

Programming with threads

Topics

1. Introduction
2. Threads in Java
3. Concurrent execution
4. Communication between threads
5. Synchronization
6. Monitors and condition synchronization
7. Semaphores
8. Properties of activity
9. Design and implementation of thread-safe classes.

Additional literature

[Ca] Campione & Walrath: The Java Tutorial. Second Edition. Object-Oriented Programming for the Internet. Add.-Wesl. 1999. 0-201-31007-4.
(Has been published in a 3. edition: 0-201-70393-9).
CD-version: Trail: Essential Java Classes: Doing Two or More Tasks at Once: Threads.

Also see: The Java Tutorials, <http://download.oracle.com/javase/tutorial/>,
Last accessed April 2011.

1 Introduction

1.1 Parallel system

A *parallel system* handles *separate activities*, which are ongoing concurrently (*at the same time/simultaneously*).

Two activities are parallel if they at a given time are ongoing somewhere between their starting point and ending point.

In some cases the separate activities will
cooperate
on solving a given problem.

In other cases the separate activities will
compete
about joint system resources which they have to share.

In order to get an activity executed, a program module which can be carried out on a processor, is required.

Generally a *process* can be characterised as an *active object*, which carry out the activity on a processor with the program module.

A *parallel system* will therefore consist of program modules which are being executed concurrently by several processes (at one or several processors).

1.2 Processes and threads

In many operating systems and languages you distinguish between *processes* and *threads*.

The *process* term is the "heavy" term.

An important purpose of *processes* is to *protect their adjoining programs from each other*, as the processes cannot in general be assumed to be cooperative – rather the contrary.

A *process* can therefore be perceived as a "*shell*", that surrounds and protects an activity, - typically the performance of an application. To the process one can tie a number of resources, which are thus protected from misuse by other processes. Processes are carried out in parallel. The activity in one process is handled by one or several threads.

As an important part of the protection of its activity the process is awarded
a separate address room.

In some literature a *thread* is also called a *lightweight process*.

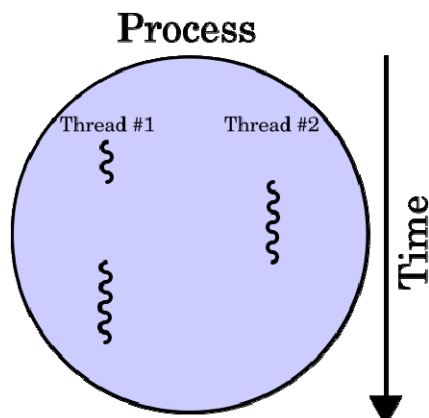


Figure 1. A process with two threads

In many contexts, such as control and regulation, one need (lightweight) processes which must be carried out simultaneously (parallel), and often *have to cooperate*. Such (lightweight) processes shall therefore *not be protected from each other*.

A *thread* (lightweight process) is an *independent sequential activity within a process*.

A process can contain several parallel *threads*, which share the process resources and are *not protected from each other*.

Thus threads have

shared address room, i.e. the process address room.

This address room includes among others

global variables and the heap.

Furthermore each thread has

its own separate stack.

This is shown in the below figure which illustrates a process with three threads.

