Advanced Object-Oriented Programming
Chapter 1 - Threads

UFAZ / 2018-2019
Cecilia ZANNI-MERK

Agenda

- Introducing Threads
- Synchronization
- Scheduling and Priority
- Thread Groups

Notion of process

A **process** is a running program. The system allocates a portion of memory to each process (to store its instructions, variables, etc.) and it associates information to it (identifier, priorities, access rights, etc.).

A process runs on a system processor. It needs resources: the processor, memory, inputs/outputs. Some resources have only one access point and can therefore only be used by one process at a time (for example, a printer).

Notion of process

Processes are then said to be **mutually exclusive** if they share the same so-called **critical resource**. It is necessary to have a **synchronization** policy for such a shared resource.

For example, regarding printing, there is a process that manages the requests and the queue for these requests. Such a process, that is invisible and always running as long as the system is running, is called a **daemon**.

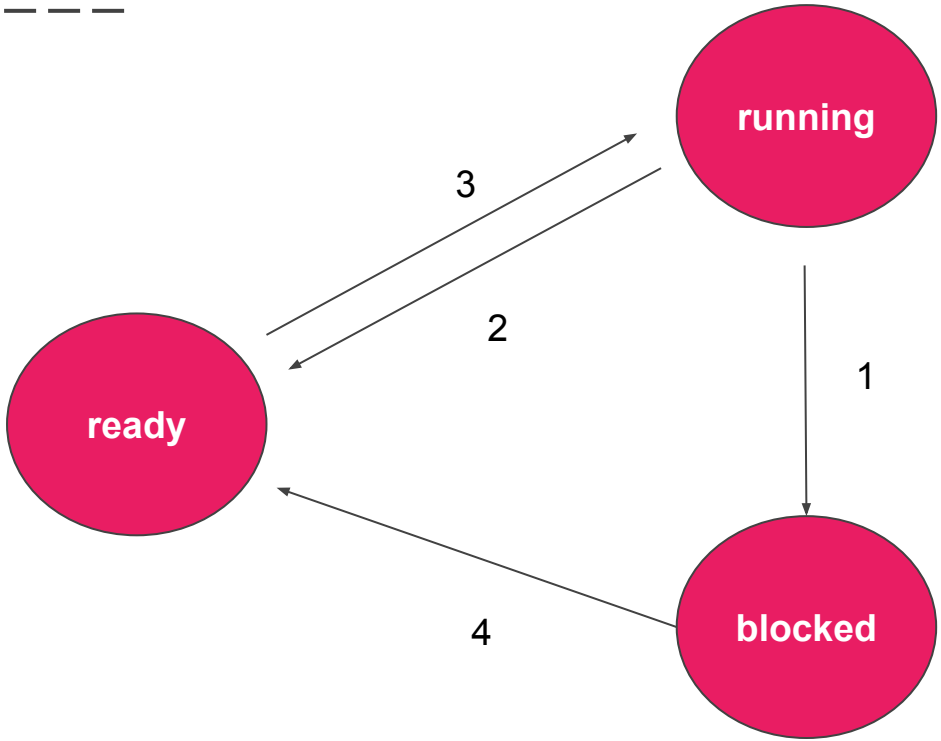
Notion of process

A **daemon** is a process that runs in the background like the print service. Usually a daemon is started by the operating system at boot time and stops at system shutdown.

A process can be in different states :

- running
- ready to run (but no free processor to run it)
- blocked (because of lack of resources)

Notion of process



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Threads

A **thread** or **light process** is a process within a process.

In Java, when we launch the virtual machine to execute a program, we launch a process; this process is composed of several threads: the main thread (which corresponds to the `main` method) and other threads like the *garbage collector*.

So a process is composed of several threads (or tasks) and will have to share its resources between its different threads.

Threads

In Java there are two ways to create a thread:

- extending the `Thread` class: we will then have an object that controls a task,
- implementing the `Runnable` interface: we will have an object that represents the source code to execute.

Inheriting from the `Thread` class

When we inherit from the `Thread` class, we must rewrite the `void run()` method so that it executes the expected behavior.

