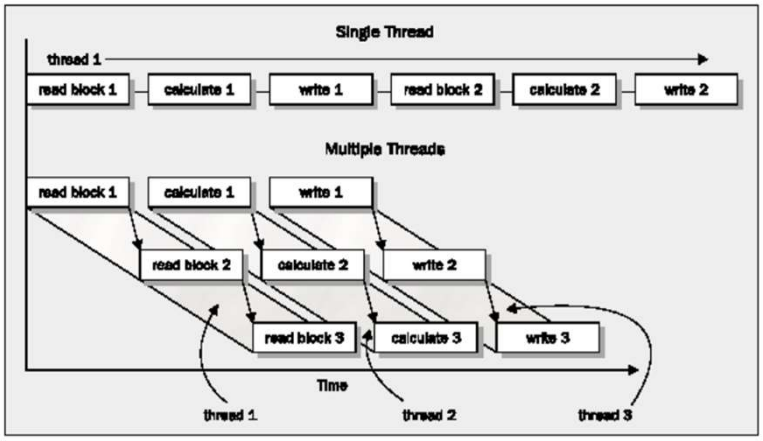
Chương 3.
Lập trình dùng chung (đa luồng)
Concurrent Programming

Nội dung

3.1. Giới thiệu luồng (Thread) và Đa luồng (MultiThread)

- Một luồng là đơn vị nhỏ nhất gồm các mã thi hành nhằm thực hiện các nhiệm vụ riêng biệt.
- Java hỗ trợ đa luồng.
- Một ứng dụng có thể chứa đựng nhiều luồng. Mỗi một luồng được chỉ rõ nhiệm vụ cần thực hiện và có thể thực hiện đồng thời cùng với với các luồng khác. Khả năng này được gọi là đa luồng (**Multithread**).



3.2. Khởi tạo và quản lý các luồng

❑ Khi thực hiện một chương trình Java, có một luồng đã sẵn sàng chạy, đó là ***main***. Luồng main rất quan trọng bởi 2 lý do sau:

- ✓ Nó là luồng để từ đó các luồng con sẽ được khởi tạo.
- ✓ Nó cũng là luồng cuối cùng được kết thúc. Nếu luồng main bị dừng thực hiện thì chương trình sẽ kết thúc.

3.2. Khởi tạo và quản lý các luồng (tiếp)

- Ta có thể khởi tạo các luồng theo 2 cách sau:
  - ✓ Khai báo một lớp là lớp con của lớp Thread mà ở đó cần phải ghi đè phương thức run() của lớp Thread.

```
class MyThread extends Thread {
  public void run( ) {
  //your thread implementation here...
  }
}
```

  - ✓ Khai báo một lớp kế thừa giao diện Runnable. Sau đó triển khai phương thức run( ).

```
class MyThread implements Runnable {
  public void run( ) {
  //your thread implementation here...
  }
}
```


3.2. Khởi tạo và quản lý các luồng (tiếp)

- Trong lớp con của Thread, cần triển khai phương thức run(). Phương thức run() là điểm đầu vào hay điểm bắt đầu của một luồng.
- Để bắt đầu một luồng, cần khởi tạo một đối tượng từ lớp Thread. Gọi phương thức start() tới đối tượng luồng. Điều này sẽ khởi tạo một luồng mới, start kích hoạt đầu vào của chương trình, sau đó gọi phương thức run() của đối tượng luồng.

```
class  SimpleThread  extends Thread {
   public void run() {
      for  ( int  count = 0;  count < 4; count++)
         System.out.println( "Message " + count +
                             " From: Mom" );
   }
}
class  TestingSimpleThread {
   public static void main( String[]  args ) {
      SimpleThread  parallel  = new SimpleThread();
      System.out.println( "Create the thread");
      parallel.start();
      System.out.println( "Started the thread" );
      System.out.println( "End" );
   }
}
```

3.2. Khởi tạo và quản lý các luồng (tiếp)

Nếu một class có ý định để được thực thi như một thread, thì ta có thể đạt được điều này bởi triển khai **Runnable** Interface.

Cần theo 3 bước cơ bản sau:

Bước 1: Triển khai một phương thức run() được cung cấp bởi **Runnable** Interface. Phương thức này cung cấp điểm đi vào cho thread.

public void run( )

Bước 2: Khởi tạo một đối tượng **Thread**, sử dụng constructor sau trong Java:

Thread(Runnable threadObj, String threadName);

- *threadObj* là đối tượng của lớp triển khai Runnable Interface
- *threadName* là tên thread mới.

