nắm giữ monitor và thực thi.

Chú ý đối với phương thức wait()

- Luồng gọi phương thức wait() sẽ nhả CPU, thôi không dùng CPU nữa.
- Luồng gọi phương thức wait() sẽ nhả monitor, thôi không khóa (lock) monitor nữa.
- Luồng gọi phương thức wait() sẽ được đưa vào danh sách hàng đợi monitor (monitor waiting pool)

Chú ý đối với phương thức notify()

- Một luồng đang ngủ được đưa ra khỏi monitor waiting pool và đi vào trạng thái ***ready***.
- Luồng vừa thức giấc (notify) phải giành lại monitor và khóa monitor lại không cho luồng khác chiếm để luồng này được thực thi.

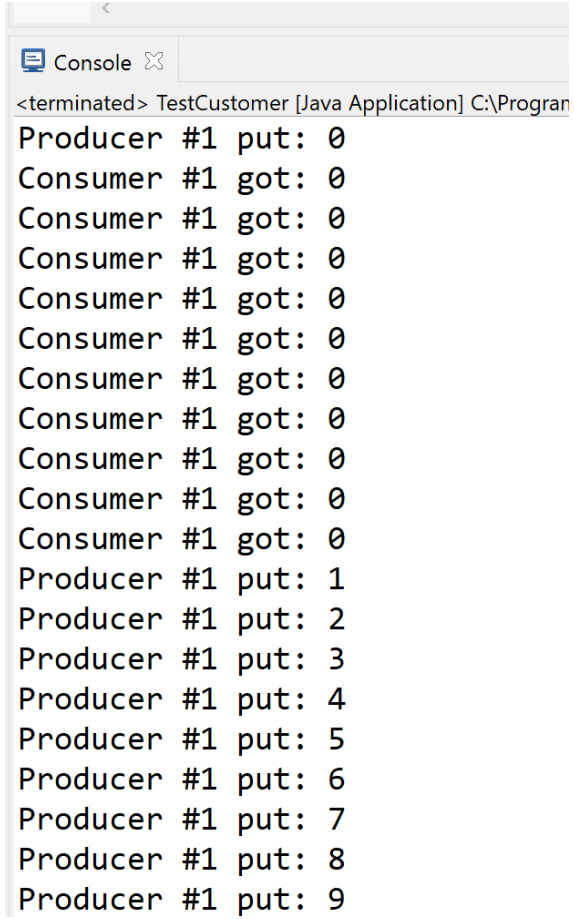
Chú ý đối với phương thức notifyAll()

- Luồng đang thực thi cảnh báo cho tất cả các luồng đang ngủ rằng "Tôi đi ngủ đây, các bạn dậy để làm việc".
- Luồng ở đầu danh sách monitor waiting pool được vào trạng thái ready

Áp dụng đồng bộ luồng trong java với bài toán: Bài toán Producer/Consumer

- Có hai luồng, một sản xuất và một tiêu thụ cả hai truy cập cùng môt đối tượng CubbyHole.
- CubbyHole là một đối tượng đơn giản lưu giữ một giá trị đơn như nội dung của nó.
- Luồng sản xuất phát sinh ngẫu nhiên các giá trị và cất giữ chúng trong đối tượng CubbyHole
- Luồng tiêu thụ lấy các giá trị này khi chúng được sinh ra bởi luồng sản xuất
- Vấn đề là đảm bảo rằng nhà sản xuất không thể thêm dữ liệu vào bộ đệm nếu nó đầy và người tiêu dùng không thể xóa dữ liệu khỏi bộ đệm trống, đồng thời đảm bảo an toàn cho luồng (thread-safe).

Minh họa vấn đề xảy ra của bài toán Producer/Consumer



Giải pháp để giải quyết vấn đề Producer/ Consumer

❖Ý tưởng:

- Giải pháp **cho nhà sản xuất là đi ngủ (wait)** nếu bộ đệm đầy. Lần tiếp theo người tiêu dùng xóa một mục khỏi bộ đệm, nó đánh thức (notify) nhà sản xuất, bắt đầu đưa dữ liệu vào bộ đệm.
- Theo cách tương tự, **người tiêu dùng có thể đi ngủ (wait) nếu thấy bộ đệm trống.** Lần tiếp theo nhà sản xuất đưa dữ liệu vào bộ đệm, nó đánh thức (notify) người tiêu dùng đang ngủ (wait).
- Trong khi làm tất cả điều này, phải đảm bảo an toàn cho luồng (thread safe).

Giải pháp để giải quyết vấn đề Producer/ Consumer

❖Kết quả:

Console

<terminated> TestCustomer [Java Application] C:\Program

```
Producer #1 put: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Producer #1 put: 1
Producer #1 put: 2
Producer #1 put: 3
Producer #1 put: 4
Producer #1 put: 5
Producer #1 put: 6
Producer #1 put: 7
Producer #1 put: 8
Producer #1 put: 9
```

synchronized

Console

<terminated> SynTestCustomer [Java Application] C:\Prograı

```
Producer syn #1 put: 0
Consumer Syn #1 got: 0
Producer syn #1 put: 1
Consumer Syn #1 got: 1
Consumer Syn #1 got: 2
Producer syn #1 put: 2
Consumer Syn #1 got: 3
Producer syn #1 put: 3
Producer syn #1 put: 4
Consumer Syn #1 got: 4
Producer syn #1 put: 5
Consumer Syn #1 got: 5
Producer syn #1 put: 6
Consumer Syn #1 got: 6
Producer syn #1 put: 7
Consumer Syn #1 got: 7
Producer syn #1 put: 8
Consumer Syn #1 got: 8
Producer syn #1 put: 9
Consumer Syn #1 got: 9
```