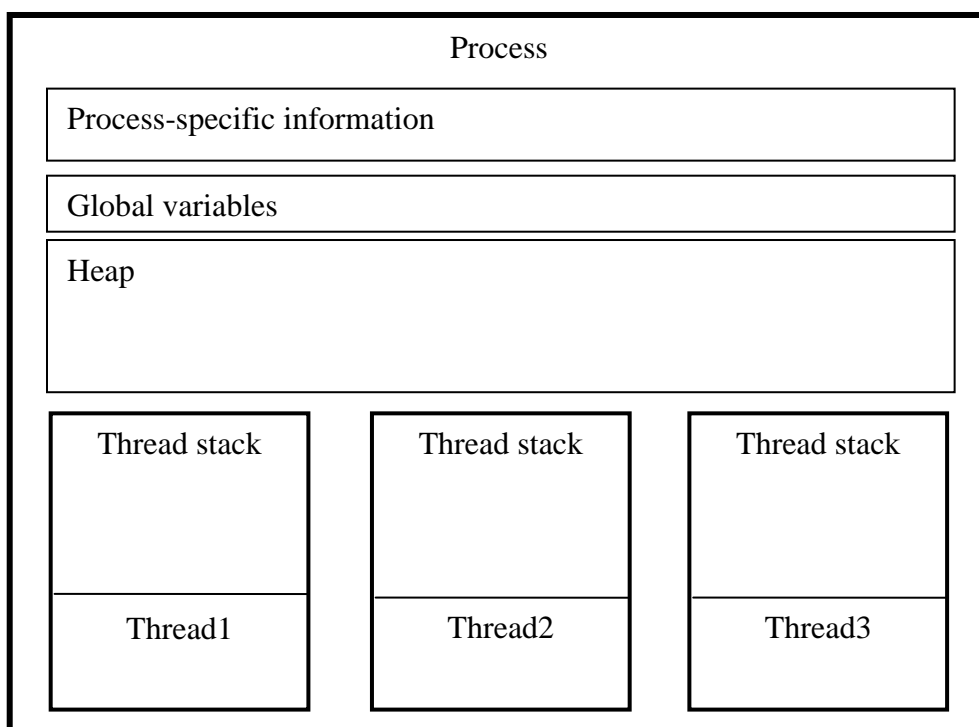


Figure 2. A process with three threads

2 Threads in Java

Also see [Ca].

In the Java package `java.lang` you can find

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

```
public interface Runnable
```

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`. This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, Runnable is implemented by class Thread. Being active simply means that a thread has been started and has not yet been stopped.

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

A thread is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

There are two ways to create a new thread of execution.

2.1 Thread in the first way

One is to declare a class to be a subclass of Thread. This subclass should override the `run` method of class Thread. An instance of the subclass can then be allocated and started.

class Thread carries out instruction from its method `run()`. The actual code which is being carried out depends on the implementation of `run()` in a derived class which appears from the following structural diagram:

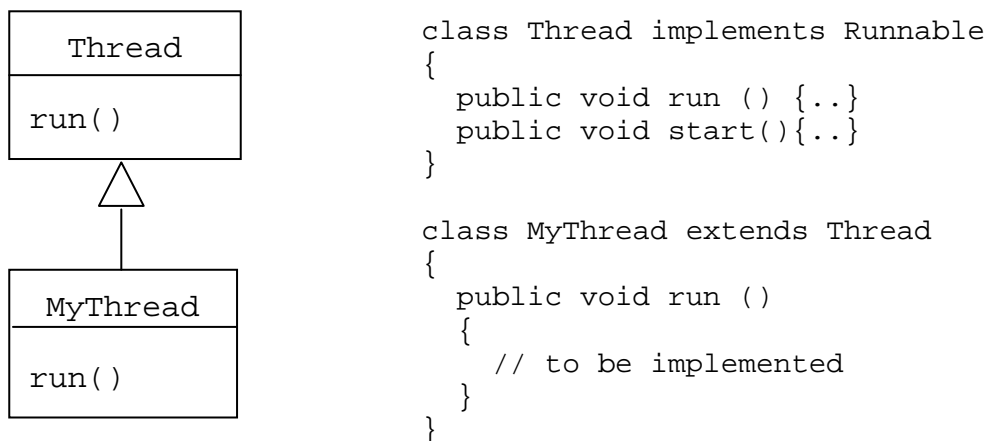


Figure 3. Implementation of `run()` by inheritance.

2.2 Thread in the second way

The other way to create a thread is to declare a class that implements the Runnable interface. That class then implements the `run` method. An instance of the class can then be allocated, passed as an argument when creating a Thread object, and started.

As Java does not have multiple inheritance it is sometimes necessary to implement the `run()`-method in a class derived from interface Runnable.