Bước 3: Khi đối tượng Thread được tạo, ta có thể bắt đầu nó bởi gọi phương thức **start()** trong Java, và nó sẽ thực thi một triệu hồi tới phương thức run().

void start( );

3.2. Khởi tạo và quản lý các luồng (tiếp)

```
class SecondMethod implements Runnable {
   public void run() {
      for ( int count = 0; count < 4; count++)
         System.out.println( "Message " + count + " From: Dad");
   }
}
class TestThread {
   public static void main( String[] args ) {
      SecondMethod notAThread = new SecondMethod();
      Thread parallel = new Thread( notAThread );
      System.out.println( "Create the thread");
      parallel.start();
      System.out.println( "Started the thread" );
      System.out.println( "End" );
   }
}
```

Tạo luồng bằng cách kế thừa từ lớp Thread

Để tạo luồng bằng cách tạo lớp kế thừa từ lớp Thread, ta thực hiện tuần tự các công việc sau :

- Khai báo 1 lớp mới kế thừa từ lớp Thread
- Ghi đè phương thức run(), 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.
- Tạo 1 đối tượng thuộc lớp vừa khai báo.
- Gọi phương thức start() của đối tượng được tạo để bắt đầu thực thi luồng.

Tạo luồng bằng cách triển khai từ Runnable

Để tạo luồng bằng cách triển khai từ Interface Runnable, ta phải thực hiện tuần tự các công việc sau :

- Khai báo 1 lớp mới kế thừa giao diện Runnable
- Triển khai phương thức run(), 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.
- Tạo 1 đối tượng thuộc lớp vừa khai báo.
  (*Ví dụ* : Tên đối tượng là r1)
- Tạo 1 đối tượng của lớp Thread bằng phương thức khởi tạo:

Thread(Runnable target)

Runnable target: Là 1 đối tượng thuộc lớp được implements từ giao diện Runnable.

VD: Thread t1=new Thread(r1);

- Gọi phương thức start() của đối tượng t1

3.2. Khởi tạo và quản lý các luồng (tiếp)

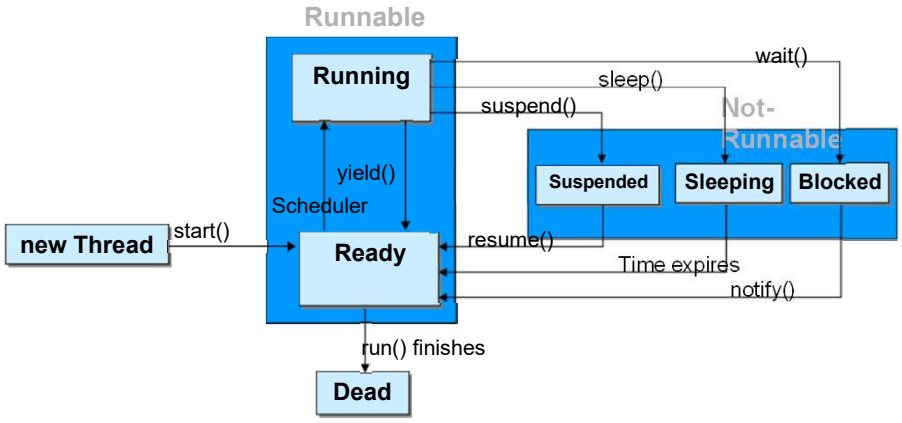
- Có một vài phương thức khởi tạo trong lớp Thread. Hai trong số chúng là:
  - public Thread(String threadname)
    - Khởi tạo một luồng có tên threadname
  - public Thread( )
    - Khởi tạo một luồng có tên "Thread" được ghép với một con số, ví dụ Thread-0, Thread-1…

3.2. Khởi tạo và quản lý các luồng (tiếp)

Ví dụ: Đặt tên cho thread:

```
package com.alliant.corejava.lesson4;
class SecondMethod extends Thread {
    public void run() {
        for (int count = 0; count < 2; count++) {
            System.out.println("Message " + count + " From: " +
                    Thread.currentThread().getName());
        }
    }
}
public class ThreadName {
    public static void main(String[] args) {
        Thread a = new SecondMethod();
        a.setName("Mom");

        Thread b = new SecondMethod();
        b.setName("Dad");

        System.out.println("Create the thread");
        a.start();
        b.start();
        System.out.println("End");
    }
}
```

