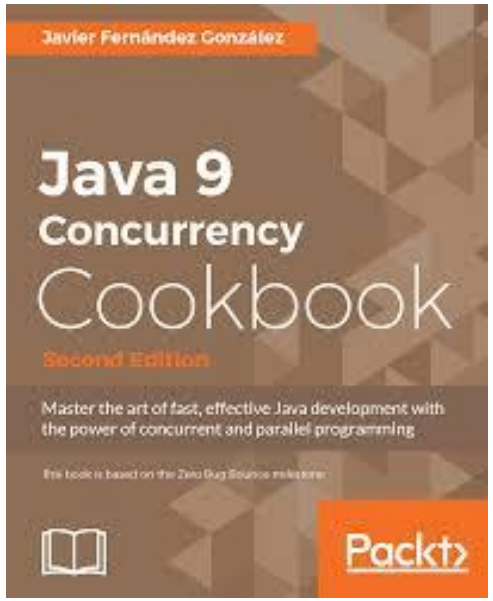
# IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

## JAVA

Ioana Leustean




https://docs.oracle.com/javase/tutorial/essential/concurrency/sleep.html

➢ Orice fir de executie (thread) este instant a clasei Thread

Atributele unui thread

- ID - unic pentru fiecare thread
  - este accesat cu getId, nu poate fi modificat

- Name - este un String
  - este accesat cu : getName, setName
- Priority - un numar intre 1 si 10
  - este accesata cu: getPriority, setPriority
  - in principiu thread-urile cu prioritate mai mare sunt executate primele
  - setarea prioritatii nu ofera garantii in privinta executiei
- Status - este accesat cu: getState, nu poate fi modificat direct (e.g. nu exista setState)

public static enum Thread.State
extends Enum<Thread.State>

Starile posibile ale unui thread:

NEW
A thread that has not yet started is in this state.
RUNNABLE
A thread executing in the Java virtual machine is in this state.
BLOCKED
A thread that is blocked waiting for a monitor lock is in this state.
WAITING
A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
TIMED_WAITING
A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
TERMINATED
A thread that has exited is in this state.

https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html

Doua modalitati de a crea obiecte de tip Thread:

- directa
- implementarea interfetei Runnable
- ca subclasa a clasei Thread

- abstracta
- folosind metodele clasei  Executors

Definirea unui thread prin implementarea interfetei Runnable

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
         Thread t = new Thread (new HelloRunnable());
          t.start();
    }}
```

```
public class HelloTh {

private static Runnable runnable = () ->
                              System.out.println("Hello from a thread!");

public static void main(String args[]) {
                          Thread t = new Thread (runnable);
                          t.start();
  }}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