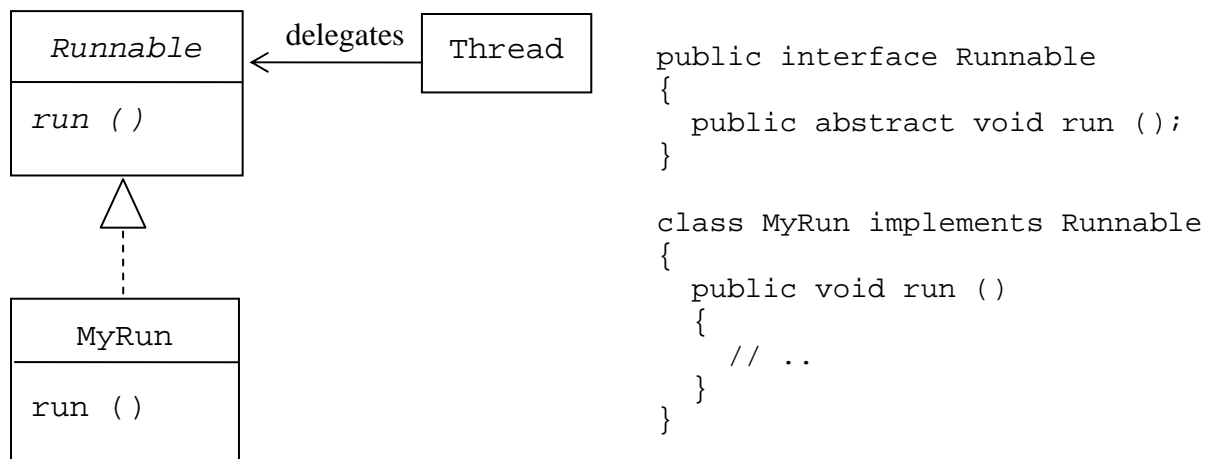


Figure 4 Implementation of `run()` by interface `Runnable`.

Notice that class `MyRun` is not a thread class. E.g. it has no `start` method.

2.3 Creation of and running a thread object

In the two cases a thread object has to be created, after which the thread can be started.

In the first case it is carried out as follows:

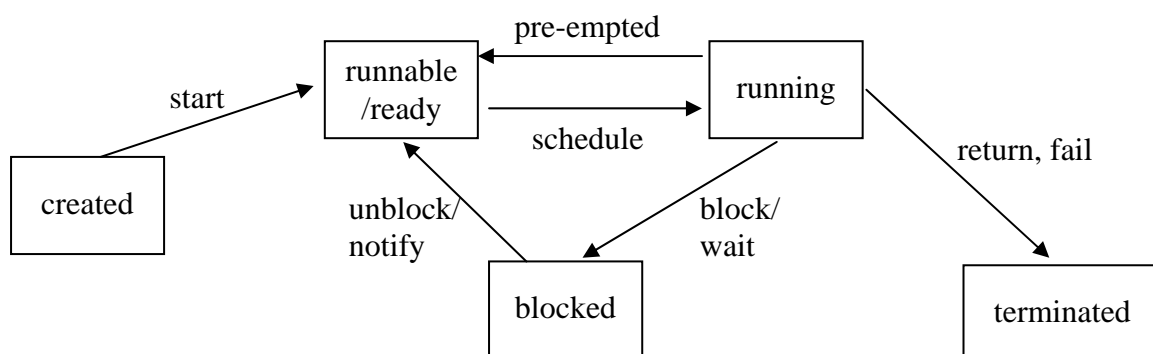
```
Thread th1 = new MyThread ();
th1.start ();
```

In the second case a thread object shall also be created but this thread delegates the running to an object of type `Runnable`, which has an implemented `run` method. In our case it is a `MyRun` object. It is carried out as follows:

```
Thread th2 = new Thread (new MyRun ());
th2.start ();
```

Notice, that the `run` method shall not be called in any of the two cases. This is carried out inside the `start` method.

2.4 Life cycle of a thread



Also see [Ca].

2.5 The methods of class Thread

See the Java-documentation. Some few methods are shown here:

```
void start()
void run()
static void sleep(long millis)
boolean isAlive()
public final void join()           waits for the thread to die
static void yield()               causes thread to pause
```

Furthermore some methods which *you must not use* (*deprecated*):

```
void stop()
void resume()
void suspend()
```

When a company like Sun (and without doubt also many other companies) can make such grave mistakes, which mistakes might we, who are only mortals, then make? Among others it shows how careful you must be when you make parallel programs with threads that communicate mutually.

Also see the Java-documentation for e.g. `stop()`, which refers to:

Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?

Why is Thread.stop deprecated? Because it is inherently unsafe.

Why are Thread.suspend and Thread.resume deprecated? Thread.suspend is inherently deadlock-prone.

Sun has been criticized from numerous sides, among others by Per Brinch Hansen, who argues that its handling of communication between threads is on a too "low level".

(Per Brinch Hansen: Java's Insecure Parallelism, 1999. It can be found on the Net.)