Then once a `Thread` object is created, this object can invoke the `void start()` method of the `Thread` class which itself invokes the rewritten `run()` method.

Example

TestThread.java TicTac.java

```
1  public class TestThread extends Thread
2  {
3      private String s ;
4
5      public TestThread (String s)
6      {
7          this.s=s ;
8      }
9
10
11     public void run()
12     {
13         while (true)
14         {
15                 System.out.println(s);
                   try {  sleep (100); }
                   catch (InterruptedException e)  {  }
           }
       }
   }
```

TestThread.java TicTac.java

```
1  public class TicTac
2  {
3
4      public static void main (String[] args)
5      {
6                  Thread tic = new TestThread( "TIC" ) ;
7                  Thread tac = new TestThread( "TAC" ) ;
8                  tic.start();
9                  tac.start();
10     }
11 }
12
```

Overview of the `Thread` class

Constants

- `static int MAX_PRIORITY : maximum priority = 10`
- `static int MIN_PRIORITY : minimum priority = 1`
- `static int NORM_PRIORITY : normal priority = 5`

Constructors

- `Thread() ;`
- `Thread(Runnable target, String name)`
- `. . .`

Overview of the `Thread` class

Methods

- `void start()` : invokes the `run` method
- `void run()` : executes the thread and can launch `InterruptedException`
- `int getPriority()` : returns the priority level
- `static Thread currentThread()` : returns the thread that is running
- `static void sleep(long m)` : suspends thread activity running for `m` milliseconds

Overview of the `Thread` class

Methods

- `static void yield()` : suspends the activity of the running thread and allows other threads to run
- `void join()` : waits for the end of the `Thread` object that invokes it
- `void interrupt()` : interrupt `this`
- `static boolean interrupted()`: tests if the current thread has been interrupted

Overview of the `Thread` class

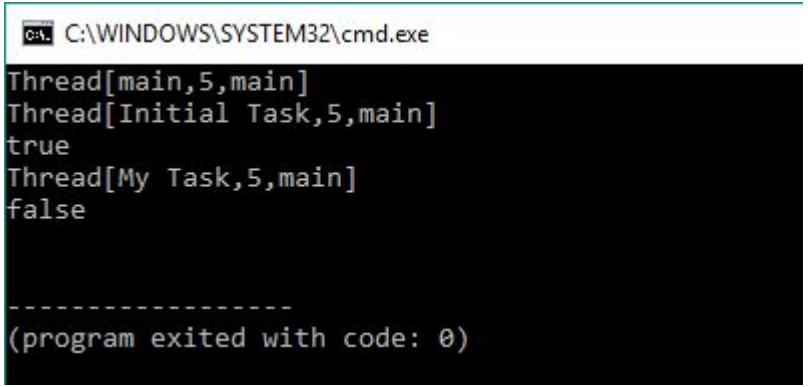
Methods

- `boolean isInterrupted()` : test if `this` has been interrupted
- `boolean isAlive()` : returns `true` if the thread has not finished
- `String toString()` : returns the thread name, its priority, its group in the `ThreadGroup` class

Example

MyTask.java

```java
public class MyTask {

    public static void main (String[] args) throws InterruptedException
    {
        System.out.println (Thread.currentThread()) ;
        Thread initialTask = Thread.currentThread() ;
        initialTask.setName ("Initial Task");
        Thread.sleep (1000); //Sleep is a class method
        System.out.println (initialTask);
        //the name of the thread has been changed, but not the name of its gorup
        System.out.println(initialTask.isAlive());
        Thread myTask = new Thread();
        myTask.setName("My Task");
        System.out.println(myTask);
        System.out.println(myTask.isAlive());
    }
}
```



Implementing the `Runnable` the interface

The `Runnable` interface contains only the `run()` method that is to be implemented.

To launch a thread with a class implementing `Runnable` we use the constructor of the `Thread` class which takes as parameter a `Runnable` object.

