Programming with Java

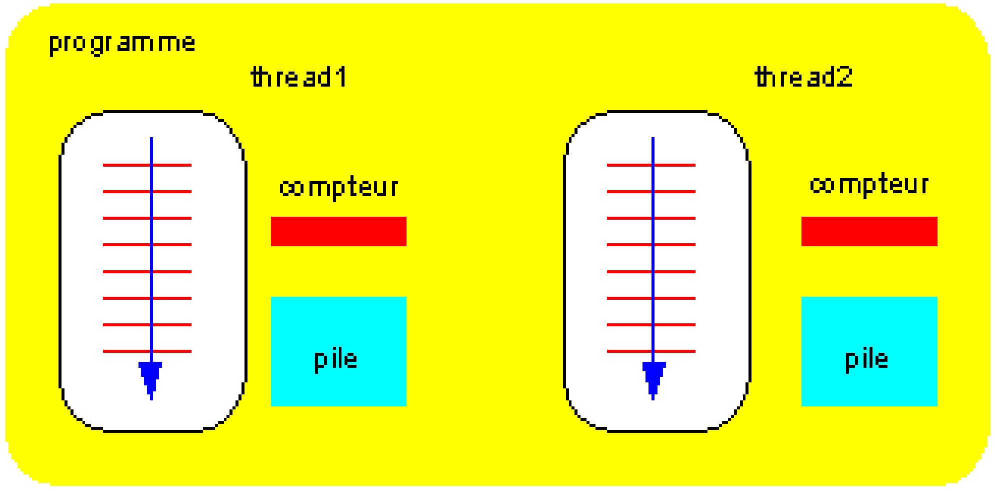
Threads and Synchronisation

P. Bakowski

bako@ieee.org

The threads

Java applications can run a simple execution of a program or several excutions of the program called threads. A thread must be run in a program; it can exploit the resources of the program but it uses its own program counter and stack.



The threads

To run a thread we must start its execution from `run()` method.
This method contains the program to be executed by the thread.

The class `Thread` provides the framework for the creation of a thread.

The `run()` method of this class is `void` and must be replaced by our `run()`method.

There are two ways to implement the `run()`method

- by creating a sub-class of the `Thread` with our `run()` method.
- by the implementation of our `run()` method in `Runnable` interface

SimpleThread extends Thread

```
public class TwoThreadsTest {
  public static void main (String[] args) {
  SimpleThread one = new SimpleThread("ONE");
  SimpleThread two = new SimpleThread("TWO");
  one.start();
  two.start(); }
}
class SimpleThread extends Thread {
  public SimpleThread(String str) { super(str); }
  public void run() {
  for(int i = 0; i < 4; i++) {
    System.out.println(i + " " + getName());
    try { sleep((long)(Math.random() * 1000));
    } catch (InterruptedException e) {}
  }
  System.out.println("END! " + getName());
  }
}
```

SimpleThread extends Thread

```
pi@raspberrypi ~/java/MyClasses
$ java TwoThreadsTest
0 ONE
0 TWO
1 ONE
1 TWO
2 TWO
2 ONE
3 ONE
END! ONE
3 TWO
END! TWO
```

Execution result for `TwoThreadsTest`

SimpleRunnable implements Runnable

```
public class SimpleRunnableTest {
  public static void main (String[] args) {
  Thread t_one = new Thread(new SimpleRunnable(),«one»);
  Thread t_two = new Thread(new SimpleRunnable(),«two»);
  t_one.start();
  t_two.start(); }
}
class SimpleRunnable implements Runnable {
  public void run() {
  for(int i = 0; i < 4; i++) {
    System.out.println(i + " " + Thread.currentThread().getName());
    try { Thread.sleep((long)(Math.random() * 1000));
    } catch (InterruptedException e) {}
  }
  System.out.println("END! " + Thread.currentThread().getName());
  }
}
```

SimpleRunnable implements Runnable

```
pi@raspberrypi ~/java/MyClasses $ java SimpleRunnableTest
0 two
0 one
1 two
1 one
2 two
3 two
END! two
2 one
3 one
END! one
```

Execution result for `SimpleRunnableTest`

Life cycle of a thread

A thread terminates normally when its `run()` method ends. We can not resume thread execution but we can know its status through the method `isAlive()`.