3 Concurrent execution

3.1 Programs with several threads that use the same object (shared object)

Example 3.1.1: A thread which updates a counter.

```
public class Counter
{
    private long value = 0;

    public void inc () {
        value++;
    }

    public long value () {
        return value;
    }
}

public class Thread0 extends Thread
{
    private Counter counter;
    private final int N = 500*1000*1000;

    public Thread0 (String name, Counter counter )
    {
        super (name);
        this.counter = counter;
    }

    public void run () {
        for (int i = 0; i < N; i++)
            counter.inc ();
        System.out.println (getName () + " finished: Counter.value = " +
                             counter.value());
    }
}

public class TestThread0
{
    public static void main (String[] args)
    {
        Counter counter = new Counter ();
        Thread th0 = new Thread0 ("Thread0", counter);
        th0.start ();
    }
}
```

Example 3.1.2: Two threads which update the *same* counter.

The above program is extended so that main contains two threads that use the *same* counter:

```
Counter counter = new Counter ();
Thread th01 = new Thread0 ("Thread01", counter);
Thread th02 = new Thread0 ("Thread02", counter);

th01.start ();
th02.start ();
```

What does the program print? Which value does the counter have at the end?

3.2 Race Condition

The above example shows a situation where two threads read and write to the same variable, namely the `counter`'s value, and where the final result is not predictable. The result depends on how the two threads incidentally are working.

Such a situation is called *race condition*, because the result depends on which thread wins the race of get access to the shared variable.

3.3 Race Condition in Example 3.1.2

To find out why race condition occurs when threads increment the same counter, we will look at how the `inc` method is translated to Java bytecode.

We use the `javap` command in a command prompt:

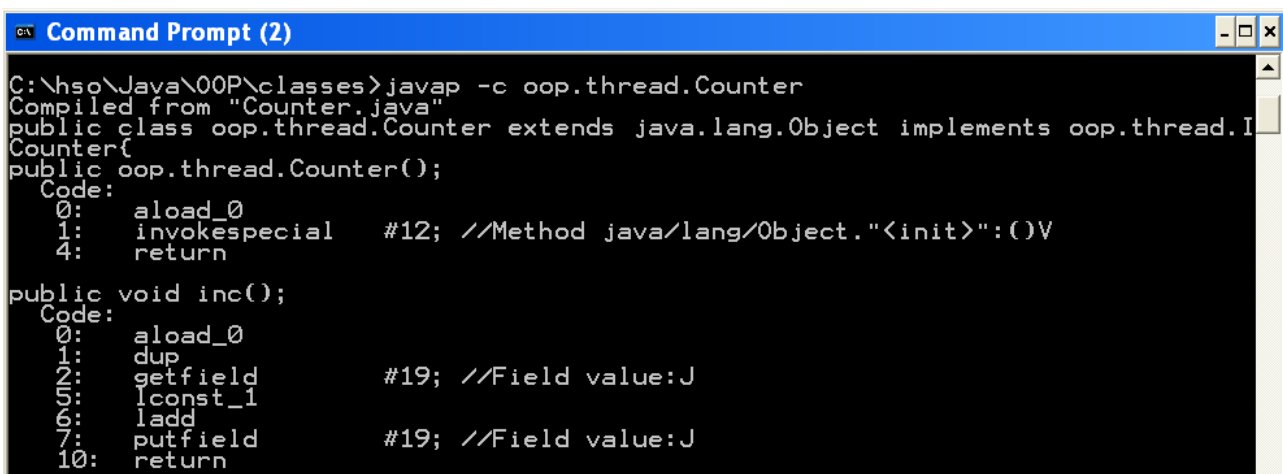
```
cd C:\hso\Java\OOP\classes
javap -c oop.thread.Counter
```

javap - The Java Class File Disassembler

The **javap** command disassembles a class file.

Option **-c**

Prints out disassembled code, i.e., the instructions that comprise the Java bytecodes, for each of the methods in the class.



```
Command Prompt (2)
C:\hso\Java\OOP\classes>javap -c oop.thread.Counter
Compiled from "Counter.java"
public class oop.thread.Counter extends java.lang.Object implements oop.thread.I
Counter{
public oop.thread.Counter();
  Code:
    0:   aload_0
    1:   invokespecial   #12; //Method java/lang/Object."<init>":()V
    4:   return

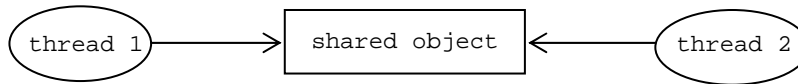
public void inc();
  Code:
    0:   aload_0
    1:   dup
    2:   getfield        #19; //Field value:J
    5:   lconst_1
    6:   ladd
    7:   putfield        #19; //Field value:J
   10:   return
```

The disassembled `inc` code is translated to seven bytecode instructions.