Definirea unui thread ca subclasa a clasei Thread

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }}
```

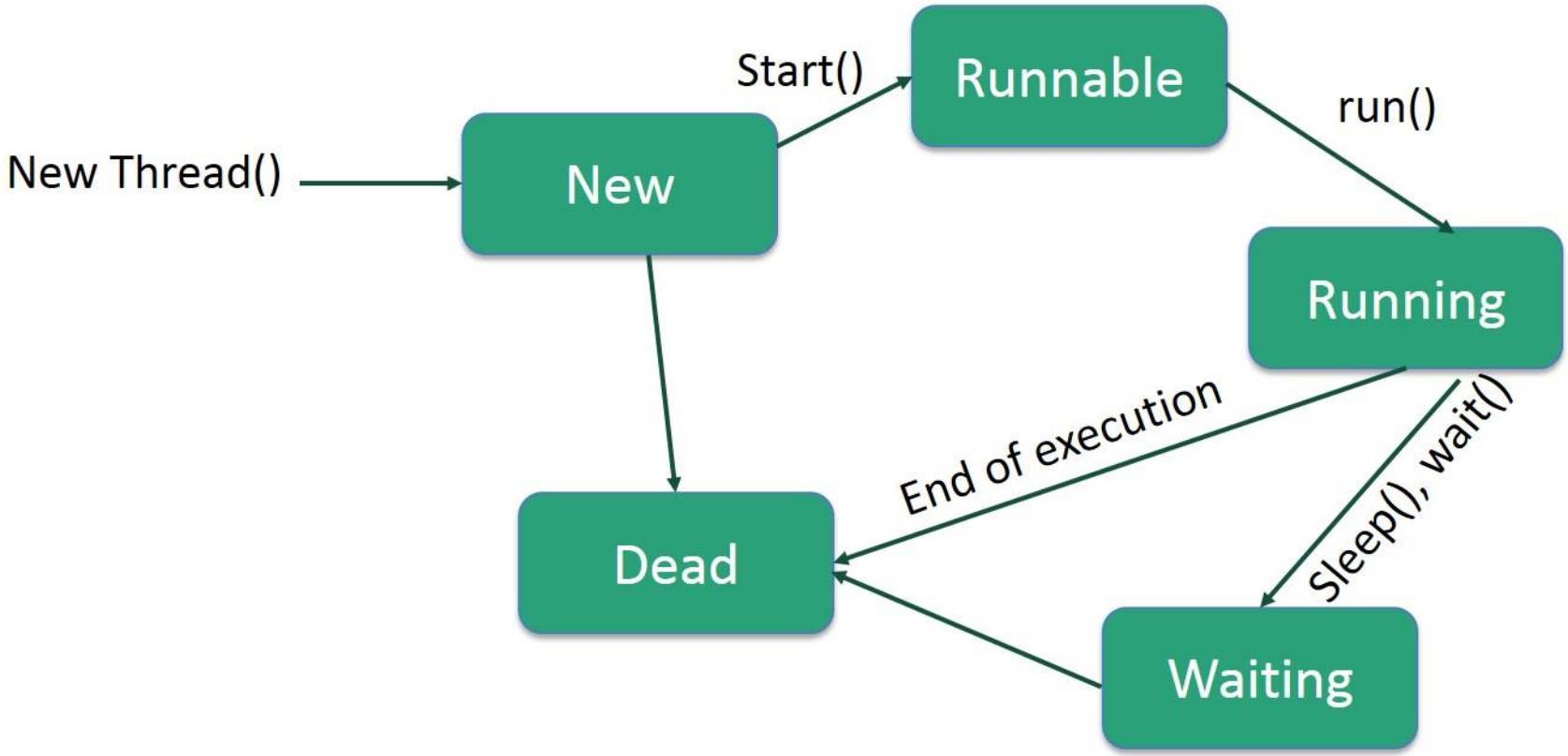
- Executia este nedeterminista

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

      Thread myThread = new Thread(){
       public void run(){
        for (int x = 0; x < 5; x = x + 1)
            System.out.println("Hello from the new thread!");
       }

      myThread.start();
      Thread.yield();
      for (int x = 0; x < 5; x = x + 1)
          System.out.println("Hello from the main thread!");
      myThread.join();
   }
}
```

```
C:\myjava>java HelloThread
Hello from the main thread!
Hello from the main thread!
Hello from the main thread!
Hello from the main thread!
Hello from the main thread!
Hello from the new thread!
Hello from the new thread!
Hello from the new thread!
Hello from the new thread!
Hello from the new thread!

C:\myjava>java HelloThread
Hello from the main thread!
Hello from the main thread!
Hello from the main thread!
Hello from the main thread!
Hello from the new thread!
Hello from the new thread!
Hello from the new thread!
Hello from the new thread!
Hello from the new thread!
Hello from the main thread!
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

# Ciclul de viata al unui thread




Sursa imaginii: https://www.tutorialspoint.com/java/java_multithreading.htm