Example

TestThread2.java

```
1  public class TestThread2 implements Runnable
2  {
3      private String s;
4
5      public TestThread2 (String s)
6      {
7          this.s = s;
8      }
9
10     public void run()
11     {
12         while (true)
13         {
14                 System.out.println(s);
15                 try { Thread.sleep (100); }
16                 catch (InterruptedException e)  {  }
17         }
18     }
19 }
```

TicTac2.java

```
1  public class TicTac2
2  {
3
4      public static void main (String[] args)
5      {
6                  Thread tic = new Thread( new TestThread2 ( "TIC" )) ;
7                  Thread tac = new Thread (new TestThread2( "TAC" )) ;
8                  tic.start();
9                  tac.start();
10     }
11 }
```

The `join()` method

The `join()` method of the `Thread` class invoked by a `Thread` object `t` holds the thread running until `t` is finished.

The `join()` method launches an `InterruptedException` type exception so it must be used in a `try catch` block.

Example

TicTac3.java

```
public class TicTac3
{

    public static void main (String[] args)
    {
                Thread tic = new TestThread3 ( "TIC" ) ;
                Thread tac = new TestThread3( "TAC" ) ;
                tic.start();
                tac.start();
                try {tic.join();}
                catch (InterruptedException e){}
                System.out.println ("it's over");
    }
}
```

TestThread3.java

```
public class TestThread3 extends Thread
{
    private String s ;

    public TestThread3 (String s)
    {
        this.s=s ;
    }


    public void run()
    {
        for  (int  i = 1 ;   i <= 2;   i ++)
        {
                System.out.print(s + "  ");
                try {  sleep (100); }
                catch (InterruptedException e)  {  }
        }
    }
}
```

```
C:\WINDOWS\SYSTEM32\cmd.exe
TIC  TAC  TAC  TIC  it's over


-----------------
(program exited with code: 0)

Appuyez sur une touche pour continuer...
```

Threads management with `synchronized`

Threads can share resources. It must then be ensured that this resource will only be used by one thread at a time.

To do this, a lock mechanism is used: any object (or array) has a lock that can be opened or closed. A `t1` thread can close a lock on an object (if the lock is not already closed) and when it finishes the execution of some code on the locked object, it reopens the lock.

To prevent another `t2` thread from executing some code on the locked object, this code portion must have the same lock mechanism on this object.

Threads management with `synchronized`

To prevent another `t2` thread from executing some code on the locked object, this code portion must have the same lock mechanism on this object.

We can synchronize

- a method `m`: `synchronized void m()` - here `this` is the object on which the lock is placed
- an object `o`: `synchronized(o)...instructions ...` - here `o` is the object on which the lock is placed

Threads management with `synchronized`

While executing a portion of code marked `synchronized` by a `t1` thread, any other `t2` thread attempting to execute a portion of code marked `synchronized` relative to the same object is suspended.

Threads management with `synchronized`

Remarks:

- attention! an unsynchronized method can modify `this` even if `this` is locked by a synchronized method
- a static method can be synchronized, it then locks its class preventing another static method from executing during its own execution
- a synchronized static method does not prevent changes to class instances by instance methods.

Example with no synchronization

JointBankAccount.java | BankCounter.java | TestBankCounter.java

```
1  public class JointBankAccount
2  {
3      private String nameM;
4      private String nameW;
5      private String accountNbr;
6      private int balance=0;
7
8      public JointBankAccount (String sM, String sW, String aNbr)
9      {
10         this.nameM = sM;
11         this.nameW = sW;
12         this.accountNbr = aNbr;
13     }
14
15     public String toString()
16     {
17         return ("The bank account number "+this.accountNbr+" has a balance of "+this.balance+" euros");
18     }
19
20     public void deposit (int amount)
21     {
22             int result = this.balance;
23             try { Thread.sleep (100); } //processing time
24             catch (Exception e) {}
25             this.balance = amount + result;
26             System.out.println ("deposit of "+amount+ " euros");
27     }
28
29 }
```

