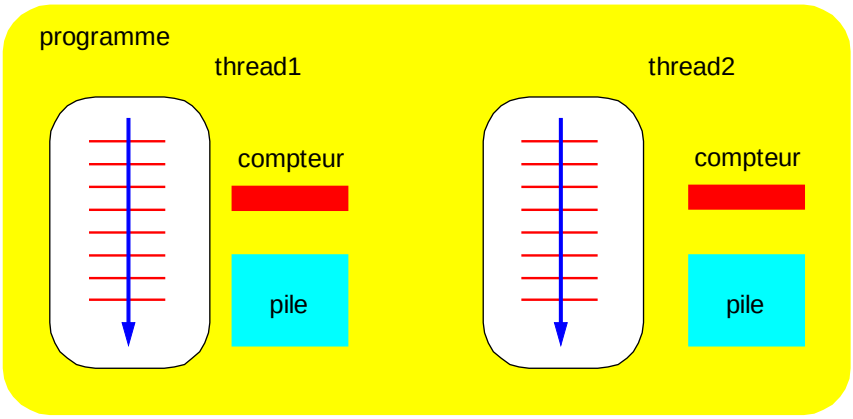
Chapter 4

Applications in Java can run as a single program or as a program with several concurrent activities called threads. Moreover, a simple program is executed as a single thread.

A thread has a beginning, a sequence of instructions and an end. A thread can not run without the framework of the program; it must be launched within a program. It can exploit the resources of a program but it uses its own program counter and its own stack.



To run a thread we must start the execution of *run()* method. This method contains the program to be executed by the thread. The *Thread* class provides the framework for the creation of a thread. The *run()* method of this class is empty and must be replaced by our *run()* method. There are two ways to implement a run () method:

• by creating a subclass of *Thread* with our *run()* method
• the implementation of our *run()* method in interface *Runnable*

Sub-classing the Thread class

An object of the Thread class is a control object to initiate and monitor the execution of a thread. When a new controller object is created, the thread starts with a call to *start()* method on that controller. The *start()* method calls the *run()* method on the object.

Example:

```
public class TwoThreadsTest {
   public static void main (String[] args) {
      SimpleThread un= new SimpleThread("UN");          // instanciation of a thread
      SimpleThread deux = new SimpleThread("DEUX");
      un.start();                                       // activation of a thread
      deux.start();
   }
}

class SimpleThread extends Thread {                    // thread – internal class
   public SimpleThread(String str) {
      super(str);
   }
```

```
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
```

The solution with a class that implements the *Runnable* interface requires the creation of an object (objects) *runnable* and then create the thread that will execute the *runnable* object. The *start()* method starts the execution.

```
class SimpleRunnable implements Runnable {  // internal class thai is runnable
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
            try {
                Thread.sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("Fini! ");
    }
}
public class TwoRunnableTest {
    public static void main (String[] args) throws Exception{
        Thread thread_un = new Thread(new SimpleRunnable(),"un");   // creation of a  thread of  runnable object
        Thread thread_deux = new Thread(new SimpleRunnable(),"deux");
        thread_un.start();              // activation of thread
        thread_deux.start();
    }
}
```