```
public class Thread
extends Object
implements Runnable
```

Metodele :

- **start()**
  porneste thread-ul intr-un fir de executie separate si invoca run

- **run()**
  este suprascrisa sau apelata din Runnable
- **join(), join(long millisecunde)**
  este invocata de thread-ul curent pe un alt doilea thread;
  thread-ul current este blocat pana cand al doilea thread isi termina executia sau
  pana cand expira timpul (milisecunde)
- **interrupt()**
  intrerupe executia thread-ului;
  este folosit in situatia in care un thread cere altui thread sa isi intrerupa executia

- **boolean isAlive()**
  intoarce true atata timp cat thread-ul nu si-a incetat executia

https://docs.oracle.com/javase/tutorial/essential/concurrency

public class Thread
extends Object
implements Runnable

Metode statice ( se aplica thread-ului current):

- **yield()**
  thread-ul cedeaza randul altui thread

- **sleep(long milisecunde)**
  thread-ul este blocat pentru numarul de milisecunde precizat

- **Thread.currentThread()**
  intoarce o referinta la thread-ul care invoca metoda

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named main of some designated class).

The Java Virtual Machine continues to execute threads until either of the following occurs:

- The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method.

https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

Thread-uri daemon = thread-uri cu prioritate mica, care au rolul de a servi thread-urile utilizator
(e.g. garbadge collector thread)

- **sleep()** arunca exceptie daca threadul este intererupt de un alt thread in timp ce sleep este activa

```
public class SleepMessages {
   public static void main(String args[]) throws InterruptedException {
      String importantInfo[] = { "s1", …,"sn"};

      for (int i = 0;  i < importantInfo.length; i++) {
                 //Pause for 4 seconds
                Thread.sleep(4000);
                System.out.println(importantInfo[i]);
      }}}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/sleep.html

Fara tratarea InterruptedException

```
C:\myjava\tutoracle>javac SleepMessages.java
SleepMessages.java:16: error: unreported exception InterruptedException;
must be caught or declared to be thrown
          Thread.sleep(4000);
```

➢ Exemplu: SimpleThreads

Thread-ul principal creaza un al doilea thread si asteapt ca acesta sa isi termine executia.
Thread-ul secundar este creat prin implementarea interfetei Runnable

```
private static class MessageLoop
      implements Runnable {
      public void run() {
         String importantInfo[] = {"s1", …, "s4"}
         try {
            for (int i = 0;  i < importantInfo.length;i++) {
                Thread.sleep(4000);
                 threadMessage(importantInfo[i]);
            }
         } catch (InterruptedException e) {
            threadMessage("I wasn't done!");
         }}}
```

Crearea thread-ului secundar

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

https://docs.oracle.com/javase/tutorial/essential/concurrency

➢ Exemplu: SimpleThreads

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

    long patience = 1000 * 60 * 60;
    long startTime = System.currentTimeMillis();

     threadMessage("Starting MessageLoop thread");
     Thread t = new Thread(new MessageLoop());
     t.start();
     threadMessage("Waiting for MessageLoop thread to finish");
     t.join();
    threadMessage("Finally!");}
```

```
C:\myjava\tutoracle>java SimpleThreads1
main: Starting MessageLoop thread
main: Waiting for MessageLoop thread to finish
Thread-0: Mares eat oats
Thread-0: Does eat oats
Thread-0: Little lambs eat ivy
Thread-0: A kid will eat ivy too
main: Finally!
```

➢ Exemplu: SimpleThreads https://docs.oracle.com/javase/tutorial/essential/concurrency/simple.html

Thread-ul principal creaza un al doilea thread si asteapt ca acesta sa isi termine executia.
Daca al doilea thread nu si-a terminat executia dupa o perioada fixata, thread-ul principal il intrerupe.
Thread-ul secundar este creat prin implementarea interfetei Runnable