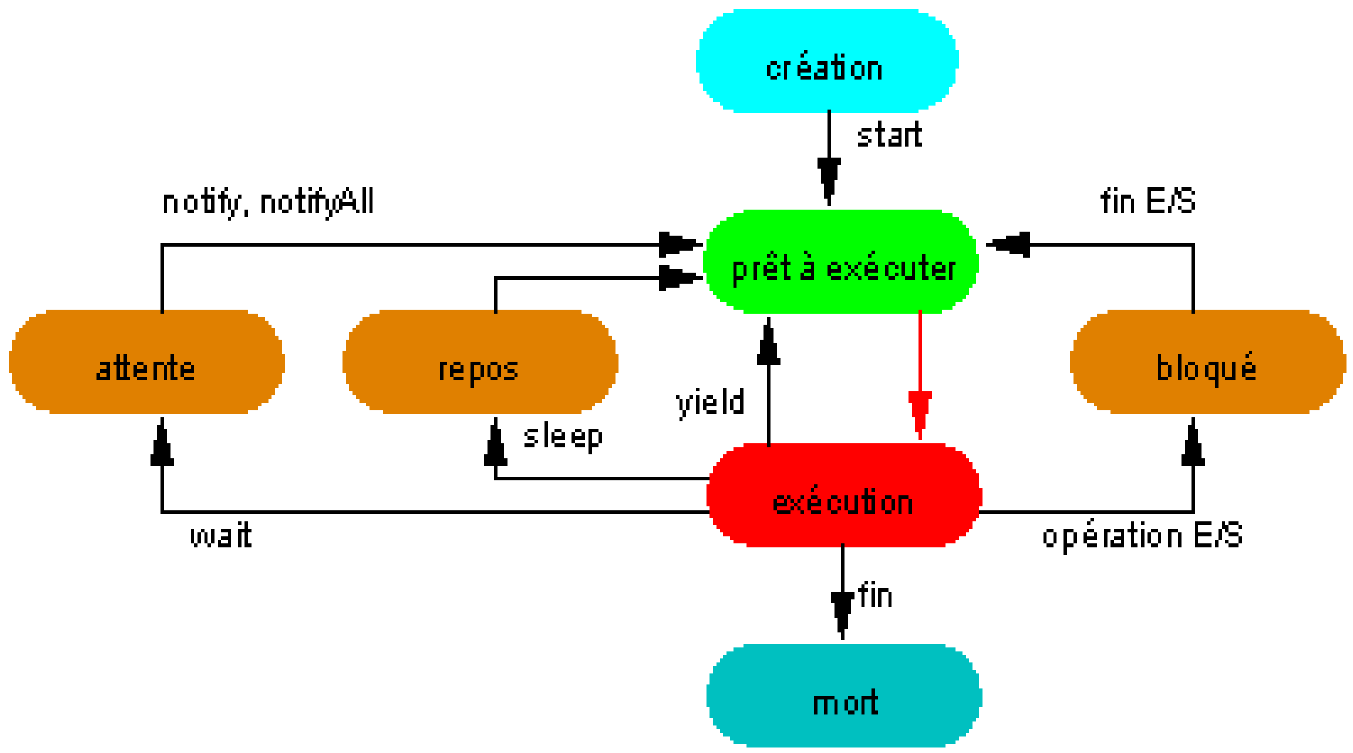
The thread is actually launched by its `start()`method, at which point it switches to the « ready to run » state. The thread with the highest priority in the « ready to run » state switches to the "execution" state.

A thread enters "off" state when making request for input-output operations. At the end of E/S operation the "off" thread becomes « ready to run » .

Calling `sleep()` method on the thread in « execution » state causes the transition to « rest » state.

When a thread performs the `wait()` call, it enters into « wait » state waiting for an object. A thread in a « wait » state is unlocked by a `notify()` generated by another thread associated to this object. Each thread in a « wait » moves to « ready to run » after the notification by `notifyAll()`.

Life cycle of a thread



Each thread has a priority between `Thread.MIN_PRIORITY` (1) and `Thread.MAX_PRIORITY` (10). At the activation, a thread starts with `Thread.NOR_PRIORITY` equal to 5.