Example with no synchronization

JointBankAccount.java | BankCounter.java | TestBankCounter.java

```
1  public class BankCounter extends Thread
2  {
3          private JointBankAccount cj;
4          private int id;
5
6          public BankCounter (JointBankAccount ba, int n)
7          {
8              this.cj = ba;
9              this.id = n;
10         }
11
12         public void run()
13         {
14             System.out.println ("start of transaction on "+this.cj+" at counter number "+this.id);
15             this.cj.deposit(100);
16             System.out.println ("end of transaction on "+this.cj+" at counter number "+this.id);
17         }
18 }
```

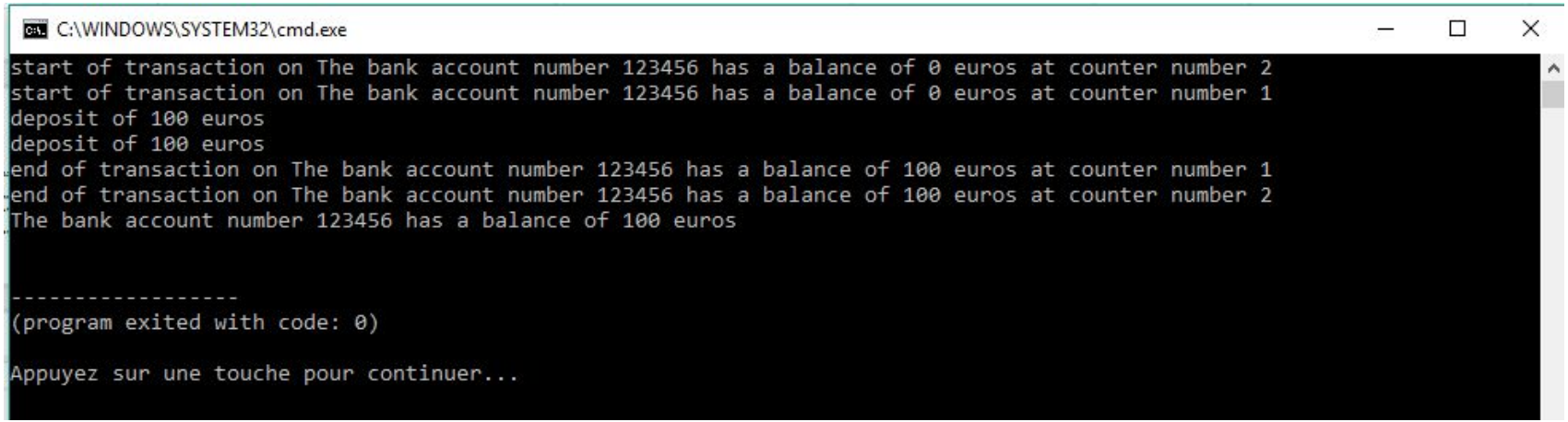
Example with no synchronization

JointBankAccount.java | BankCounter.java | TestBankCounter.java

```
1  public class TestBankCounter {
2
3      public static void main (String[] args) {
4          JointBankAccount anAccount = new JointBankAccount ("Bob", "Alice", "123456");
5          BankCounter bc1 = new BankCounter(anAccount, 1);
6          BankCounter bc2 = new BankCounter(anAccount, 2);
7          bc1.start();
8          bc2.start();
9          try
10         {
11             bc1.join();
12             bc2.join();
13         }
14         catch (InterruptedException e) {}
15         System.out.println (anAccount);
16
17     }
18 }
```