```
private static class MessageLoop
      implements Runnable {
      public void run() {
         String importantInfo[] = {"s1", …, "s4"}
         try {
            for (int i = 0;  i < importantInfo.length;i++) {
                Thread.sleep(4000);
                 threadMessage(importantInfo[i]);
            }
         } catch (InterruptedException e) {
            threadMessage("I wasn't done!");
         }}}
```

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

➢ Exemplu: SimpleThreads (cont)

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

    long patience = 1000 * 60 * 60;
    long startTime = System.currentTimeMillis();

     threadMessage("Starting MessageLoop thread");
     Thread t = new Thread(new MessageLoop());
     t.start();

     threadMessage("Waiting for MessageLoop thread to finish");
     while (t.isAlive()) {
                              threadMessage("Still waiting...");
                              t.join(1000);   // de facut modificarea t.join() si vazut ca asteapta sa scrie
                              if (((System.currentTimeMillis() - startTime) > patience)  && t.isAlive()) {
                                                        threadMessage("Tired of waiting!");
                                                        t.interrupt();
                                                        t.join(); }}
  threadMessage("Finally!");}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

Rulare pentru patience=8000, Thread.sleep(4000)

```
C:\myjava\tutoracle>java SimpleThreads
main: Starting MessageLoop thread
main: Waiting for MessageLoop thread to finish
main: Still waiting...
main: Still waiting...
main: Still waiting...
main: Still waiting...
Thread-0: Mares eat oats
main: Still waiting...
main: Still waiting...
main: Still waiting...
main: Still waiting...
Thread-0: Does eat oats
main: Tired of waiting!
Thread-0: I wasn't done!
main: Finally!
```

Variabile locale thread-ului: variabile care pot fi scrise si citite numai de acelasi thread

```
private ThreadLocal<String>  myThreadLocal = new ThreadLocal<String>();
 myThreadLocal.get()
myThreadLocal.set("s")
```

```
public static class MyRunnable implements Runnable {

    private ThreadLocal<Long> threadLocal =
            new ThreadLocal<Long>();

  public void run() {
        threadLocal.set( Thread.currentThread().getId());

        try {
           Thread.sleep(2000);
        } catch (InterruptedException e) {
        }

        System.out.println("Name:" +  Thread.currentThread().getName()+" Id:"+threadLocal.get());
    } }
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

Variabile locale thread-ului

```
public class ThreadLocalEx {
  public static class MyRunnable implements Runnable {

      private ThreadLocal<Long> threadLocal =
            new ThreadLocal<Long>();

public void run() { …}
}

   public static void main(String[] args) throws InterruptedException {
      MyRunnable sharedRunnableInstance = new MyRunnable();

      Thread thread1 = new Thread(sharedRunnableInstance);
      Thread thread2 = new Thread(sharedRunnableInstance);

      thread1.start();
      thread2.start();
       thread1.join();
      thread2.join();
   }}
```

```
C:\myjava\tutoracle>java ThreadLocalEx
Name:Thread-0 Id:12
Name:Thread-1 Id:13
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

Comunicarea intre thread-uri: doua thread-uri care incrementeaza acelasi contor

```
public class ThreadInterference{
   private Integer counter = 0;

   public static void main (String[] args) {
      ThreadInterference demo = new ThreadInterference();
      Task task1 = demo.new Task();
      Thread thread1 = new Thread(task1);

      Task task2 = demo.new Task();
      Thread thread2 = new Thread(task2);

      thread1.start();
      thread2.start();
   }

 private class Task implements Runnable {…}
private void perform Task() { …}

}
```

```
   private class Task implements Runnable {


      public void run () {
         for (int i = 0; i < 5; i++) {
            performTask();
         }
      }
   }

private void performTask () {
      int temp = counter;
      counter++;
      System.out.println(Thread.currentThread()
                        .getName() + " - before: "+temp+" after:"
counter);
   }
```

https://www.logicbig.com/tutorials/core-java-tutorial/java-multi-threading/java-thread-synchronization.html

https://docs.oracle.com/javase/tutorial/essential/concurrency