Hiện thực bài toán Producer/Consumer

```
public class SynCubbyHole {
private int contents=0;
private boolean available = false;
public synchronized int get() {
      while (available == false) {
         try {
            wait();
         }
       catch (InterruptedException e) {
         }
      }
       notifyAll();
       available = false;
      return contents;
   }
```

```
public synchronized void put(int value) {
      while (available == true) {
         try {
            wait();
         }
         catch (InterruptedException e) {
         }
      }
      contents = value;
       notifyAll();
       available = true;
   }
}
```

Hiện thực bài toán Producer/Consumer

```
public class SynCustomer extends Thread{
private SynCubbyHole synCubbyHole;
private int who;
public SynCustomer(SynCubbyHole
synCubbyHole, int who) {
super();
this.synCubbyHole = synCubbyHole;
this.who = who;
}
public void run() {
int value=0;
for (int i = 0; i < 10; i++) {
value=synCubbyHole.get();
System.out.println("Consumer Syn #"
+ this.who
+ " got: " + value);
}
}

}
```

Hiện thực bài toán Producer/Consumer

```
public class SynProducer extends Thread{
private SynCubbyHole synCubbyHole;
private int who;
public SynProducer(SynCubbyHole
synProducer, int who) {
super();
this.synCubbyHole = synProducer;
this.who = who;
}
public void run() {
for (int i = 0; i < 10; i++) {
synCubbyHole.put(i);
System.out.println("Producer syn #" +
this.who
+ " put: " + i);
try {
sleep((int) (Math.random()*100));
}catch(InterruptedException ex) {

}
}
}

}
```

Hiện thực bài toán Producer/Consumer

```
public class SynTestCustomer {
public static void main(String[] args) throws InterruptedException {
SynCubbyHole cub = new SynCubbyHole();
SynProducer pro= new SynProducer(cub, 1);
pro.start();
SynCustomer cus= new SynCustomer(cub, 1);
cus.start();
}

}
```

TÓM TẮT

- Luồng là biện pháp chia công việc thành các đơn vị cụ thể (concrete) nên có thể được dùng để thay thế vòng lặp.
- Lập trình đa luồng làm tăng hiệu suất CPU trên những hệ thống "bận rộn". Tuy nhiên hiệu suất của từng ứng dụng lại bị giảm đang kể (chậm ba bốn lần do các tác vụ đồng bộ hóa), quá trình biên dịch cũng chậm vì trình biên dịch phải tính toán cơ chế quản lý các luồng. Do vậy trong các ứng dụng đòi hỏi yếu tố hiệu suất thời gian là quan trọng, nên tránh sử dụng kỹ thuật đồng bộ hóa. → Nhiều lập trình viên không thích lập trình đa luồng mà chỉ dùng lập trình lập trình đơn luồng để tăng hiệu suất của ứng dụng.
- Java cung cấp kỹ thuật lập trình đa luồng bằng lớp **Thread** và interface **Runnable**.
- Khi 1 ứng dụng Java thực thi, có 1 luồng đang chạy đó là luồng chính (main thread). Luồng chính rất quan trọng vì (1) Đây là luồng có thể sinh ra các luồng con, (2) Quản lý việc kết thúc ứng dụng vì luồng main chỉ kết thúc khi tất cả các luồng con của nó đã kết thúc.

TÓM TẮT

- Hiện thực 1 luồng bằng 1 trong 2 cách:
  - Hiện thực 1 lớp con của lớp **Thread**, override phương thức **run()** của lớp này.
  - Khai báo lớp mà ta xây dựng là implement của interface **Runnable** và định nghĩa phương thức **run()**.
- Mỗi java thread có 1 độ ưu tiên từ 1 (MIN) đến 10 (MAX) với 5 là trị mặc định. JVM không bao giờ thay đổi độ ưu tiên của luồng.
- Có 8 contructor của lớp Thread nhưng 2 constructor thường dùng: **Thread()** và **Thread(String TênLuồng), Thread(ĐốiTượngChứa)**.
- Các phương thức *Thread.suspend(), Thread.resume(), Thread.stop()* không còn được dùng nữa kể từ Java 2.
- Luồng *daemon* là luồng chạy ngầm nhằm cung cấp dịch vụ cho các luồng khác. Nếu muốn 1 luồng là daemon, hãy dùng ***public final void setDeamon (boolean)*** và kiểm tra 1 luồng có là daemon hay không, hãy dùng ***public final boolean isDaemon()***.

TÓM TẮT

- Dữ liệu có thể bị mất nhất quán(hư hỏng) khi có 2 luồng cùng truy xuất dữ liệu tại cùng 1 thời điểm.
- Đồng bộ là 1 quá trình bảo đảm tài nguyên (dữ liệu, file,…) chỉ được 1 luồng sử dụng tại 1 thời điểm. Tuy nhiên, chi phí cho việc này lại làm giảm hiệu suất thời gian của ứng dụng **xuống 3, 4 lần**.
- Phương thức ***wait()*** sẽ làm 1 luồng đi vào trạng thái ngủ.
- Phương thức ***notify()*** sẽ đánh thức luồng thứ nhất trong danh sách luồng đang chờ trên cùng 1 đối tượng monitor.
- Phương thức ***notifyAll()*** sẽ đánh thức tất cả các luồng trong danh sách luồng đang chờ trên cùng 1 đối tượng monitor.
- Deadlock xẩy ra khi 2 luồng có sự phụ thuộc vòng trên một cặp đối tượng quản lý việc đồng bộ (synchronized object).