Example with no synchronization

Execution



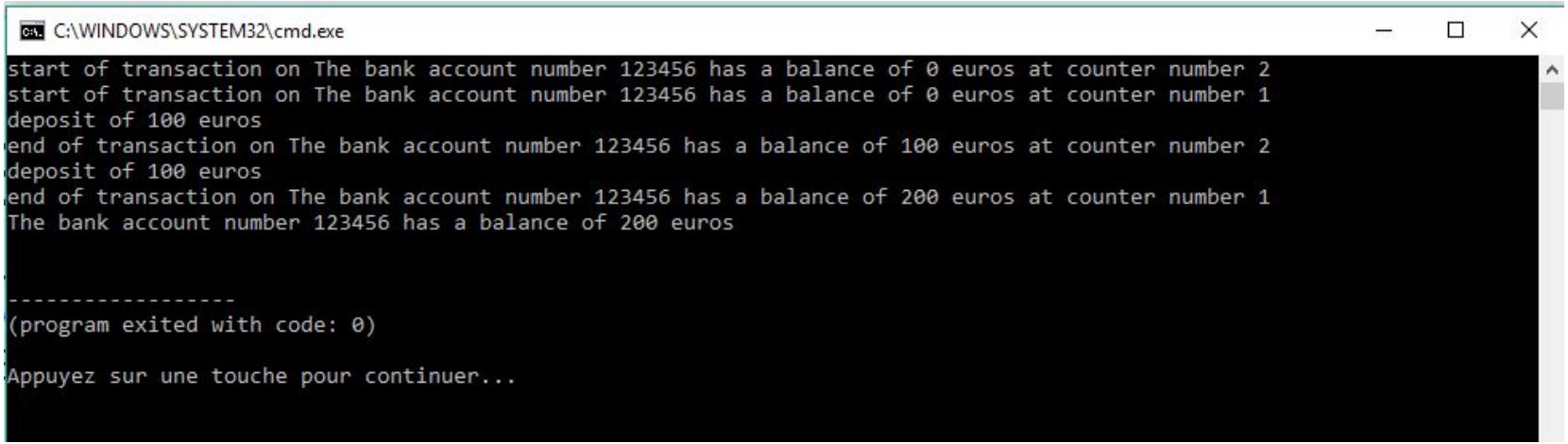
Example with synchronization

JointBankAccount.java | BankCounter.java | TestBankCounter.java

```
1  public class JointBankAccount
2  {
3      private String nameM;
4      private String nameW;
5      private String accountNbr;
6      private int balance=0;
7
8      public JointBankAccount (String sM, String sW, String aNbr)
9      {
10         this.nameM = sM;
11         this.nameW = sW;
12         this.accountNbr = aNbr;
13     }
14
15     public String toString()
16     {
17         return ("The bank account number "+this.accountNbr+" has a balance of "+this.balance+" euros");
18     }
19
20     public synchronized void deposit (int amount)
21     {
22             int result = this.balance;
23             try { Thread.sleep (100); } //processing time
24             catch (Exception e) {}
25             this.balance = amount + result;
26             System.out.println ("deposit of "+amount+ " euros");
27     }
28
29 }
30
```

Example with synchronization

Execution



`wait()` and `notify()`

Another tool to coordinate actions of multiple threads in Java is **guarded blocks**.

Such blocks keep a check for a particular condition before resuming the execution.

With that in mind, we can use:

- `Object.wait()` - to suspend a thread
- `Object.notify()` - to wake a thread up

wait() and notify()

More in detail:

- public final void wait()
- public final void wait(long maxMilli)
- public final void wait(long maxMill, int maxNano)
- public final void notify()
- public final void notifyAll()

`wait()` and `notify()`

Simply put, a call to `wait()` forces the current thread to wait until some other thread invokes `notify()` or `notifyAll()` on the same object.

For this, the current thread must have executed:

- a synchronized instance method for the given object
- the body of a synchronized block on the given object
- synchronized static methods for objects of type Class

Example

In this example we have 4 classes:

- a `Producer` class that places objects in a warehouse
- a `Consumer` class that takes the objects from a warehouse
- a `Warehouse` class
- a `Test` class

Example

A `Producer` can put an object only if the warehouse is not full; it must therefore wait until a Consumer has emptied the warehouse,

A `Consumer` cannot take an item if the warehouse is empty; it must therefore wait until a Producer has filled in the warehouse.

