Threads

Running several threads concurrently
Coherence of shared data
thread termination

Running several threads concurrently

- a thread is a lightweight process
  → swapping between such processes is very fast

- several threads may run concurrently, "in parallel"
  → if only one CPU, it executes alternatively each of the threads

definition of a thread:

declare a subclass of the **Thread** class that overrides its **run** method

run contains the task which will be carried out
by the thread instance of the subclass:

```
class MyThread extends Thread
{
   thread variables

   MyThread()
   {
      initialisation of the thread variables
   }

   public void run()
   {
      task carried out by the thread
   }
}
```

running a thread:

create and initialize the thread:

```
MyThread mythread = new MyThread();
```

Start, “in parallel” with the current thread, the execution of the **run** method of the thread:

```
mythread.start();
```

stopping a thread temporarily:

force the current thread to sleep for a certain number of milliseconds (**long**); thereby allowing other threads to be executed in their turn:

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

fairness:

if the execution of the current thread is too time consuming , give punctual opportunity to other threads (of same priority) to run as well:

```
Thread.yield();
```

sleep(...) and **yield()** are static methods of the class **Thread**

sleep(...) must ALWAYS be called within a try - catch statement

Coherence of shared data

problem:
when several threads read or write simultaneously some shared data
, the coherence of this data can be destroyed if each thread
does not access it exclusively from the others

→ the variables of an object (= shared data) must be accessed
by only one thread at a given moment

→ these variables must be accessed only through some methods of the object
that are synchronized together

= when one of these methods is being called by a thread
, no other thread can call another one of these methods

synchronized modifier for each of these methods (all within the same class) :

```
class Data
{
   for each of the variables:

   type variable;

   synchronized void write_variable(type value) { variable = value; }

   synchronized type read_variable() { return variable; }
}
```

→ within a thread: call **write_variable(value)**

→ within another thread: call **read_variable()**

example:

```
class Data
{
   int a,b;
   synchronized void write (int i) {  a=i;    System.out.print(".");    b=i;  }

   synchronized void read_and_check ()
   {  if (a!=b)  System.out.println("incoherent data");  }
}


class Thread1 extends Thread
{
   Data data;    Thread1(Data data)  {  this.data = data;  }

   public void run()
   {  int i=0;
      while (true)  {  i+=1;    data.write (i);    Thread.yield();  }  }
}


class Thread2 extends Thread
{
   Data data;    Thread2(Data data)  {  this.data = data;  }

   public void run()
   {  while (true)  {  data.read_and_check ();    Thread.yield();  }  }
}


public class Myprogram
{
   public static void main(String[] arg)
   {
      Data data = new Data();
      Thread1 thread1 = new Thread1(data);    thread1.start();
      Thread2 thread2 = new Thread2(data);    thread2.start();
   }
}
```

other version:

the critical data may belong to a thread:

```
class MyThread extends Thread
{
    for each of the variables:
    type variable;
    synchronized void write_variable(type value) { variable = value; }
    synchronized type read_variable() { return variable; }

   MyThread()
   {
      initialisation of the thread variables
   }

   public void run()
   {
      task carried out by the thread
      → within this task, call write_variable(value) or read_variable()
         or call a method with synchronized modifier
         (must be executed fast to prevent starvation of other threads)
   }
}
```

→ within another thread:

```
MyThread mythread = new MyThread();
mythread.start();
. . . . . .
mythread.write_variable(value);
. . . . . .
mythread.read_variable(value);
. . . . . .
```

Thread termination

a given thread should not force another thread to terminate suddenly !

BUT a given thread can *request* another thread to terminate as soon as possible by means of a synchronized flag **end** that is shared between the two threads:

the first thread sets **end** at true

→ the second thread reads the value of **end** (in the test of its main loop) and consequently terminates in an orderly and clean way

```
class MyThread extends Thread
{
   boolean end;
   synchronized void write_end(boolean end) { this.end = end; }
   synchronized boolean read_end() { return end; }

   MyThread()
   {
      end = false;
      . . . . . .
   }

   public void run()
   {
      while ( ! read_end() )
      {
        . . . . . .
      }
   }
}
```

→ within another thread:

```
MyThread mythread = new MyThread();
mythread.start();
. . . . . .
mythread.write_end(true);
```

Coordinating threads

waiting for another thread to finish:

```
try { other_thread.join(); } catch(InterruptedException e) { }
```

giving up and regaining the lock over an object inside synchronized methods:

When executing a synchronized method over an object, a thread obtains a "lock" on this object (it is the only one allowed to access it)

→ the thread gives up its lock by applying method **wait()** on this object (it will wait until it regains the lock):

```
try { this.wait(); } catch(InterruptedException e) { }
```

→ another thread then applies method **notifyAll()** on this object to give back their lock to all threads waiting on this object (they can continue their execution):

```
this.notifyAll();
```

wait() and **notifyAll()** are instance methods of the class **Object**

wait() and **notifyAll()** must *ALWAYS* be used inside some synchronized methods

join() and **wait()** must ALWAYS be called within a try - catch statement

example:

```
public class MyProg
{
   public static void main(String[] arg)
   {
      Sleeping_thread st = new Sleeping_thread();
      st.start();
      try { st.join(); } catch(InterruptedException e) { }
      System.out.println("the sleeping thread has finally joined us...");
   }
}


class Sleeping_thread extends Thread
{
   public void run()
   {
      try { Thread.sleep(10000); } catch(InterruptedException e) { }
      System.out.println("finished with sleeping");
   }
}
```

example:

```
public class MyProg
{
   public static void main(String[] arg)
   {
      Product p = new Product();
      Producer producer = new Producer(p);
      Consumer consumer = new Consumer(p);
      producer.start();
      consumer.start();
   }
}


class Product
{
   boolean present;
   double value;
   Product()  {  present = false;  }

   synchronized void add(double v)
   {
      while (present)  try {  this.wait();  }  catch(InterruptedException e)  {  }
      present = true;
      value = v;
      this.notifyAll();
   }

   synchronized double take()
   {
      while (!present)  try {  this.wait();  }  catch(InterruptedException e)  {  }
      present = false;
      this.notifyAll();
      return value;
   }
}
```

example continued:

```
class Producer extends Thread
{
   Product p;

   Producer(Product p)  {  this.p = p;  }

   public void run()
   {
      for (int i = 0 ; i < 20 ; i++)
      {  p.add(i);
         try {  Thread.sleep(500);  }  catch(InterruptedException e)  {  }  }
   }
}


class Consumer extends Thread
{
   Product p;

   Consumer(Product p)  {  this.p = p;  }

   public void run()
   {
      while (true)
      {   double v = p.take();
          System.out.println(v);  }
   }
}
```