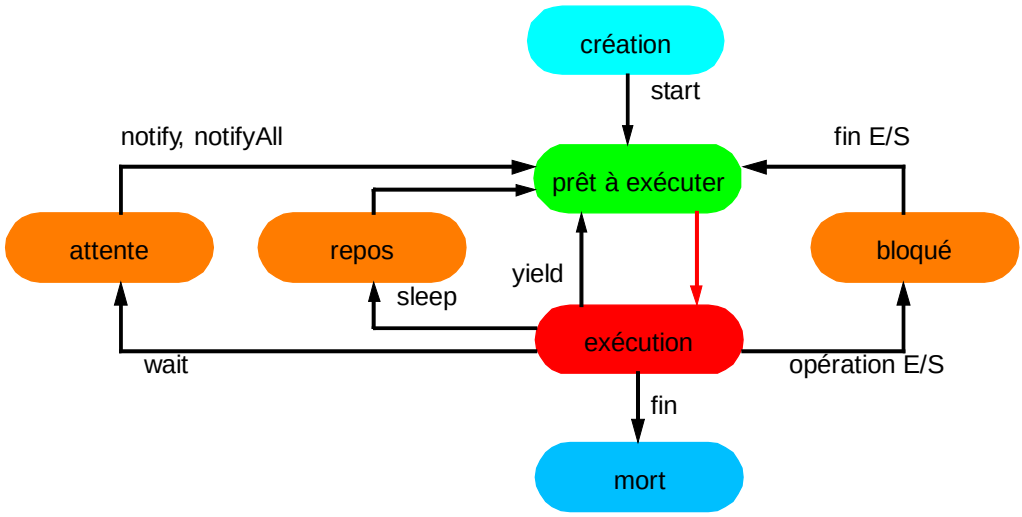
Life cycle of a thread

A thread terminates normally when its *run()* method ends. The thread is then removed from the virtual machine, but the controller object is always accessible. It does not resume execution of the thread, but it helps to know his status with the method *isAliv ()*.

The thread is actually initiated by its *start()* method, at this moment it goes to state "ready to run". The thread "ready to run" of the highest priority transits to the state "running" when the system assigns a processor to the thread. A thread enters the state "dead" when its *run()* method ends or ends with an event.

Typically, a thread enters the state "blocked" when performing an input/output query. A "blocked" thread becomes "ready to run" when the input/output operation ends. Calling the method sleep () by a thread in an "execution" causes its transition to a state of "rest". When a thread is running and makes the call wait (), it

enters into a state "waiting" for the given object on which the call wait () has been made. A "waiting" thread is released by a call notify () generated by another thread for this purpose. Each "waiting" thread goes to state "ready to run" after notification by notifyAll () call perpetrated by another thread for this purpose.



Each thread has a priority between *Thread.MIN_PRIORITY* (value 1) and *Thread.MAX_PRIORITY* (value 10). Upon activation, a thread receives priority *Thread.NOR_PRIORITY* equal to 5. Depending on the type of operating system supports threads, thread management may or may not be performed in time division mode time-sharing). In the time-sharing mode, each thread receives a brief interval of time known as the processor quota, in which the thread has access to the processor ("running"). Upon expiry of the quota the processor is allocated to another thread of same priority.

The Java VM scheduler ensures that multiple threads are running identical priorities fairly in time. Depending on their priorities, threads are organized into several queues. All threads in a queue higher priority are executed before the threads in a queue at a lower level.

Note: Threads that have the lowest priority may wait indefinitely for their turn to run (the phenomenon of famine).

The priority of a thread is changed by the method *setPriority()*; his argument must be between 1 and 10. The method getPriority () gives the current priority of a thread.

A thread can stop the execution itself (transition to state "ready to run") by the method call yield (). This method gives the opportunity to another thread of equal priority to take control of the processor.

Example:

```
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 crees");
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
                System.out.println("Threads en etat pret");
    }
}
class ThreadState extends Thread {     // classe interne
private int timeSleep;
public ThreadState(String name) {
super(name);
                timeSleep = (int) (Math.random()*5000);
                System.out.println("Nom: " + getName() + "  temps de repos: " + timeSleep);
}
public void run() {
                try {
                System.out.println(getName() + "  Mise en repos");
                Thread.sleep(timeSleep);
                } catch (InterruptedException e) {
                System.out.println(e.toString()); }
System.out.println(getName() + "  repos fini");
}
}
```

The result of execution: (interpretation?)