First ver sation

```
public class Warehouse
{
    private static final int MAX_NBR = 3;
    private int nbObj = 0;
    private String s;

    public Warehouse (String s) { this.s = s; }

    public int getNbObj() { return this.nbObj; }

    public boolean isEmpty() { return (this.nbObj==0); }

    public boolean isFull() { return (this.nbObj==this.MAX_NBR); }

    public String toString() { return (" ( "+this.nbObj+" objets) in "+this.s) ; }

    public synchronized void put()
    {
        try
        {
            while (isFull()) {  wait(); System.out.println ("producer aslept"); }
        }
        catch (Exception e) {}
        this.nbObj++;
        notifyAll();
    }

    public synchronized void take()
    {
        try
        {
            while (isEmpty()) { wait(); System.out.println ("consumer aslept"); }
        }
        catch (Exception e) {}
        this.nbObj--;
        notifyAll();
    }
}
```

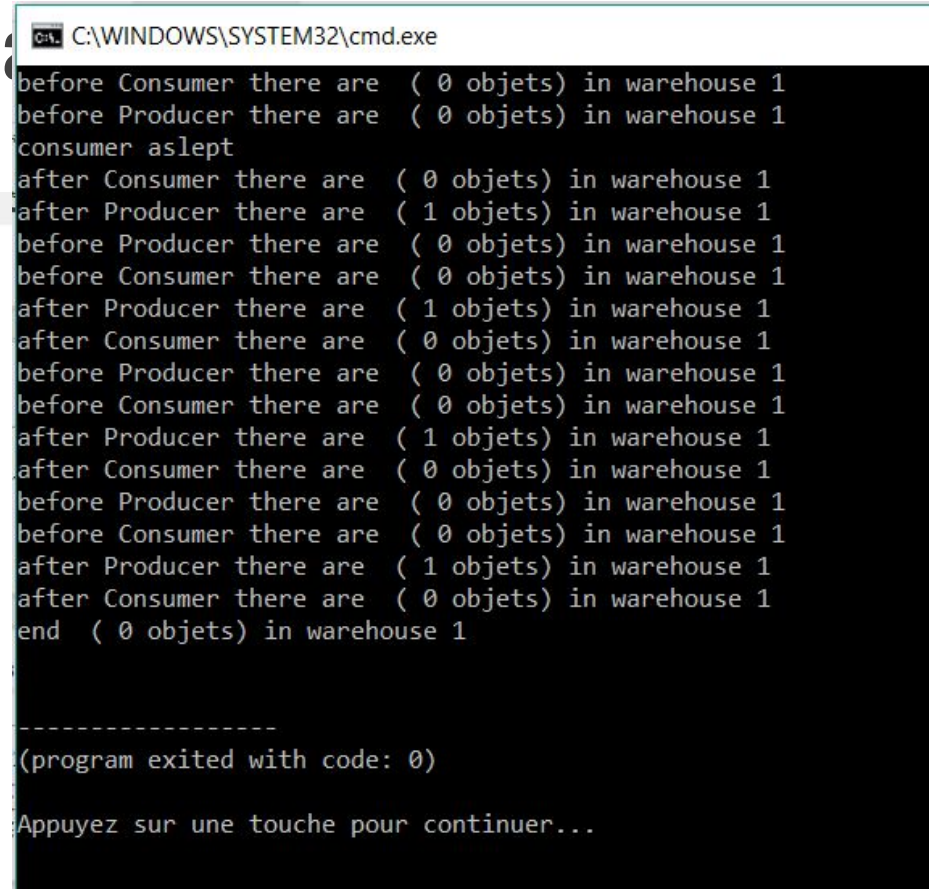
First version: `Warehouse` takes care of synchronisation

```
public class Producer extends Thread
{
    private Warehouse e;
    private String nom;

    public Producer (Warehouse w, String s) { this.e = w; this.nom = s;}

    public void run()
    {
        for (int i=1; i<=4; i++)
        {
            System.out.println ("before Producer there
            e.put();
            System.out.println("after Producer there a
        }

    }
}
```

```
public class Consumer extends Thread
{
    private Warehouse e;
    private String nom;

    public Consumer (Warehouse w, String s) { this.e = w; this.nom = s;}

    public void run()
    {
        for (int i=1; i<=4; i++)
        {
            System.out.println ("before Consumer there are "+e);
            e.take();
            System.out.println("after Consumer there are "+e);
        }

    }
}
```

