Threads

parallele Prozesse
auf
sequenziellen Prozessoren

Prozesse und Threads

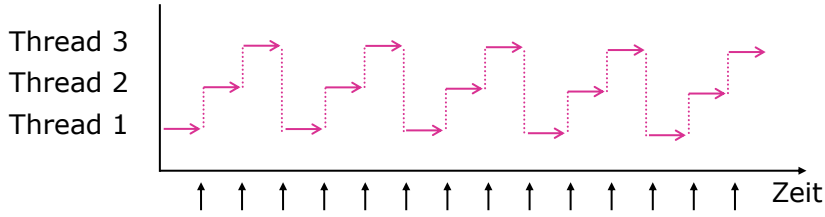
Es gibt zwei unterschiedliche Programme:

Ein Process ist ein typisches Programm, mit eigenem Addressraum im Speicher.

Ein Thread ist eine Sequenzfolge von Operationen innerhalb eines Prozesses. Mehrere Threads in einem Prozess teilen sich denselben Adressraum.

Parallel Processing kann somit durch mehrere Prozesse oder durch mehrere Threads erreicht werden.

Parallele Prozesse



Threads laufen nur quasi-parallel!
In Wirklichkeit läuft jeder Thread nur für sehr kurze Zeit und gibt dann Kontrolle an den nächsten Thread ab.

Thread class

Einfaches erzeugen eines Threads:

- Eigene Klasse von **`java.lang.Thread`** ableiten.
- Eigenen Kode in die **`run()`** Methode schreiben.
- Thread mit der **`start()`** Methode starten.

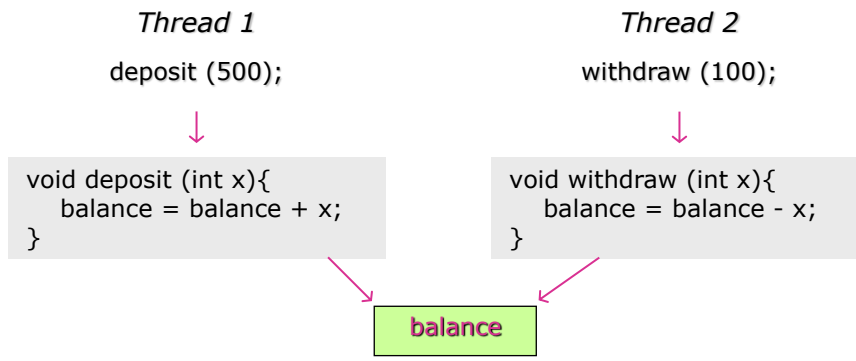
Threads: ein Beispiel

```
public class CharPrinter extends Thread{
   char signal;
   public CharPrinter (char ch) { signal = ch;}
   public void run() {
            for(int i =0; i<20;i++){
                     System.out.print(signal);
                     int delay = (int) (Math.random() * 400);
                     try {sleep(delay);}
                     catch(Exception e){return;}
            }
   }
}
```

```
class Programm{
   public static void main (String[] arg){
            CharPrinter thread1 = new CharPrinter('.');
            CharPrinter thread2 = new CharPrinter('*');
            thread1.start();
            thread2.start();
            System.out.print('+');
            }
}
```

Threads und externe Daten

Wenn Threads externe Daten verändern, bedarf dies spezieller Kontrolle!



Zugriff auf dieselben Daten durch zwei verschiedene Threads kann Konflikte verursachen.
Werden Daten zwischen dem Lesen und dem Schreiben durch einen zweiten Thread verändert, bleibt dies für den Ersten unbemerkt.

Synchronisation von Threads

```
class  Account {
   int  balance = 0;
   ……
   synchronized void deposit (int x){ balance = balance + x;}
   synchronized void withdraw (int x){ balance = balance - x;}
}
```

Ist eine Methode als *synchronized* deklariert verhindert Java, dass eine andere *synchronized-Methode* aktiv wird.

Synchronized bei langen Methoden

synchronized kann in die Methode hineingezogen werden um den relevanten Block zu schüzen.

```
synchronized void myMethod    (int x)  {
          ……
          balance = balance + x;
          …          }
```

```
void myMethod    (int x)  {
          ……
           synchronized (this) {
                    balance = balance + x;
          }
          …
}
```

Vorteil ist, dass nur der wichtige Teil blockiert wird, ansonsten sind beide Versionen gleichwertig.
Nachteil, der Bytecode wird etwas länger.

wait() und notify()

Threads werden im Allgemeinen automatisch verwaltet .

Wenn jedoch *Thread A* genau dann starten soll wenn *Thread B* abgearbeitet ist, muss B dies A mitteilen .

1. Möglichkeit:
   B verändert eine Variable deren Zustand als Nachricht dient und **A** fragt den Zustand dieser Variablen permanent ab.

   Nachteil: *Thread A* muss immer im Abfragezyklus bleiben und kann nicht ganz anhalten.

2. Möglichkeit
   wait() und notify() Methoden der Klasse Objekt verwenden.

wait() und notify() (2)

Die *wait()*-Methode wird innerhalb einer syncronized Anweisung (Block) verwendet.
Sobald die *wait()*-Methode ausgeführt wird, können andere syncronized Anweisungen starten und der Thread wartet bis er benachrichtigt wird.

Die *notify()*-Methode wird ebenfalls innerhalb einer syncronized Anweisung aufgerufen. Beim Aufruf benachrichtigt sie einen Thread der wartet.
Warten mehrere, dann wird ein zufälliger benachrichtigt.

```
class  MyClass {
        ……
        // Tread A
        public synchronized void waitForMessage(){
                try {  wait(); }
                catch (InterruptedException ex ) {  }
        }
        // Tread B
        public synchronized void sendMessage(){
                …….
                notify( );
        }
}
```