

**Programming Languages  
Spring 2003**

**Concurrent Object-Oriented  
Programming in JAVA**

**By:  
Kaoutar El Maghraoui**

**Submitted to:  
Prof. Randolph Franklin**

**1-May-03**

# Paper Outline

1. Introduction.....	3
1. History and Motivation .....	4
2. Approaches to Concurrent Programming .....	5
3.1 Processes and Threads .....	5
3.2 The Shared Memory Model .....	7
3.3 Message Passing.....	8
3. Concurrency Constructs in Java .....	9
4.1 Multithreading in Java .....	9
4.2 Synchronization.....	12
References .....	14

# 1. Introduction

The term Concurrency refers informally to trying to do more than one thing at the same time. Likewise, in the programming paradigm, concurrent programs try to perform or execute simultaneous activities at the same time. Concurrent programming, on the other hand, deals with the communication and coordination issues that arise between both concurrent and parallel activities. Parallel programming is different in the sense that it is more concerned with techniques for mapping activities to processors. Doug Lea in his book “Concurrent Object-Oriented Programming” defines it in an operational way from a JAVA prescriptive [1]:

“A JAVA virtual machine and its underlying operating system (OS) provides mappings from apparent simultaneity to physical parallelism (via multiple CPUs), or lack thereof, by allowing independent activities to proceed in parallel when possible and desirable, and otherwise by timesharing. Concurrent programming consists of using programming constructs that are mapped in this way”

Sequential programs consist of a single active unit of execution where a sequence of instructions is being executed in a predefined order. On the other hand, concurrent programs go one step higher and represent a set of sequential processes trying to execute concurrently and compete for different resources such as CPU, memory, etc. They consist of multiple processes each one of them modeling a single active unit of execution. Concurrent programs are structured using the same programming techniques as sequential programs but with additional complexities such as:

- Non-deterministic execution: The behavior of a concurrent program is inherently nondeterministic.
- Possible data inconsistency and integrity problems: A process may be in course of modifying a value while another one reading the old value.
- Deadlock: if the concurrent processes are competing for shared resources, they may be indefinitely waiting for each other to release resources.

Therefore, more rigorous programming techniques need to be provided in order to avoid these inherent concurrency problems.

This paper is organized as follows. Section 2 gives a brief history about concurrent programming and the motivation behind it. Section 3 introduces the concepts of threads and processes and discusses the two main models in concurrency: Shared Memory and Message Massing. Section 4 discusses how Java supports concurrency.

## **1. History and Motivation**

Concurrency is not a novel concept. Much of its theoretical background was laid back the in the early 1960s with the development of independent device controllers. In the 1940s, single-user machines ceased to be a source of good economic profit since the cheapest machine cost millions of dollars. Programmers or Operators could not afford to waste the machine's idle recourse while examining the output or trying to fix a bug. For this reason, batch processes were introduced. Still this simple form of parallelism did not fully utilize the machine idle time due to I/O busy waiting. Interrupt drive I/O was then developed to make use of the lost cycles to busy waiting. These developments have laid the very first foundations of concurrency within the operating system and multiprocessor machines by the late 1960s [2].

In the late 1970s and early 1980s, computer networks have been introduced (Ethernet and Arpanet) which have given rise to distributed programming. Parallel computing has become very massive in the mid to late 1990s with the development of client-server architectures.

Concurrency was originally tailored at the system level with the advent of multi-programming and time-sharing systems. Its scope has started to widen progressively and has reached the application domain when multithreading was introduced. The advent of object-oriented programming in particular with its new approaches to encapsulation, data hiding, extensibility, modularity, and safety has altered the traditional classic synchronization approached and has fostered new research in the area of concurrent object programming. Concurrency has gained recently widespread attention and has become pervasive and central to providing more powerful programming abstractions.