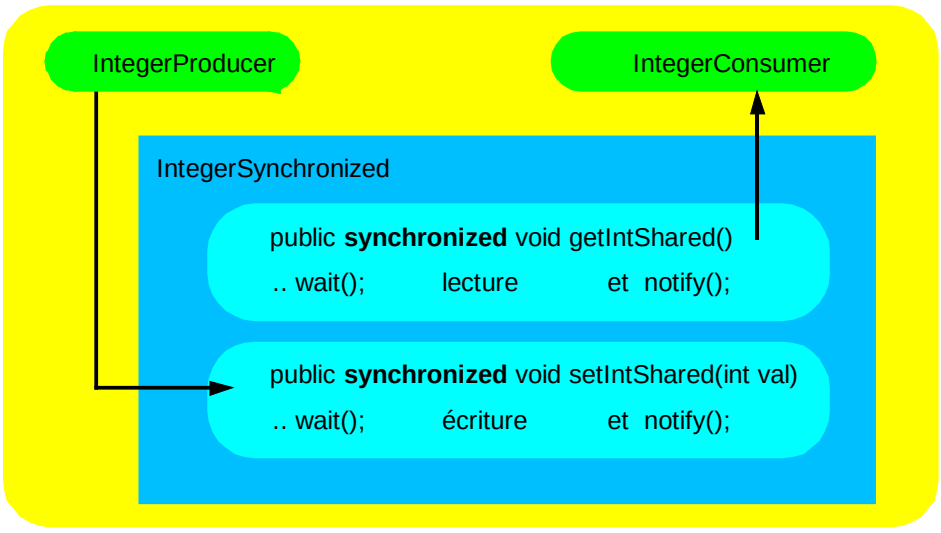
```
Nom: thread1  temps de repos: 4564
Nom: thread2  temps de repos: 322
Nom: thread3  temps de repos: 1346
Nom: thread4  temps de repos: 4013
Threads crees
Threads en etat pret
thread3  Mise en repos
thread4  Mise en repos
thread1  Mise en repos
thread2  Mise en repos
thread2  repos fini
thread3  repos fini
thread4  repos fini
thread1  repos fini
```

Synchronization

Sharing and protection of resources used by multiple threads for execution requires a synchronization mechanism. Java uses the mechanism of monitors to ensure synchronization. Each object that has synchronized methods is a monitor. The monitor is used to lock the execution of a method. If there are several methods synchronized, only one of these methods is active at a time on an object and all other items that attempt to invoke synchronized methods must wait. When a synchronized method completes its execution, the lock on the object is released and the monitor leaves the thread "ready to run" to invoke the execution of a synchronized method.

A thread that requests a synchronized method can voluntarily release the processor by calling *wait()* to wait until execution of another thread in possession of the method at that time. The thread that finishes execution of a synchronized method may report the process on hold by the *notify()*method. If a thread calls *notifyAll(),* then all threads waiting for the object becomes eligible to enter the state "ready to run". In principle, the threads that have explicitly invoked *wait()* can not continue until they receive notification *notify()* or *notifyAll()* sent by another thread.

Example: In the following example, we use the synchronization methods in an application that runs two threads: the producer thread and the consumer thread. The values (integers) generated by the producer are deposited in a box by the synchronized method *setIntShared(int val)* and retrieved by another synchronized method *getIntShared()* initiated by the consumer thread.



Program example:

```
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 cconsumer
  }
}
```

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


public class IntegerConsumer extends Thread {
private IntegerSynchronized ikeep;
public IntegerConsumer(IntegerSynchronized ipar) {
super("IntegerConsumer");
            ikeep = ipar;
}
public void run() {
            int val, sum = 0;
            do {                                                           // loop read
            try {  Thread.sleep((int) (Math.random()*3000));
            } catch (InterruptedException e) { System.out.println(e.toString()); }
             val = ikeep.getIntShared();                                   // reading in shared variable
            sum += val; }
            while (val != 10);
System.out.println(getName() + "  end of consumption - the sum is " + sum);}
}
public class IntegerSynchronized {
            private int intShared = -1;
            private boolean inscriptible = true;                   // starting for writing
public synchronized void setIntShared(int val) {       // method to synchronis the d'acces to an integer
while(!inscriptible) {                                 // waiting for validation
  try {       wait(); }                                      // waiting to be notified 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();                                              // notification of writing
}
```

```
public synchronized int getIntShared() {              // method to synchronise  the reader
while(inscriptible) {                                 // waiting for validation
try {       wait(); }                                       // waiting for notify notify()
            catch(InterruptedException e) { e.printStackTrace(); } }
inscriptible = true;
notify();
System.out.println(Thread.currentThread().getName() + " get value from intShared " + intShared);
return intShared; }                                   // writing shared variable
}
```

The result of an execution:

```
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 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
IntegerConsumer  end of consumption - the sum is 55
```

Summary

In this chapter we studied the Thread mechanism designed to launch multiple activities with concurrent execution. The threads change over time and may be in different states (ready, active, suspended ,..); at a given time only one thread can be in the state "running". The threading is based on the concept of priority. Different methods of thread management can change their priority, terminate or suspend a thread. In order to organized the synchronization and communication between threads, Java introduces the mechanism of monitor. The monitors also permit to synchronize the access to resources shared by multiple threads.

The mechanism of threads is also used for the implementation of applets that will be discussed in the next chapter.