Life cycle of a thread

```
public class ThreadStateTest {
public static void main (String[] args) {
ThreadState thread1 = new ThreadState("thread1");
ThreadState thread2 = new ThreadState("thread2");
ThreadState thread3 = new ThreadState("thread3");
ThreadState thread4 = new ThreadState("thread4");
thread3.setPriority(10);thread4.setPriority(10);
System.out.println("Threads created");
thread1.start();thread2.start();
thread3.start();thread4.start();
System.out.println("Threads in ready state"); }
}
```

Life cycle of a thread

```
class ThreadState extends Thread { //  internal class
private int timeSleep;
  public ThreadState(String name) { super(name);
  timeSleep = (int) (Math.random()*5000);
  System.out.println("Name:"+getName()+"time of rest:"+
                      timeSleep);
  }
  public void run() {
  try {
    System.out.println(getName() + " put in rest");
    Thread.sleep(timeSleep);
    } catch (InterruptedException e) {
    System.out.println(e.toString()); }
    System.out.println(getName() + " rest ended");
  }
}
```

Life cycle of a thread

Execution result :

```
Name:thread1 time of rest:3551
Name:thread2 time of rest:2356
Name:thread3 time of rest:1566
Name:thread4 time of rest:1466
Threads created
thread1 put in rest
thread2 put in rest
thread3 put in rest
thread4 put in rest
thread4 rest ended
thread3 rest ended
thread2 rest ended
thread1 rest ended
```

Synchronization

Sharing and protection of resources used by multiple threads require synchronisation mechanism.
To provide the synchronization Java uses the mechanism of monitors.

Each object containing `synchronized` methods is a monitor.

The monitor allows you to lock the execution of a method.

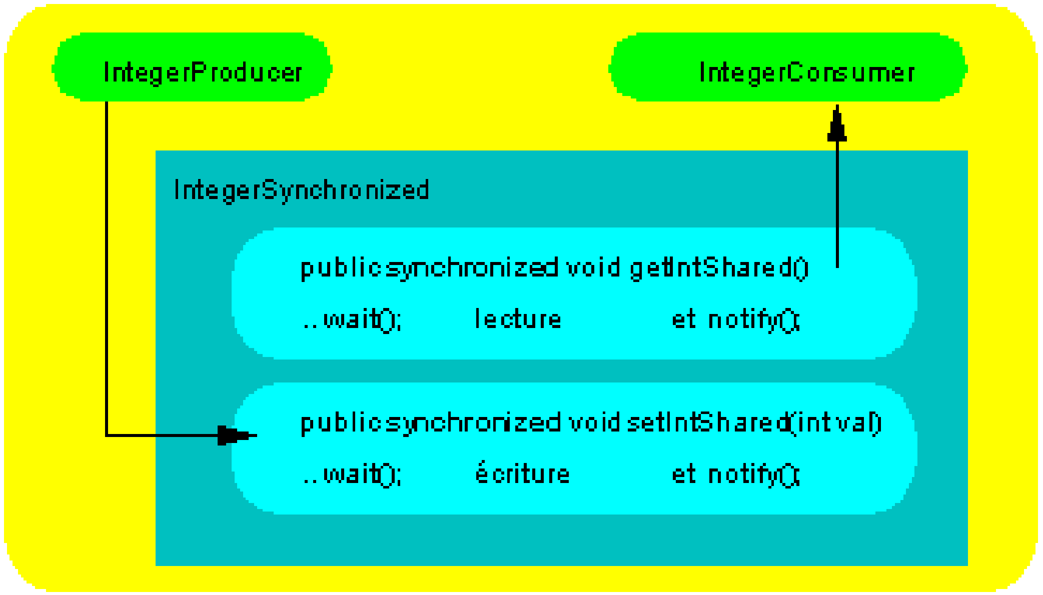
If there are multiple `synchronized` methods, one of these is active at a time on an object. All other objects that attempt to invoke synchronized methods must wait.

When a synchronized method completes execution, the lock on the object is released and the monitor signals this event to waiting process by the `notify()` method.
If the thread calls `notifyAll()`, then all threads waiting for the object move to the state « ready to run ».

Synchronization

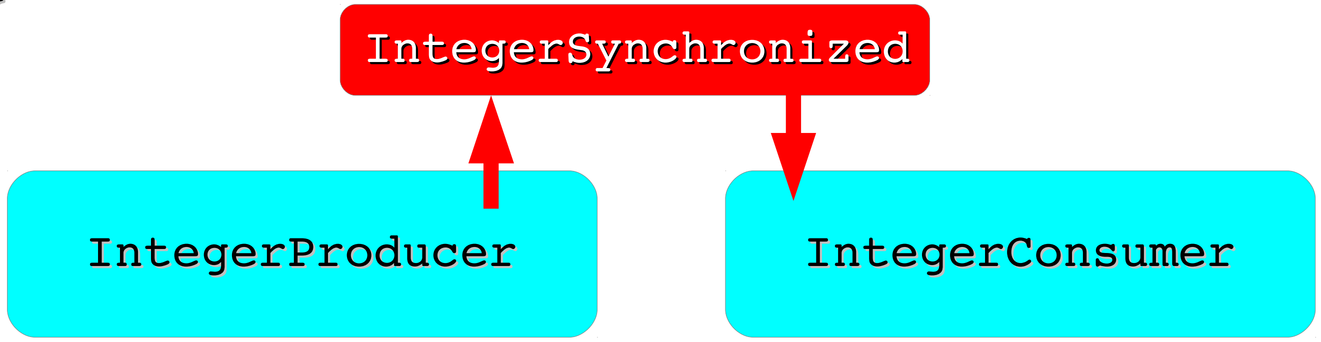
Example:
In this example, we use the monitors in two correlated threads:
the producer thread and the consumer thread.