Knowing this, it should be possible to explain why race condition occurs in the example.

4 Communication between threads

Communication between threads is carried out by shared objects.



If a thread uses shared objects, these have to be available, either as global objects, or the thread has to get a reference to the shared object so that the `run`-method of the thread has access to the object.

This can be carried out with the following template:

```
class Shared
{
    .
    public void method1 ()
    {
    }

    public void method2 ()
    {
    }
}

class MyRun1 implements Runnable
{
    private Shared sharedObj;

    public MyRun1 (Shared s)
    {
        sharedObj = s;
    }

    public void run ()
    {
        ..
        sharedObj.method1 ();
        ..
    }
}

class MyRun2 implements Runnable
{
    // analogue
    private Shared sharedObj;

    public MyRun2 (Shared s)
    {
        sharedObj = s;
    }

    public void run ()
    {
        // here something else is
        // carried out on sharedObj:
        ..
        sharedObj.method2 ();
        ..
    }
}
```

The creating and initializing class contains the following:

```
Shared x = new Shared ();          // create shared object

Thread t1 = new Thread (new MyRun1 (x));
Thread t2 = new Thread (new MyRun2 (x));
t1.start(); t2.start();
```

Of course, it is also possible to implement the above with `Thread`-classes instead of `Runnable`-classes.

How would the implementation then look like?

4.1 Examples

See the Counter example 3.1.1 and my Thread exercises 1 and 2.

5 Synchronization

5.1 Synchronization between threads

In programs with parallel threads there will often be a need to synchronize them in relation to each other. Synchronization is

any kind of limitation of the free running of the threads in relation to each other.

Synchronization is especially necessary when using shared resources. These are typically shared variables encapsulated in objects.

5.2 Mutual exclusion

Thus, when synchronizing we have to use another procedure to ensure that the threads coordinate their activities so that a thread for example cannot change data while another thread is working with these data.

Therefore, we must be able to make a *mutual exclusion* which ensures that only one thread at a time can carry out a certain sequence of statements. This *sequence of statements must be carried out as an indivisible operation*.

5.3 Critical section

A sequence of statements which is carried out as an indivisible operation is called a *critical section*.

A template for implementing a critical section looks like the following:

```
Before Critical Section
Enter Critical Section
  Critical Section
Leave Critical Section
After Critical Section
```

5.4 Critical region

If we have concurrent threads, each using a critical section, these critical sections together make a *critical region*.

The characteristic of a critical region is that ***only one thread at a time is allowed to be in the critical region***.

Thus, the *threads* must be synchronized so that they *are never in the critical region at the same time*. *If a thread is in its critical section at a time where another thread reaches its critical section, the latter thread must wait until the first thread leaves its critical section.*

Notice, that it *does not mean* that a thread cannot be interrupted/suspended when it is in its critical section. It can, but a new thread cannot get access to its critical section until the first thread leaves its critical section.

When implementing critical sections, the above condition is not the only condition that must be fulfilled.

[Ta], p. 35 mentions *four conditions which have to be fulfilled to get a good solution*:

1. Mutual exclusion: Two threads may not be in their critical sections at the same time.
2. Independence: No assumptions may be made about the performance speeds of the threads

or about the number of CPUs.

3. Kindness: No thread running outside its critical section may block other threads.
4. Resolution: No thread should have to wait forever to enter its critical section.

5.5 Mutual exclusion in Java

Also see [Ca]: Locking an Object.

Notice that *critical section* is directly syntactically supported in Java with the keyword *synchronized*.

A critical section belonging to the critical region of an object can be established using *synchronized* in two ways:

At the method level:

```
public synchronized void method (...)  
{  
    // critical section  
}
```

At the block level in a method:

```
public void method (...)  
{  
    // before critical section  
    synchronized (this)  
    {  
        // critical section  
    }  
    // after critical section  
}
```

All the critical sections together constitute the critical region of the object.

The locking mechanism (mutex) with the belonging waiting queue is inherited from `class Object`, i.e. any object has its own locking mechanism (mutex) with a belonging waiting queue. The interested reader can find more on this subject in [Le], chap 6, e.g. from p. 104: "Now let's look at how Java implements mutual exclusion", to p. 106.