First version: `Warehouse` ta

```
public class Test
{
    public static void main (String[] args)
    {
        Warehouse e = new Warehouse ("warehouse 1");
        Producer p = new Producer (e, "producer");
        Consumer c = new Consumer (e, "consumer");
        p.start();
        c.start();
        try
         {
             p.join();
             c.join();
         }
         catch (Exception ex) {}
         System.out.println ("end "+e);
            }
}
```



2nd version: `Producer` and `Consumer` manage synch.

```
public class Warehouse
{
    private static final int MAX_NBR = 3;
    private int nbObj = 0;
    private String s;

    public Warehouse (String s) { this.s = s; }

    public int getNbObj() { return this.nbObj; }

    public boolean isEmpty() { return (this.nbObj==0); }

    public boolean isFull() { return (this.nbObj==this.MAX_NBR); }

    public String toString() { return (" ( "+this.nbObj+" objets) in "+this.s) ; }

    public synchronized void put()
    {   this.nbObj++;   }

    public synchronized void take()
    {   this.nbObj--;   }
}
```

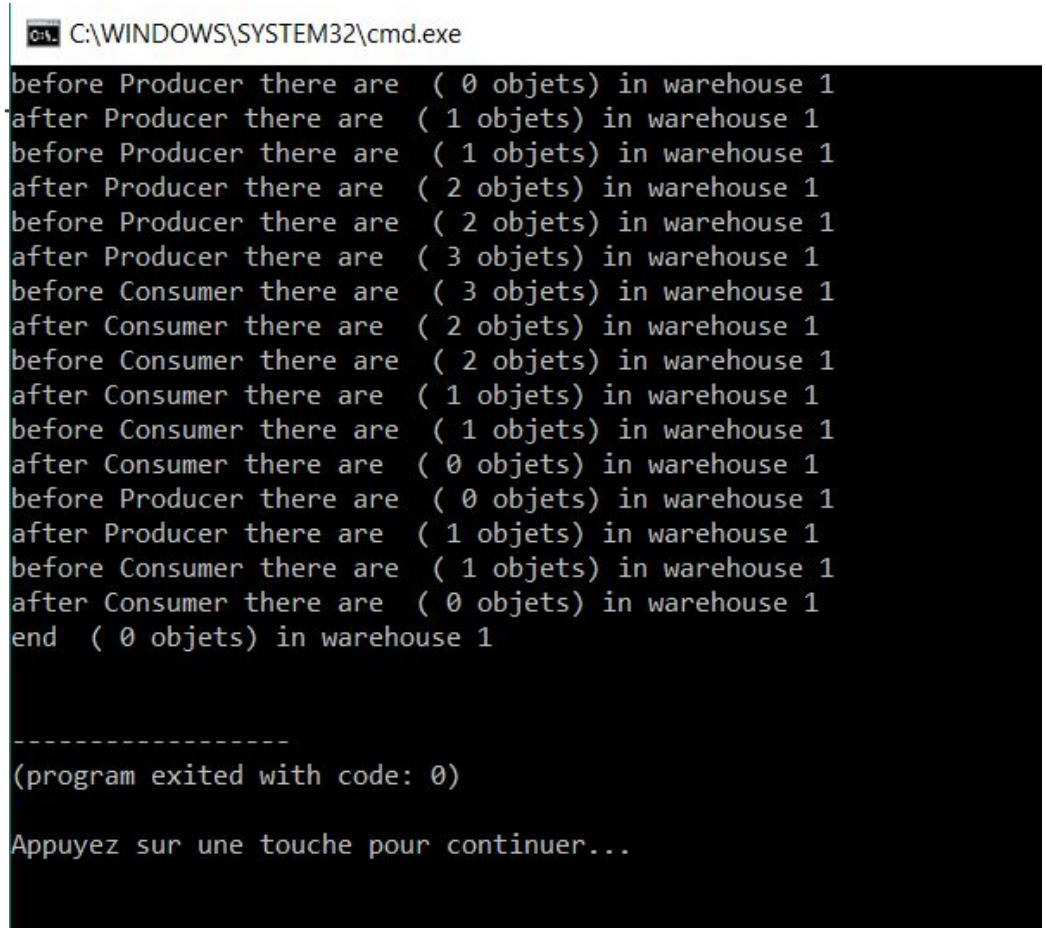
2nd version: `Producer` and `Consumer` manage synch.

```
public class Producer extends Thread
{
    private Warehouse e;
    private String nom;

    public Producer (Warehouse w, String s) { this.e = w; this.nom = s;}

    public void run()
    {
        try
        {
            for (int i=1; i<=4; i++)
                synchronized ( this.e )
                {
                    while (e.isFull()) {e.wait();} // e locks the access of the Producer
                    System.out.println ("before Producer there are "+e);
                    e.put();
                    System.out.println("after Producer there are "+e);
                    e.notifyAll();
                }
        }
        catch (Exception ex) {}
    }
}
```