There are many reasons as to why concurrent programming has become very important:

- To increase the performance of hardware (multiprocessing hardware) by running more than one processor at the same time.
- To increase the application throughput. I/O can occur while processing other tasks. It blocks only one thread of execution.
- To increase the responsiveness of the applications
- To achieve more appropriate application structures. Many applications are inherently concurrent such as servers, and graphical applications. They need to keep track of many tasks at the same time. To capture the correct logical structure for such applications, concurrency needs to be taken into account.

## **2. Approaches to Concurrent Programming**

This section describes the two main concurrent programming models: the message passing model and the shared memory model. The section starts first with defining processes and threads and the differences between the two.

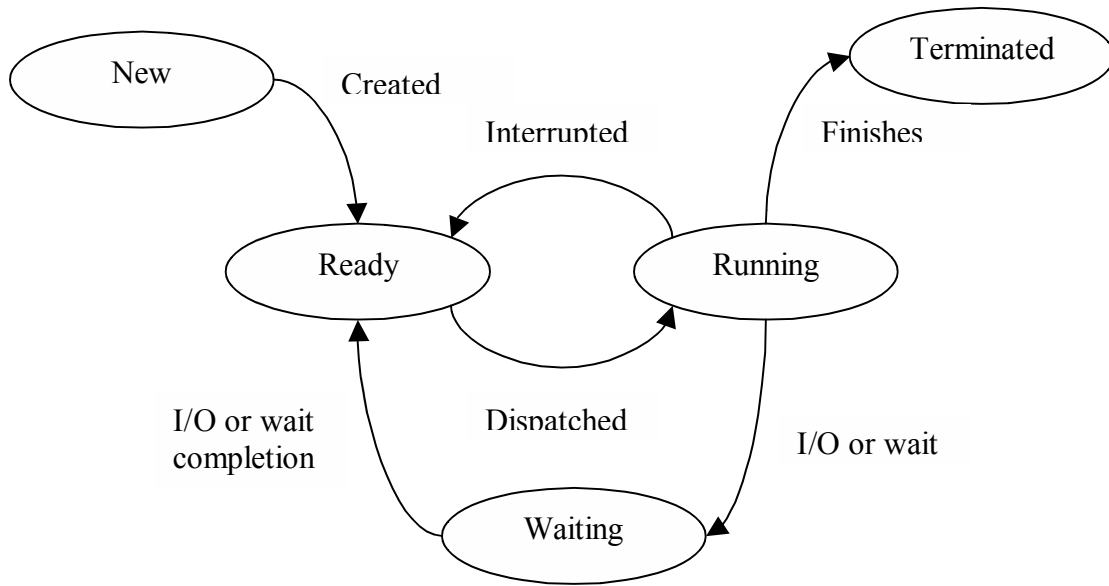
### **3.1 Processes and Threads**

Informally, a process is a program in execution. It is the execution of a sequential program where one instruction is executed at a time. From an Operating System's perspective, a process is represented by its code, data, the contents of the processor's registers, stack of execution holding temporary data (e.g. subroutine parameters, return values, and temporary variables) and state of the and the value of the program counter. A process can be in one of five possible states (Fig 1) [5]:

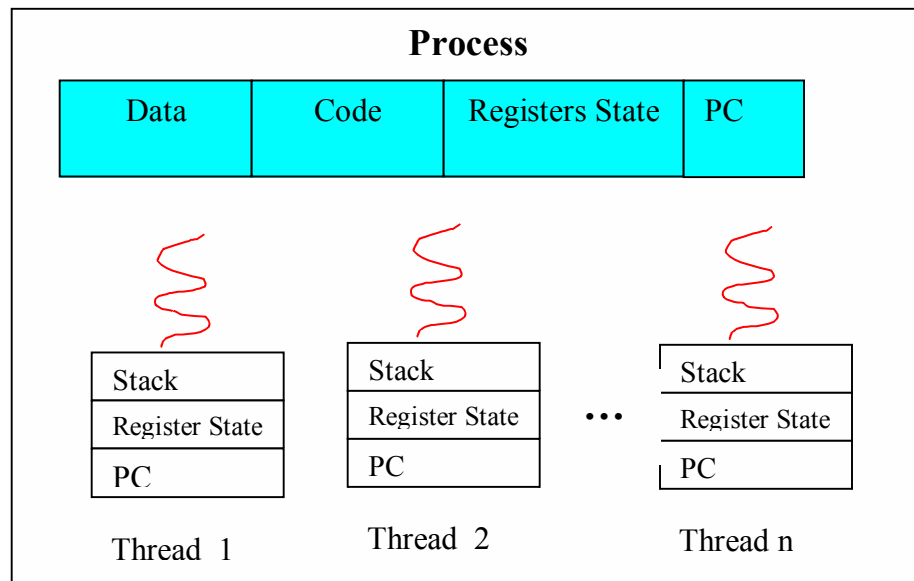
- New: when the process is first created
- Running: One of its instructions is being executed
- Waiting: The process can be waiting for some event to occur such as I/O
- Read: The process is ready but waiting to be assigned to a processor
- Terminated: the process has finished execution

