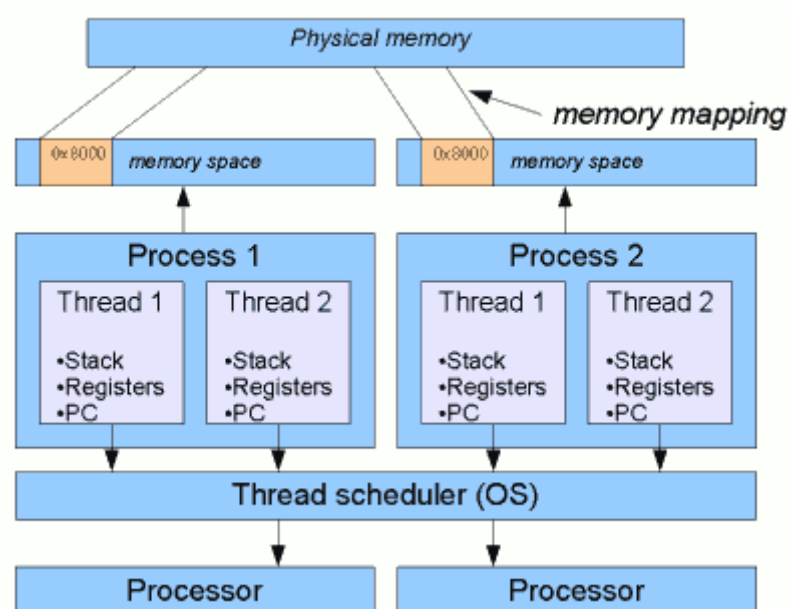


Java & Threads

Threads sind genau genommen **Subprozesse**.

Man kann sich diese als Teilaufgaben eines gesamten Programmes vorstellen, die quasi gleichzeitig laufen. Abhängig von der Anzahl der verfügbaren CPU's und der Anzahl der gleichzeitig laufenden Threads werden manche dieser Threads wirklich gleichzeitig laufen, während andere durch jeweils kurzfristiges Verwenden einer CPU den Eindruck des gleichzeitig Laufens vermitteln. Ein Teil des Betriebssystems, der so genannte **thread scheduler** kümmert sich um die Zuweisung der vorhandenen CPU's an die Threads.



Beispiel: Prozess → Browser

Thread 1: Seiten laden und anzeigen

Thread 2: Download starten

Thread 3: Musikdatei abspielen

all dies kann „gleichzeitig“ in einem Browserfenster passieren

Möglichkeiten in Java

1. Threads erzeugen

- class MyThread **extends Thread**
- class MyThread2 **implements Runnable**

für beide Versionen gilt: es ist eine spezielle Methode **run()** zu implementieren, mit deren Hilfe der Thread gesteuert wird !! (Üblicherweise haben/brauchen diese Klassen auch keine main() – Methode → diese befindet sich (wie gewohnt) in einer – zusätzlichen- Programmklasse, in der die Threads aufgerufen werden. Der Aufruf der Threads MUSS über die Methode **start()** erfolgen !!!

Programmbeispiele:

ad 1)

```
class DateThread extends Thread {  
    public void run() {  
        for (int i = 0; i < 2000; i++) {  
            System.out.println(new Date());  
        }  
    }  
}
```

```
public static void main(String[] args) {  
  
    DateThread dt1 = new DateThread();  
    dt1.start();  
  
    //anonymes Objekt  
    new DateThread().start();  
}
```

Ad 2)

```
class CounterThread implements Runnable {  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 2000; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

```
public static void main(String[] args) {  
    //anonymes CounterThread-Objekt  
    Thread ct1 = new Thread( new CounterThread() );  
    ct1.start();  
  
    //alles anonym  
    new Thread(new CounterThread()).start();  
  
    //anonymes Thread-Objekt  
    CounterThread ctol = new CounterThread();  
    new Thread(ctol).start();  
}
```

Die erzeugten Ausgaben (speziell die Reihenfolge) sind immer unterschiedlich

2. Eigenschaften von Threads

.) Name eines Threads

Alle Threads werden grundsätzlich durchnummeriert und haben defaultmäßig den Namen „thread-nn“

Name ermitteln :

... extends Thread → getName()

... implements Runnable → Thread.currentThread().getName()

Jedes Programm (main-Methode) ist ihrerseits ein Thread → „main“

- Zuweisen eines Namens

➔ Wenn von Thread abgeleitet: Übergabe einer Bezeichnung im Konstruktoraufbau -> Konstruktor mit Stringargument muss implementiert werden

```
class DateThread2 extends Thread {  
    public DateThread2(String name) {  
        super(name);  
    }  
    public void run() {  
        for (int i = 0; i < 20; i++) {  
            System.out.println(getName() + ": " + new Date());  
        }  
    }  
}
```

➔ Wenn Runnable implementiert: Name als zweites Argument beim Anlegen des Thread-Objektes

```
Thread ct1 = new Thread( new CounterThread2(), "counterThread" );  
ct1.start();
```

➔ setName() – Methode;

- Unterbrechen eines Threads – (generell eines Programmes):

- Thread.sleep(millis) ... statische Methode der Thread-Klasse ; muss InterruptedException behandeln

- Warten auf das Ende eines/mehrerer Threads:
 - `threadObjekt.join()`

```
public static void main(String[] args) {
    //anonymes CounterThread-Objekt

    System.out.println(Thread.currentThread().getName());

    DateThread3 dt1 = new DateThread3("dateThread");
    dt1.start();

    Thread ct1 = new Thread( new CounterThread3(), "counterThread" );
    ct1.start();

    try {
        dt1.join();
        ct1.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("alles erledigt");
}
```

- Das Schlüsselwort `synchronized`

Greifen mehrere Threads auf dieselben Ressourcen zu, so kann es zwecks Problemvermeidung notwendig sein Threads zu synchronisieren.

Eine Methode können wir durch das Schlüsselwort *synchronized* kennzeichnen

```
synchronized void xxx()
{
    // ...
}
```

Die VM wird diese Methode nun bei Bedarf automatisch sperren und entsperren.

Auch einzelne Code-Blöcke können synchronisiert werden (folgt später)

```
synchronized(objekt)
{
    // ...
}
```