2nd version: `Producer` and `Consumer` manage synch.

```
public class Consumer extends Thread
{
    private Warehouse e;
    private String nom;

    public Consumer (Warehouse w, String s) { this.e = w; this.nom = s;}

    public void run()
    {
        try
        {
            for (int i=1; i<=4; i++)
                synchronized (this.e)
                {
                    while (this.e.isEmpty()) {this.e.wait();} // e locks the access of the Consumer
                    System.out.println ("before Consumer there are "+this.e);
                    this.e.take();
                    System.out.println("after Consumer there are "+this.e);
                    this.e.notifyAll();
                }
        }
        catch (Exception ex) {}

    }
}
```

2nd version: `Prod` nch.



Final remarks

Supervised exercises

1. Download and test all the examples presented during the lecture
   a. Slide 19. Execute several times the example and compare the results of those executions. What happens if you change the number of milliseconds in the `sleep` method? Explain and justify
   b. Do the same for the examples in slides 29 and 37
   c. Compare the two versions of the same problem in slides 37 and 41. Discuss the advantages and disadvantages of both approaches
2. Download and analyse the source code in slide 43 (classes `MyObject.java`, `MyThread.java`, `Deadlock.java`)
   a. Explain where there are mistakes and propose solutions.
   b. Study the notions of **starvation**, **livelock** and **deadlock** and how to prevent them

Supervised exercises

```
public class MyThread extends Thread
{
    private MyObject o1, o2;

    public MyThread (MyObject obj1, MyObject obj2)
    {
        this.o1 = obj1;
        this.o2 = obj2;
    }

    public void run()
    {
        o1.action1(o2);
    }
}
```

```
public class MyObject
{
    public MyObject() {}

    public synchronized void action1 (MyObject o)
    {
        try { Thread.currentThread().sleep(200); }
        catch (InterruptedException e) {return;}
        o.action2(this);
    }

    public synchronized void action2 (MyObject o)
    {
        try { Thread.currentThread().sleep(200); }
        catch (InterruptedException e) {return;}
        o.action1(this);
    }
}
```

```
public class Deadlock {

    public static void main (String[] args) {
        MyObject obj1 = new MyObject();
        MyObject obj2 = new MyObject();
        MyThread t1 = new MyThread (obj1, obj2);
        t1.setName ("t1");
        MyThread t2 = new MyThread (obj2, obj1);
        t2.setName ("t2");
        t1.start();
        t2.start();

    }
}
```

Mini-Project

To be developed by pairs of students and to be defended in 2 weeks

- Write a `Counter` class that inherits from the `Thread` class; it has a `String` type attribute; its `run()` method counts from `1` to `n` by making a random pause of 0 to 5 seconds between two increments, it displays each incremented value with its name and then displays an end message. Test this class in a `TestCounter` class that starts several `Counter` objects.
- Modify the `run()` method of the `Counter` class so that the thread displays the end message with its order of arrival. Test the change.