A thread is the unit of execution within a process. It consists of a program counter, a register set, and a stack space. A thread is often referred to as “light-weight process” as opposed to the “heavy-weight” process. The scheduling of threads is pre-

emptive, i.e. a currently executing thread may be suspended at any time to allow another thread to run. A process supports one to many concurrently executing threads of control (Fig 2).



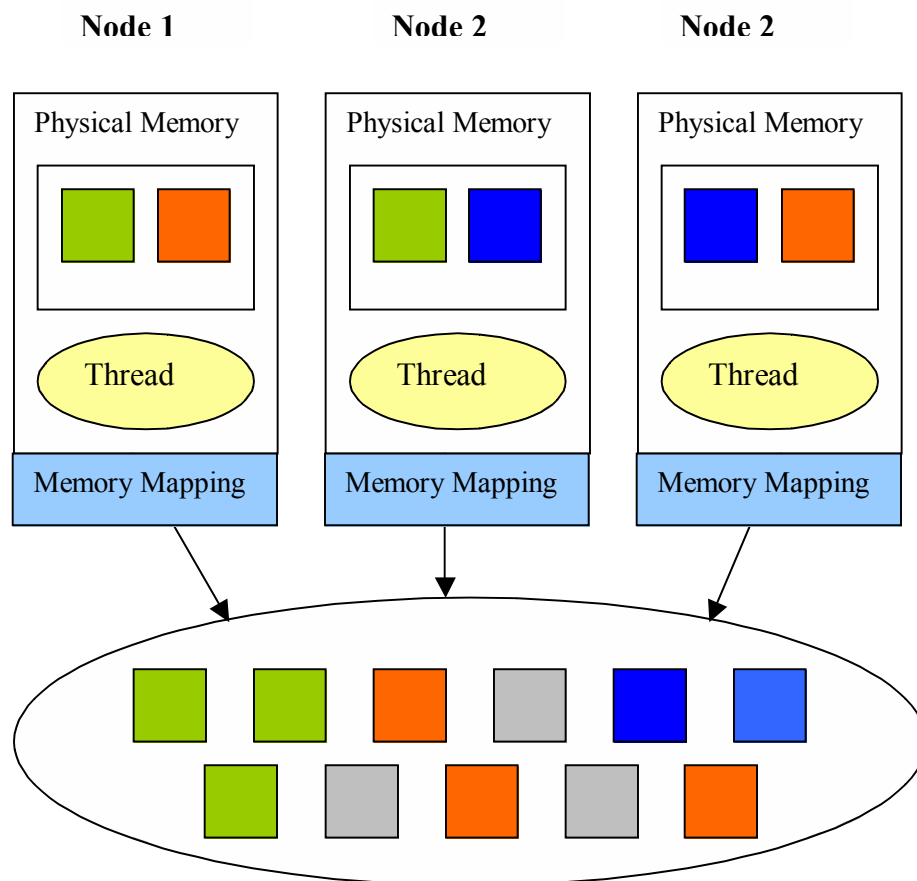
**Fig 1: Process Life Cycle**



**Fig 3: Process and threads**

### 3.2 The Shared Memory Model

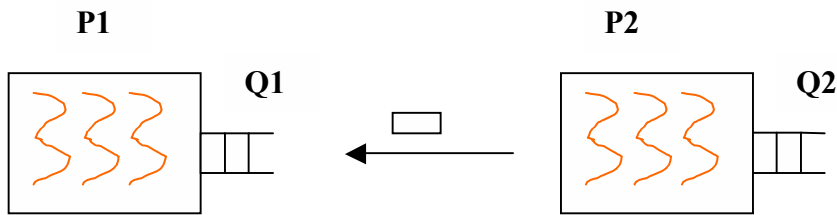
In the shared memory approach, processes execute concurrently and communicate by sharing a common pool of memory. The shared memory doesn't have to be in one physical location. It can be distributed across different nodes (Fig 1). This model is used in most multi-threaded applications where all the threads have access to the global data of a process. Developers have to be aware of critical section and data corruption problems that may arise in this model.



**Fig 3:** Distributed Shared Memory

### 3.3 Message Passing

In the message passing model, messages are used as the form of communication between the concurrent running processes. It is usually used when processes are distributed in client-server or peer-to-peer architectures. Systems that support remote procedure call adhere to this model. Primitives such as send and receive are used for inter-process communication. Each process usually maintains a message queue where message are received. Table 1 shows a comparison between the two models.



**Fig 4:** Message Massing Model

	<b>Shared Memory</b>	<b>Message Passing</b>
<b>Interface to Communication</b>	Implicit and transparent through the system bus	Explicit communication through messages
<b>Architecture Complexity</b>	Leverages conventional architecture better since existing processors can be added to the shared bus system easily.	Simpler multiprocessing architecture overall. Code should be rewritten for new platforms due to the explicit interface to communication.