5.6 Examples

Example 5.6.1

The counter from example 3.1.1 is rewritten with *synchronized*, making all its methods appear as critical sections:

```
class Counter2  
{  
    private long value;  
  
    public synchronized void inc () {  
        value++;  
    }  
  
    public synchronized long value () {  
        return value;  
    }  
}
```

Implement the entire program and test it with several threads. Also notice that it will take longer time to run it when synchronized methods are used.

6 Monitors and conditioned synchronization

The monitor term was already introduced in the early 70ies by Hoare and Brinch Hansen (Brinch Hansen is Danish). In the 80ies the monitor term was not much in focus but from the mid 90ies it has returned to favour, after among others *Java* (and also Ada95 and POSIX) has selected the monitor term as the basal synchronization mechanism.

A *monitor* is a program module which *encapsulates data and methods allowing only one thread to be active within the monitor at a time*, i.e. only one thread at a time can carry out one of the methods of the monitor.

In *Java* a monitor is therefore implemented as a class where all data attributes are declared private and all methods are declared synchronized:

```
class Monitor
{
    private data_attributes;

    public synchronized void method () {
        // critical section with indivisible access to the class attributes
    }
}
```

6.1 Conditional synchronization

In examples such as *producer/consumer with a bounded buffer*, the monitor requires a bounded buffer to be able to suspend a thread until a certain event takes place (a certain condition has been fulfilled). It shall also be possible to inform the suspended threads that a certain event has taken place.

To provide a mechanism for *conditional synchronization*, a monitor has to contain *condition variables* with the two operations `wait()` and `signal()`.

A thread waiting for a condition variable (`wait()` is called), is suspended and releases the lock of the monitor allowing other threads to get access to the monitor.

A thread sending a signal to a condition variable (`signal()` is called), wakes up a thread from the set of threads that are suspended on this condition variable. If no thread is suspended, `signal()` has no effect. A thread that is waked up is placed in the ready-queue of threads (runnable, cf. 2.4). Here it waits for permission to continue carrying out its monitor-method. When the thread again continues to carry out its monitor-method, the monitor is locked first.

A monitor condition variable can be implemented in different ways (depending on the programming language):

<i>signal and exit</i>	i.e. after signal the signal thread exits the monitor immediately
<i>signal and wait</i>	i.e. after signal the signal thread is suspended; the thread that has received the signal is started in the monitor; when it has finished the signal thread continues its performance inside the monitor
<i>signal and continue</i>	the thread that has received the signal waits in ready-queue until the signal thread exits the monitor after which the waiting signal thread can get access to monitor, - in competition with other threads that are waiting for access, too.

Java monitors use signal and continue.

In Java, the `wait()` and `signal()` operations are inherited from class `Object` and look as follows:

```
public final void wait() throws InterruptedException
public final void notify()      // signal to one waiting thread
public final void notifyAll()   // signal to all waiting threads
```

These three methods manipulate the condition variable that belongs to the object. The *condition variable* of an object is always tied to the critical region of the object which is implemented by the critical sections of the methods with `synchronized`. Therefore the three methods can only be used within the critical sections of the methods.

See [Ca]: Using the `notifyAll` and `wait` Methods, and Java's online documentation.

Due to the signal-and-continue strategy there is no guarantee that a thread which is suspended on its condition variable by `wait()` will get immediate access to the monitor when it receives a signal. E.g. another thread could have got access to the monitor first and changed the condition that caused a signal.

Therefore *the following implementation does not work* (using `if`).

```
public synchronized void method() throws InterruptedException
{
    if (! condition) // will not work
        wait();

    // modify monitor data attributes
    notifyAll(); // or: notify();
}
```

But the following *implementation is correct*, also see [Ca]: Using the `notifyAll` and `wait` Methods.

```
public synchronized void method() throws InterruptedException
{
    while (! condition)
        wait();

    // modify monitor data attributes
    notifyAll(); // or: notify();
}
```

6.2 Examples

6.2.1 Bounded counter.

Used to control the arrival to and departure of cars from a parking space with limited capacity. Program in connection with [Ma], pp. 85-86.

6.2.2 Bounded buffer.

The example I have demonstrated is from [Ma], pp. 94-97.

7 Semaphores (only to be read)

Semaphores as a synchronization mechanism was first suggested by Dijkstra in 1965.