3.3. Vòng đời của một luồng



3.3. Vòng đời của một luồng (tiếp)

- Một luồng phải được khởi tạo và nó sẽ không chạy lập tức ngay sau khi được khởi tạo. Nó đợi phương thức *start( )* của nó được gọi, và ngay sau đó nó sẵn sàng để chạy phương thức *run()*.
- Ta có thể tạm dừng thực hiện của một luồng trong một khoảng thời gian bởi phương thức *sleep*( ). Luồng sẽ đi vào trạng thái ngủ và quay lại khi thời gian ngủ hết hiệu lực.
- Một luồng sẽ đi vào trạng thái *waiting* khi một luồng đang chạy gọi phương thức *wait.* Và nó sẽ quay lại sau khi *notify* được gửi đến bởi luồng liên đới.
- Một luồng sẽ đi vào trạng thái *blocked* khi nó đang thi hành các toán tử Input/Output và quay lại sau khi I/O được hoàn thành.
- Một luồng sẽ đi vào trạng thái *dead* sau khi thực hiện hoàn thành phương thức run() hoặc phương thức stop được gọi.

3.4. Các phương thức của lớp Thread

Phương thức và Miêu tả
✓ **public void start():** Bắt đầu một Thread, sau đó triệu hồi phương thức run() trên đối tượng Thread này
✓ **public void run():** Nếu đối tượng Thread này được khởi tạo bởi sử dụng một đối tượng Runnable, phương thức run() được triệu hồi trên đối tượng Runnable đó
✓ **public final void setName(String name):** Thay đổi tên của đối tượng Thread. Phương thức **getName()** để lấy tên.
✓ **public final void setPriority(int priority):** Thiết lập quyền ưu tiên của đối tượng Thread này. Giá trị có thể có nằm trong khoảng từ 1 tới 10
✓ **public final void join(long millisec):** Thread hiện tại triệu hồi phương thức này trên thread thứ hai, làm cho Thread hiện tại block tới khi thread thứ hai kết thúc hoặc sau một số lượng mili giây đã xác định
✓ **public final boolean isAlive():** Trả về true nếu thread này là đang sống.

3.4. Các phương thức của lớp Thread

Các phương thức trên được triệu hồi trên một đối tượng Thread cụ thể. Các phương thức sau trong lớp Thread là static. Triệu hồi một trong các phương thức này thực hiện hoạt động trên thread đang chạy hiện tại.

Phương thức và Miêu tả
✓ **public static void yield():** Làm cho thread đang chạy hiện tại chuyển tới bất kỳ thread nào khác. Khi gọi phương thức này luồng sẽ bị ngừng cấp CPU và nhường cho luồng tiếp theo trong hàng chờ.
✓ **public static void sleep(long millisec):** Làm cho thread đang chạy hiện tại bị dừng thời gian mili giây đã xác định
✓ **public static Thread currentThread():** Trả về một tham chiếu tới thread đang chạy hiện tại.

3.4. Các phương thức của lớp Thread

- **Thread.join()**
  - ✓ Nếu luồng A gửi phương thức join() tới luồng B, thì luồng A sẽ không chạy cho đến khi phương thức run của luồng B kết thúc. Lúc đó luồng A mới chạy lại.
  - ✓ Hàm Join được sử dụng để giữ cho quá trình thực thi của hàm đang chạy không bị gián đoạn bởi các thread khác, nói một cách khác nếu một thead đang chạy các thread khác sẽ phải chờ cho đến khi thread đó thực thi xong.
  - ✓ Tạo sao lại sử dụng hàm Join?
    Trong một chương trình Java thường có nhiều hơn một thread, trong đó có main thread - có chức năng khởi tạo và kích hoạt để chạy các thread khác, tuy nhiên các main thread không đảm bảo các thread thực thi và kết thúc theo đúng thứ tự mà chúng đã được khởi chạy.
    Câu hỏi đưa ra là: Làm thế nào để các thread thực thi và kết thúc theo đúng thứ tự mà chúng được khởi chạy. Câu trả lời là: Sử dụng hàm Join sẵn có của Java.