```
C:\myjava\tutoracle>java ThreadInterference
Thread-1 - before:  1 after:2
Thread-0 - before:  0 after:2
Thread-1 - before: 2 after:3
Thread-0 - before: 3 after:4
Thread-1 - before: 4 after:5
Thread-1 - before: 6 after:7
Thread-0 - before: 5 after:6
Thread-1 - before: 7 after:8
Thread-0 - before: 8 after:9
Thread-0 - before: 9 after:10
```

race

incrementarea contorului!

https://docs.oracle.com/javase/tutorial/essential/concurrency

## ➢ Sincronizarea thread-urilor

- Metode sincronizate

```
private synchronized void syncMethod () {
   //codul metodei
 }
```

- Instructiuni sincronizate

```
synchronized (object reference){
   // instructiuni
 }
```

O metoda sincronizata poate fi scrisa ca bloc sincronizat:

```
private void syncMethod () {
      synchronized (this){
        //codul metodei
   }
}
```

## ➢ Mecanismul de sincronizarea thread-urilor

- Fiecare obiect are un lacat intern (*intrinsic lock*, *monitor lock*).

- Un thread are acces la obiect numai dupa ce detine (aquire) lacatul intern, pe care il elibereaza release) dupa ce a terminat de lucrat cu obiectul; atat timp cat un thread detine lacatul intern al unui obiect, orice alt thread care doreste sa faca aquire este blocat.

- Cand un thread apeleaza o metoda sincronizata se face aquire pe lacatul obiectului care detine metoda; pentru metodele statice, lactul este al obiectului *Class* asociat clasei respective.

- Lacatul este pe obiect, accesul la toate metodele sincronizate este blocat, dar accesul la metodele nesincronizate nu este blocat.

**Atentie!**

- Un thread poate face aquire pe un lacat pe care deja il detine (reentrant synchronization)
- **Thread.sleep()** nu elibereaza lacatul, thread-ul il va detine mai mult timp
- **ob.wait()** elibereaza lacatul obiectului

https://docs.oracle.com/javase/tutorial/essential/concurrency

- Metode sincronizate

```
public class ThreadInterference{
   private Integer counter = 0;

   public static void main (String[] args) {
      ThreadInterference demo = new ThreadInterference();
      Task task1 = demo.new Task();
      Thread thread1 = new Thread(task1);

      Task task2 = demo.new Task();
      Thread thread2 = new Thread(task2);

      thread1.start();
      thread2.start();
   }

 private class Task implements Runnable {…}
private  synchronized void perform Task() { …}
}
```

```
   private class Task implements Runnable {


      public void run () {
         for (int i = 0; i < 5; i++) {
            performTask();
         }
      }
   }

private synchronized void performTask () {
      int temp = counter;
      counter++;
      System.out.println(Thread.currentThread()
                        .getName() + " - before: "+temp+" after:" +
counter);
   }
```

https://www.logicbig.com/tutorials/core-java-tutorial/java-multi-threading/java-thread-synchronization.html

https://docs.oracle.com/javase/tutorial/essential/concurrency