Values (integers) produced by the producer are placed in a box synchronized by the method `setIntShared(int val)` and recovered by another synchronized method `getIntShared()` initiated by the consumer thread.


IntegerProducer
IntegerConsumer
IntegerSynchronized
public synchronized void getIntShared()
...wait(); lecture et notify();
public synchronized void setIntShared(int val)
...wait(); écriture et notify();

Producer - Cunsumer

```
public class ProducerConsumer {
public static void main (String[] args) {
IntegerSynchronized is = new IntegerSynchronized();
IntegerProducer ip = new IntegerProducer(is);
IntegerConsumer ic = new IntegerConsumer(is);
ip.start(); // activation of producer
ic.start(); // activation of consumer
}
}
```



Producer : IntegerProducer

```
public class IntegerProducer extends Thread {
private IntegerSynchronized ikeep;
public IntegerProducer(IntegerSynchronized ipar) {
super("IntegerProducer");
ikeep = ipar;
}
public void run() {
for(int counter=1; counter<=10;counter++) {
try { Thread.sleep((int) (Math.random()*3000));
} catch (InterruptedException e)
{ System.out.println(e.toString()); }
ikeep.setIntShared(counter); }
// writing to shared variable
System.out.println(getName() + " end of production"); }
}
```

Consumer : IntegerConsumer

```
public class IntegerConsumer extends Thread {
private IntegerSynchronized ikeep;
public IntegerConsumer(IntegerSynchronized ipar) {
super("IntegerConsumer");
ikeep = ipar;
}
public void run() {
int val, sum = 0;
  do { // boucle de lecture
  try { Thread.sleep((int) (Math.random()*3000));
      } catch (InterruptedException e)
  { System.out.println(e.toString()); }
  val = ikeep.getIntShared();
  // reading from the shared variable
  sum += val; }
  while (val != 10);
System.out.println(getName()+"end of consumption-the sum
is "+sum);}
}
```

Variable - synchronized

```
public class IntegerSynchronized {
private int intShared = -1;
private boolean inscriptible = true; // write permission
public synchronized void setIntShared(int val) {
while(!inscriptible) {
try { wait(); }  // waiting for notify()
catch(InterruptedException e) { e.printStackTrace(); } }
System.out.println(Thread.currentThread().getName() + " put
intShared in " + val);
intShared = val; // writing to shared variable
inscriptible = false;
notify();
}
public synchronized int getIntShared() {
while(inscriptible) { // in writing ?
try { wait(); }
catch(InterruptedException e) { e.printStackTrace(); } }
inscriptible = true;
notify();
System.out.println(Thread.currentThread().getName() + " get value
from intShared " + intShared);
return intShared; }  // reading from shared variable
}
```

Variable - synchronized

```
IntegerProducer put intShared in 1
IntegerConsumer get value from intShared 1
IntegerProducer put intShared in 2
IntegerConsumer get value from intShared 2
IntegerProducer put intShared in 3
IntegerConsumer get value from intShared 3
IntegerProducer put intShared in 4
IntegerConsumer get value from intShared 4
IntegerProducer put intShared in 5
IntegerConsumer get value from intShared 5
IntegerProducer put intShared in 6
IntegerConsumer get value from intShared 6
IntegerProducer put intShared in 7
IntegerConsumer get value from intShared 7
IntegerProducer put intShared in 8
IntegerConsumer get value from intShared 8
IntegerProducer put intShared in 9
IntegerConsumer get value from intShared 9
IntegerProducer put intShared in 10
IntegerProducer end of production
IntegerConsumer get value from intShared 10
IntegerConsumerend of consumption-the sum is 55
```

Summary

- The mechanism of `Thread`
- The states of thread (ready, active, waiting,..);
- The priority of thread
- Synchronization and monitor mechanism