3.4. Các phương thức của lớp Thread

- Trạng thái Sleep
  - Mục đích của làm cho luồng hiện tại tạm dừng thực thi trong một khoảng thời gian.
  - Luồng không bị mất đi.
  - Sau thời gian ngủ, nó sẽ thức dậy và tiếp tục chạy.
  - Sử dụng bằng cách gọi: **sleep**(long millis)

3.5. Các luồng Deamon

- Một chương trình Java bị dừng chỉ sau khi tất cả các luồng kết thúc. Có 2 kiểu luồng trong chương trình Java
  - Các luồng thông thường
  - Các luồng Daemon
- Chúng chỉ khác nhau ở cách thức ngừng hoạt động. Trong một chương trình các luồng thông thường và Deamon chạy song song với nhau. Khi tất cả các luồng thông thường kết thúc, mọi luồng Deamon cũng sẽ bị kết thúc theo bất kể nó đang làm việc gì.
- Lớp thread có 2 phương thức dành cho các luồng daemon. Đó là:
  - public void setDaemon(boolean on) - on = [true; false]
  - public boolean isDaemon()

  Chú ý: chỉ có thể gọi hàm setDeamon(boolean) khi thread chưa được chạy.

Ví dụ

- Luồng Deamon
- Set luồng Deamon
- Ví dụ về luồng Deamon bị dừng khi tất cả các luồng khác kết thúc.

3.6. Sự ưu tiên của luồng

- Hầu hết các chương trình Java đều làm việc với đa luồng.
- Các nhân của CPU thì có khả năng chạy chỉ 1 luồng tại một thời điểm.
- Các thread với quyền ưu tiên cao hơn là quan trọng hơn với một chương trình nên được cấp phát thời gian bộ vi xử lý trước các thread có quyền ưu tiên thấp hơn. Tuy nhiên, các quyền ưu tiên của thread bảo đảm thứ tự trong đó các thread thực thi và phụ thuộc rất nhiều vào platform.

3.6. Sự ưu tiên của luồng (tiếp)

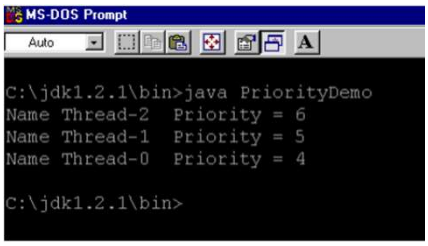
- Mỗi một luồng có quyền ưu tiên của chúng.
  Giá trị ưu tiên nằm trong khoảng:
  - **Thread.MIN_PRIORITY** = 1
  - **Thread.MAX_PRIORITY** = 10
- Ngầm định một luồng có quyền ưu tiên **Thread.NORM_PRIORITY** thì có giá trị 5.
- Quyền ưu tiên có thể được thiết lập bởi phương thức **setPriority(int)**.
- Để lấy giá trị ưu tiên, dùng phương thức **getPriority()**.

☞Khi một luồng mới được khởi tạo, giá trị quyền ưu tiên của nó là bao nhiêu?

3.6. Sự ưu tiên của luồng (tiếp)

```
class ThreadPriority extends Thread {
    int sleep;
    static int prio = 3;

    public ThreadPriority() {
        sleep += 100;
        prio++;
        setPriority(prio);
    }

    public void run(){
        try {
            Thread.sleep(sleep);
            System.out.println("Name "+getName()+ "  Priority = "+ getPriority());
        }
        catch(InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

```
class PriorityDemo
{
    ThreadPriority t1, t2, t3;

    public PriorityDemo(){
        t1 = new ThreadPriority();
        t1.start();
        t2 = new ThreadPriority();
        t2.start();
        t3 = new ThreadPriority();
        t3.start();
    }

    public static void main(String args[]){
        new PriorityDemo();
    }
}
```



Last thread with highest priority (6) will run first

3.7. Đồng bộ luồng
Thread Synchronization

- Khi làm việc với nhiều luồng, có thể xảy ra trường hợp nhiều luồng đều muốn truy cập đến cùng 1 biến trong cùng 1 thời gian.
- Trong trường hợp này, ta cần để 1 luồng kết thúc nhiệm vụ của nó, sau đó mới cho phép luồng tiếp theo thực thi. Điều này có thể thực hiện được thông qua từ khóa **synchronized()**.

Không sử dụng Synchronized

```
public class Counter {
    private int c = 0; // Start with 0

    public void increment() {
        c++; // Increase by 1
    }

    public int value() {
        return c;
    }
}
```


ThreadInterference.java

```
public class ThreadInterference extends Thread {

    private Counter counter = new Counter();

    static void threadMessage(String message) {
        String threadName = Thread.currentThread().getName();
        System.out.format("%s: %s%n", threadName, message);
    }

