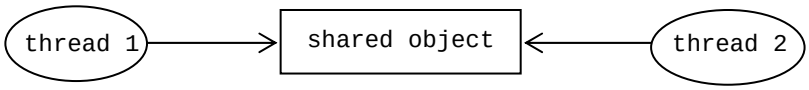
4 Communication between threads

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 treads 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 par 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):

signal and exit	i.e. after signal the signal thread exits the monitor immediately
signal and wait	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
signal and continue	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 InterrupterException
{
  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 InterrupterException
{
  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:

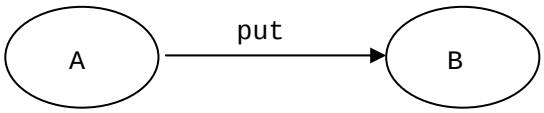


Figure 9.2.1 Push-based flow

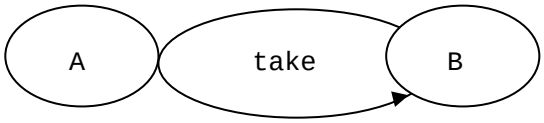


Figure 9.2.2 Pull-based flow

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

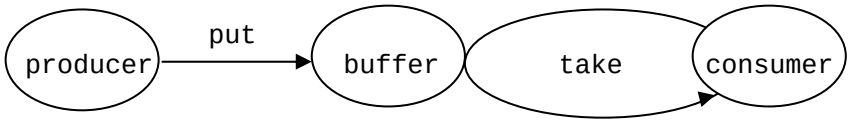


Figure 9.2.3 Put-take buffer

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 usedSlots;
  private int putPtr, takePtr;
  private Object [] theQueue;

  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 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.