<b>Convenience</b>	Serial code runs without modification	Message passing libraries needed for a wide range of platforms
<b>Protocols</b>	Protocols are embedded in the system hardware. Rarely under user control	Communication protocols are needed and they are under the user's control
<b>Performance</b>	Generally faster because it has hardware support	Protocols are generally complex to programmers which cause it to be treated as I/O for portability reasons. This can be expensive and slow

**Table 1:** Shared Memory vs. Message Passing

### 3. Concurrency Constructs in Java

This section discusses how Java supports and implements concurrency. The first section introduces multithreading, mainly how to establish and manage concurrency using Thread objects in Java. The second section discusses how to maintain consistent states of objects by preventing undesired interference between concurrent threads through synchronization constructs. The third section gives a complete example that illustrates the discussed concepts in a real application.

#### 4.1 Multithreading in Java

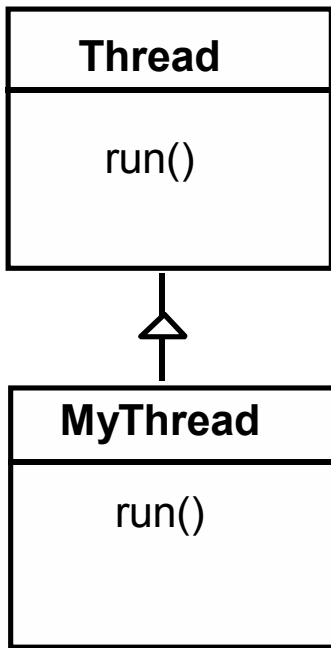
A Java thread is an object that encapsulates a single sequential thread of control. Threads may be create and deleted automatically. Java programs may consist of multiple threads running concurrently. Multithreading capabilities have been integrated very tightly within the Java language and its core packages unlike many other languages where they have been added later mainly through libraries.

There are two ways to create a java Thread [1] [4] [6]:

1. Extending the class **java.lang.Thread** and implementing a run() method that provides the block of the code that will be execute once the thread starts executing. **java.lang.Thread** is a core class that defines basic methods for querying thread properties, starting, and ending the execution of a thread. This approach is described in figure 5.
2. Implementing the **java.lang.Runnable** interface that defines a run() method.