    @Override
    public void run() {
        try {
            counter.increment();
            Thread.sleep(1000); //Simulate we are doing something here
            threadMessage("" + counter.value());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

ThreadInterferenceExample.java

```
public class ThreadInterferenceExample {

    public static void main(String[] args) {
        ThreadInterference test = new ThreadInterference();
        Thread t1 = new Thread(test);
        t1.start();

        Thread t2 = new Thread(test);
        t2.start();
    }
}
```

Here, we expected that:
Thread t1 should start Counter object with 0, then increase to 1.
Thread t2 should start after that, increase Counter from 1 to 2.

Actual Output

```
Thread-2: 2
Thread-1: 2
```

Thread t1 result is lost, overwritten by Thread t2.

Sử dụng Synchronized

```
public class ThreadInterference extends Thread {

    private Counter counter = new Counter();

    static void threadMessage(String message) {
        String threadName = Thread.currentThread().getName();
        System.out.format("%s: %s%n", threadName, message);
    }

    @Override
    public void run() {
        synchronized (counter) {
            try {
                counter.increment();
                Thread.sleep(1000); //Simulate we are doing something
                  here
                threadMessage("" + counter.value());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```


Kết quả thực tế

```
Thread-2: 1
Thread-1: 2
```

Đây là kết quả chính xác, ta hằng mong đợi.

Tác động qua lại giữa các luồng
Các phương thức wait(), notify(), và notifyAll()

Khi luồng A bắt gặp 1 phương thức wait(), nó sẽ đợi. Luồng A lưu lại trạng thái này cho đến khi một luồng khác đánh thức nó bằng phương thức notify().

Đọc tài liệu và chạy các ví dụ trong Core Java Tutorial

Ví dụ 1: wait() và notify()

```
1  class ThreadA {
2      public static void main(String [] args) {
3          ThreadB b = new ThreadB();
4
5          b.start();
6
7          synchronized(b) {
8              try {
9                  System.out.println("Waiting for b to complete...");
10                 b.wait();
11             } catch (InterruptedException e) {}
12         }
13
14         System.out.println("Total is: " + b.total);
15     }
16 }
17
18 class ThreadB extends Thread {
19     int total;
20
21     public void run() {
22         synchronized(this) {
23             for(int i=0;i<100;i++) {
24                 total += i;
25             }
26
27             notify();
28         }
29     }
30 }
```

Ví dụ 2: wait() và notifyAll()

```
1  public class Reader extends Thread {
2      Calculator c;
3
4      public Reader(Calculator calc) {
5          c = calc;
6      }
7
8      public void run() {
9          synchronized(c) {
10             try {
11                 System.out.println("Waiting for calculation...");
12                 c.wait();
13             } catch (InterruptedException e) {}
14         }
15         System.out.println("Total is: " + c.total);
16     }
17
18     public static void main(String [] args) {
19         Calculator calculator = new Calculator();
20         calculator.start();
21         new Reader(calculator).start();
22         new Reader(calculator).start();
23         new Reader(calculator).start();
24     }
25 }
26
27 class Calculator extends Thread {
28     int total;
29
30     public void run() {
31         synchronized(this) {
32             for(int i=0;i<100;i++) {
33                 total += i;
34             }
35             notifyAll();
36         }
37     }
38 }
```

Using notifyAll() When Many Threads May Be Waiting

3.8. Deadlock

- *Deadlock* (khóa chết) miêu tả một trạng thái mà có 2 luồng trở lên bị ngăn chặn vĩnh viễn, đợi lẫn nhau.
- Khi một chương trình Java bị đình trệ hoàn toàn, các luồng đình trệ đợi nhau mãi. Trong khi các luồng khác cần phải tiếp tục chạy, cuối cùng ta sẽ phải ngắt chương trình (bằng một phương pháp cứng nào đó).

Ví dụ Deadlock

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s has bowed to me!%n",
                    this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s has bowed back to me!%n",
                    this.name, bower.getName());
        }
    }
```


Ví dụ Deadlock

```
    public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

Kết quả Deadlock

```
Alphonse: Gaston has bowed to me!
Gaston: Alphonse has bowed to me!
```

- Then, the application stand here, cannot escape this situation. We have only choice to kill this application.

Q&A