Semaphore: sign carrier; i.e. signal post, traffic light.

Also see Wikipedia: <http://en.wikipedia.org/wiki/Semaphore>

A semaphore has two operations:

down and up or wait and signal or red and green.

It must be possible to carry out these operations indivisibly, i.e. as atomic operations.

They are defined by:

```
down(s): while (s ≤ 0)
           wait;
           s--;

up(s):    s++;
```

Semaphores are used in two variants:

Binary semaphore

Counting semaphore

A binary semaphore corresponds to a traffic light (red and green) or a lock (lock and unlock), which is used for protection of a critical section:

```
Semaphore s = new Semaphore (1); // the initial value 1 corresponds to
                                   // green/unlock

s.wait ();
critical section
s.signal ();
```

A counting semaphore can count the number of free spaces (e.g. in a buffer/parking space):

```
Semaphore notFull = new Semaphore (N); // The initial value N corresponds
                                         // to the number of free spaces
```

Or the number of objects/cars in the buffer/parking space:

```
Semaphore notEmpty = new Semaphore (0); // the initial value 0 corresponds
                                         // to no objects in the buffer
```

Semaphores are low-level constructions, which among others are used for the implementation of monitors.

Actually monitors were introduced in 1972 to avoid synchronizing errors in programs so they could replace the use of semaphores. Therefore it cannot be recommended to use semaphores if it can be avoided.

Implementation of semaphores in Java, see [Ma] p. 91.

8 Activity properties

8.1 Deadlock

Deadlock can be defined as follows:

Some threads are deadlocked if any of the threads wait for an event that only one of the other waiting threads can provoke.

As all threads are waiting, none of them will ever provoke an event which can wake up one of the threads. All threads will therefore wait forever.

In most cases the event, which each thread awaits, will be released from a resource owned by one of the other threads.

Of course deadlock should be avoided.

[Ta] chap 6: Deadlocks, gives a detailed description of deadlock, deadlock detection and recovery, deadlock avoidance, and deadlock prevention.

8.1.1 Example: Dining Philosophers

8.2 Safety

Safety means that *nothing must go wrong during the execution*, i.e. the program must not come in a wrong state.

Example: [Ma]: Single-Lane Bridge Problem.

8.3 Liveness

Liveness means that *something good will happen*, i.e. there is *progress* in the execution.

The opposite of progression is *starvation*, which describes a situation where a certain action will never be carried out.

Example: [Ma]: Single-Lane Bridge Problem.

9 Design and implementation of thread-safe classes

9.1 Flow Patterns

Doug Lea [Lea 1] uses *Flow Patterns* as a simple but very illustrative means of description, illustrating how some objects flow through a system of cooperating objects of which some are active and some are passive.

The *active objects* are threads.

The *passive objects* are typically *counters, buffers, or the like*.

The objects that flow through the system are called *representation objects* and can represent a variety of things such as all kinds of products, temperatures, pressure, certain events etc.

An example of a system which is described with Flow Pattern is a production company with production lines where products of some kind are produced.

Another example is the luggage check-in in an airport where the luggage is checked in, weighed, applied with flight and destination labels and put on a conveyor. You can finish the example yourself and describe what happens with the luggage.

In [Lea 2] Doug Lea now calls it *flow network* instead of flow pattern.

9.2 Flow-types

There are two basal kinds of flow:

push-based flow and *pull-based flow*.

These can be illustrated as follows; we call the adjoining operations `put` and `take`, respectively:

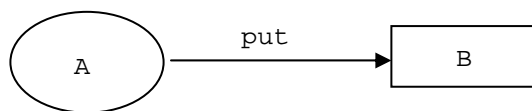


Figure 9.2.1 Push-based flow



Figure 9.2.2 Pull-based flow

A flow pattern for a bounded buffer can be illustrated as follows:

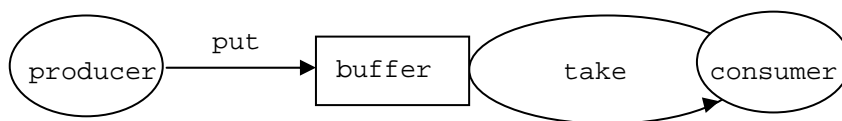


Figure 9.2.3 Put-take buffer

9.3 Design of thread-safe classes

As classes with synchronization are both much more difficult to construct without errors and even more difficult to test for errors, a couple of good and simple design principles will improve the construction of such classes.