```
C:\myjava\tutoracle>java ThreadInterference
Thread-0 - before: 0 after:1
Thread-0 - before: 1 after:2
Thread-0 - before: 2 after:3
Thread-0 - before: 3 after:4
Thread-0 - before: 4 after:5
Thread-1 - before: 5 after:6
Thread-1 - before: 6 after:7
Thread-1 - before: 7 after:8
Thread-1 - before: 8 after:9
Thread-1 - before: 9 after:10
```

Numai un singur thread poate detine lacatul obiectului la un moment dat.
Un thread detine lacatul intern al unui obiect daca:

- executa o metoda sincronizata a obiectului
- executa un bloc sincronizat de obiect
- daca obiectul este  Class, thread-ul executa o metoda static sincronizata

Metode ale obiectelor

**void wait() , wait(milisecunde)**

threadul intra in asteptare pana cand primeste notifyAll sau notify de la alt thread

**void notifyAll()**

trezeste toate threadurile care asteapta lacatul obiectului

**void notify()**

trezeste un singur thread care asteapta lacatul obiectului;
thread-ul este ales arbitrar

**ob.wait()**

- poate fi apelata de orice obiect ob
- trebuie apelata din blocuri sincronizate
- elibereaza lacatul intern al obiectului
- asteapta sa primeasca o notificare prin notify/notifyAll
- dupa ce primeste notificare re-incearca sa detina lacatul obiectului

**Thread.sleep()**

- poate fi apelata oriunde
- thread-ul curent se va opri din executie pentru perioada de timp precizata
- nu elibereaza lacatele pe care le detine

**Atentie!**

Metodele **wait(), sleep() si join()** pot arunca **InterruptedException** daca un alt thread intrerupe threadul care le executa.

https://docs.oracle.com/javase/tutorial/essential/concurrency

➢ Modelul Producator-Consumator


Doua threaduri comunica prin intermediul unui buffer (memorie partajata):

- thread-ul Producator creaza datele si le pune in buffer
- thread-ul Consumator ia datele din buffer si le prelucreaza

Probleme de coordonare:

- Producatorul si consumatorul nu vor accesa bufferul simultan.
- Producatorul nu va pune in buffer date noi daca datele din buffer nu au fost consumate
- Cele doua thread-uri se vor anunta unul pe altul cand starea buferului s-a schimbat

- Modelul Producator-Consumator


```
public class PCDrop {

   private String message;
   private boolean empty = true;

   public synchronized String take() {
    …
  return message;  }

  public synchronized String put(String message) {
    …
  }
```

implementarea buffer-ului:
accesul se face prin metode sincronizate

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

- Thread-ul producator

```
public class PCProducer implements Runnable {
   private PCDrop drop;

   public PCProducer(PCDrop drop) {
      this.drop = drop;
   }

   public void run() {
      String importantInfo[] = {   "Mares eat oats","Does eat oats","Little lambs eat ivy","A kid will eat ivy too"};
      Random random = new Random();

      for (int i = 0;  i < importantInfo.length; i++) {
                                                          drop.put(importantInfo[i]);
                                                          try {
                                                                Thread.sleep(random.nextInt(5000)
                                                             } catch (InterruptedException e) {}
                                                           }
      drop.put("DONE");
   }}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

- Thread-ul consumator

```
import java.util.Random;

public class Consumer implements Runnable {
   private PCDrop drop;

   public  Consumer(PCDrop drop) {
      this.drop = drop;
   }

   public void run() {
      Random random = new Random();
      for (String message = drop.take();  ! message.equals("DONE"); message = drop.take())
                  {
                      System.out.format("MESSAGE RECEIVED: %s%n", message);
                     try {
                           Thread.sleep(random.nextInt(5000));
                          } catch (InterruptedException e) {}
      }
   }
}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

- Modelul Producator-Consumator

- implementarea foloseste *blocuri cu garzi*
- thread-ul este suspendat pana cand
  o anume conditie este satisfacuta

```
public class PCDrop {

   private String message;
   private boolean empty = true;

   public synchronized String take() {
    …
  return message;  }

  public synchronized String put(String message) {
    …
  }
```

```
public synchronized String take() {
      while (empty) {
         try {
            wait();
         } catch (InterruptedException e) {}
      }
       empty = true;  notifyAll();
      return message;
}
```

```
public synchronized void put(String message) {
     while (!empty) {
         try {
            wait();
         } catch (InterruptedException e) {}
      }
      empty = false;
      this.message = message;
      notifyAll();
   }}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

```
public class ProducerConsumer {
   public static void main(String[] args) {

      PCDrop drop = new PCDrop();

      (new Thread(new Producer(drop))).start();

      (new Thread(new Consumer(drop))).start();
   }
}
```

```
C:\myjava\tutoracle> java producerconsumer/PCExample
MESSAGE RECEIVED: Mares eat oats
MESSAGE RECEIVED: Does eat oats
MESSAGE RECEIVED: Little lambs eat ivy
MESSAGE RECEIVED: A kid will eat ivy too
```