Figure 6 shows the steps involved in using this approach in Java.

Now that the thread has been created, we can start the execution of the thread by invoking the start() method on the newly created thread. This method will call the run() method. The thread terminates when its run method terminates execution or when an uncaught exception is thrown.



**Step1:** Deriving a new class from the class Thread and implementing the run method

```

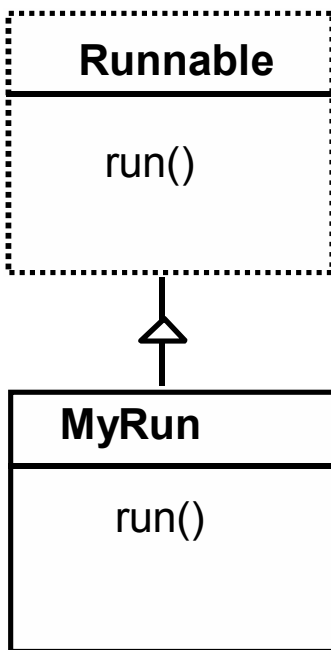
class MyThread extends Thread {
    public void run() {
        //.....
    }
}
  
```

**Step2:** Creating an instance of the class Thread.

```

Thread x = new MyThread();
  
```

**Fig 5:** Using the Thread class to create threads



**Thread**

```

Public interface Runnable {
    Public abstract void run();
}
  
```

**Step1:** Implementing the interface Runnable

```

class MyRun implements Runnable{
    Public void run(){//...}
}
  
```

**Step2:** Creating the Thread

```

Thread x = new Thread(new MyRun());
  
```

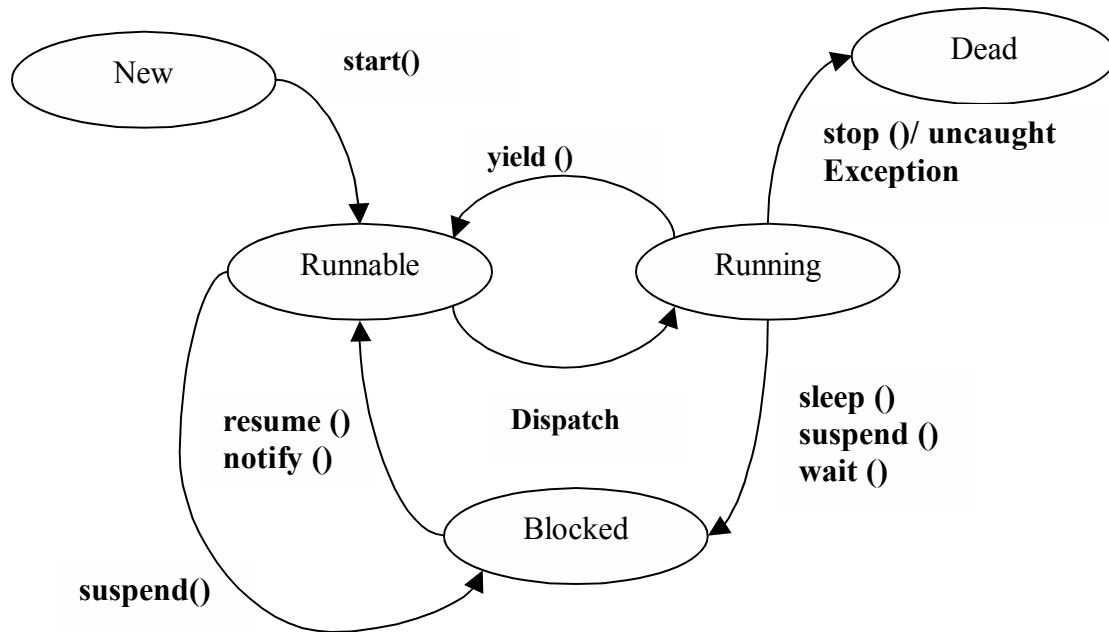
**Fig 6:** Using the Runnable Interface to create threads

Once a thread has been started, it becomes alive and it can be in one of these states (7):