We will use two simple design patterns of which the basic idea is to

Separate the implementation of the regular functionality of the class from the implementation of the synchronization.

In reality this is a divide-solve-and-combine problem solving strategy.

The division of the problem can be carried out in two ways:

by sub-classing
by using the adapter-pattern.

9.3.1 Implementation of a bounded buffer by sub-classing

A bounded buffer is a queue which has been made thread-safe. The queue has a fixed size.

By sub-classing we separate the implementation of the queue itself from the implementation of the thread-safe bounded buffer.


```
class Queue
{
    private int qSize;
    private int count;
    private int front, rear;
    private Object [] queue;

    public Queue () {...}

    public void put (Object x) {...}
    public Object take () {...}
    public int count () {...}
    public int capacity () {...}

    public boolean isEmpty () {...}
    public boolean isFull () {...}
}
```

This Queue is normally implemented as a **circular array**.
At first the Queue has perhaps been specified by an interface.

Implementation of the bounded buffer:

```
class BoundedBuffer extends Queue
{
    ...
    public synchronized void put (Object x)
    {
        while (super.isFull ())
        { // wait, until not full
            try { wait (); }
            catch (InterruptedException e) {};
        }
        // now not full
        super.put (x);

        // notify: an object has been put on buffer
        notifyAll ();
    }

    public synchronized Object take ()
    {
        while (super.isEmpty ())
        { // wait, until not empty
            try { wait (); }
            catch (InterruptedException e) {};
        }
        // now not empty
        Object x = super.take ();

        // notify: an object has been taken from buffer
        notifyAll ();

        return x;
    }
    ...
}
```

9.3.2 Implementation of a bounded buffer from existing class `Queue`

If we already have a class `Queue` (e.g. from SDJ I2), it will be evident to use this.

If class `Queue` has the same methods as the class `BoundedBuffer` must have, the sub-classing will be an evident possibility, cf. above.

However, it might occur that

the methods in class `Queue` have other names than the methods in class `BoundedBuffer`, e.g.

enter	instead of	put
leave	instead of	take

or

class `Queue` has some methods we do not need or want in class `BoundedBuffer`, e.g.

front	"services" the front object in the queue without taking it out of the queue.
-------	--

None of the methods are suitable for class `BoundedBuffer`.

Using the *adapter-pattern* solves the problem in a clean way. Here the responsibility for the queue functionality is delegated to class `Queue`.

```
public class BoundBuffUsingDelegation
{
    private Queue buffer;

    public BoundBuffUsingDelegation(int size)
    {
        buffer = new Queue(size);
    }

    public synchronized void put (Object x)
    {
        while (buffer.isFull ())
        { // wait, until not full
            try { wait (); }
            catch (InterruptedException e) {};
        }
        // now not full
        buffer.enter (x);

        // notify: an object has been put on buffer
        notifyAll ();
    }

    ...
}
```

9.3.3 Exercise

Implement class `BoundedBuffer` on the basis of class `Queue` by using the adapter-pattern. If you already have a circular array implementation of a queue, you are welcome to use this.

Literature

- [Han] Per Brinch Hansen: Java's Insecure Parallelism.
ACM SIGPLAN Notices. Vol. 34. April 1999. pp. 39-45.
- [Ha] Stephen J. Hartley: Concurrent Programming. The Java Programming Language.
Oxford University Press. 1998. 0-19-511315-2.
- [Hy] Paul Hyde: Java Thread Programming. SAMS. 1999. 0-672-31585-8.
- [Le] Bill Lewis & Daniel J. Berg: Multithreaded Programming with Java Technology.
Sun Microsystems/Prentice. 2000. 0-13-017007-0.
- [Lea 1] Doug Lea: Concurrent Programming in Java. Design Principles and Patterns.
Add-Wesl. 1997.
- [Lea 2] Doug Lea: Concurrent Programming in Java. Design Principles and Patterns.
2. Edition. Add-Wesl. 2000. 0-201-31009-0.
- [Ma] Jeff Magee & Jeff Kramer: Concurrency. State Models & Java Programs.
Wiley. 1999. 0-471-98710-7.
- [Oa] Scott Oaks & Henry Wong: Java Threads. O'Reilly. 1997. 1-56592-216-6.
- [Ta] Andrew S. Tanenbaum: Modern Operating Systems. Prentice. 1992. 0-13-595752-4.