- New: when the thread is newly created
- Running: A thread is in the running state when its code starts execution.
- Runnable: Once the run() method is invoked the thread becomes Runnable. A Runnable thread is not necessary in the running state. A running thread is equivalent to a ready process. It becomes running once it has been assigned the CPU by the OS scheduler.
- Blocked: A thread becomes blocked as a result of one of these actions:
  1. The sleep() method has been invoked on this thread.
  2. The thread performs an I/O operation and blocks waiting for the completion of the I/O operation.
  3. The thread invokes the wait() method.
  4. The thread tries to acquire a lock that is already obtained by another thread. This will be discussed in more details in the section 4.2
  5. The method suspend() is called. This method has been deprecated in the recent releases of Java.
- Dead: The thread terminates when one of these two events happen:
  - Normal termination of the code in the run() method
  - Abnormal termination due to an uncaught exception

A thread can be inspected for aliveness using the method isAlive().

Every thread in java has a priority. By default every thread inherits the priority of its parent thread. The user can set alter this default priority using the setPriority() method. The value of the priority ranges for MIN\_PRIORITY (value of 1) to MAX\_PRIORITY (value of 10). The default priority is NORM\_PRIORITY (value of 5).



**Fig 7:** A Java Thread Life Cycle

## 4.2 Synchronization

When multiple concurrent threads access some shared data, destructive updates can happen and this will violate the consistency of the data. This problem is referred to as interference [3]. Interference is caused by the interleaving of read and write actions on a shared block of data. Mutual exclusion provides a solution to this problem. In Java mutual exclusion is supported through the use of the keyword `synchronized`

There are two forms of using the `synchronized` keyword: synchronizing an object or synchronizing a method as shown in figure 8.

Once a method is tagged with the keyword `synchronized`, it is guaranteed that the method will finish execution before any other thread can execute any `synchronized` method on the same object. On entry to a `synchronized` method a lock is obtained by the running thread and released on exit. If a thread tried to obtain a lock while another thread is executing a `synchronized` method on the same object, it will be blocked until it can acquire the lock. A thread can acquire the locks of multiple objects at the same time. However, the lock of an object can be obtained by only one thread at a time.

*Synchronizing a method*

```
synchronized void myMethod{  
    // body  
}
```

*Synchronizing an object*

```
Synchronized (object) {  
    // statements  
}
```

**Fig 8:** The synchronized keyword in Java

In addition to locks, every object in Java has also a wait set that is maintained by the JVM through the methods notify, notifyAll, and ThreadInterrupt. Each wait set keeps track of the threads that were blocked by the wait method and who are waiting to be notified. The actions of these methods are described in the table 2:

<b>Method</b>	<b>Action</b>
wait	The thread is blocked if it was not interrupted before. Otherwise an Interrupt Exception is thrown. The thread is then placed in a wait set. All locks owned by this thread are released.
notify	An arbitrary thread is removed from the wait set and made Runnable. Then the thread is resumed from where it left off before the wait invocation.
notifyAll	Its action is similar to notify. The only difference is that all threads in the wait set are notified.
Interrupt	If the thread is suspended in a wait, the same action as notify happens except that after re-acquiring the lock the thread's interruption status is set to false and an InterruptedException is thrown.

## References

- [1] Doug Lea, “Concurrent Programming in Java™”, Second Edition, Addison-Wesley, 1999.
- [2] Michael L. Scott, “Programming Language Pragmatics”, Library of Congress, 2000.
- [3] Rekesh John, “Issues in Concurrent Programming”, White Paper,  
<http://www.calsoft.co.in/techcenter/windows/concurrent.html>
- [4] Cay S. Horstmann, Gray Cornell, “Core JAVA”, Volume II-Advanced Features, Prentice Hall, 2000.
- [5] Silberschatz Galvin, “Operating System Concepts”, Fifth Edition, Addison Wesley, 1998.
- [6] David Flanagan, “JAVA Examples in a Nutshell”, Second Edition, O’Reilly, 2000.