Überblick

5 Effiziente Programme

- Schnelle und langsame Programme
- Speicherplatz

Speichersystem

Java verwendet ein sehr bequemes und zuverlässiges Speichersystem.

Mit $x =$ **new** *Person*(...) wird Speicher für ein neues Objekt alloziert.

Durch eine *garbage collection* werden regelmäßig nicht mehr benötigte Objekte wieder freigegeben.

Dies passiert transparent.

Aber: Jedes **new** kostet Zeit und jedes Objekt verschwendet ein bißchen Speicherplatz.

Wenn dies eine Rolle spielt, dann sind primitive Datentypen und Arrays die schnellste und sparsamste Lösung.

Initialisierung eines Arrays:

```
static int[ ] arrayTest(int n) {
   int a[ ] = new int[n];
   for(int i = 0; i < n; i++) {
      a[i] = i;
   }
   return a;
}
```

Dies dauert mit $n = 100.000.000$ z.B. 0.16 Sekunden.

Jetzt verwenden wir eine $\mathit{ArrayList}\langle\mathit{Integer}\rangle$.

```
static List<Integer> listTest(int n) {
    List<Integer> l = new ArrayList<Integer>();
    for(int i = 0; i < n; i++) {
        l.add(i);
    }
    return l;
}
```

Laufzeit jetzt 14.2 Sekunden.

Neues Kapitel

Überblick

Computer haben oft mehrere Prozessoren.

Tendenz ist zunehmend.

Aufgenommen im Deutschen Museum von Clemens Pfeiffer.

Fazit: Paralleles Programmieren immer wichtiger.

Ein Programm kann mehrere laufende *threads* besitzen.

Diese laufen gleichzeitig oder scheinbar gleichzeitig.

So können wir einen *thread* starten:

```
Thread thread = new Thread(somethingRunnable);
thread.start();
```

Hierbei ist *somethingRunnable* ein Objekt, das das Interface *Runnable* implementiert.

Hierfür muß es eine Methode **public void** *run*() besitzen.

Beispiel

```
class ZahlenZeiger implements Runnable {
  private int id;
  private volatile boolean finished = false;
  public ZahlenZeiger(int id) {
    this.id = id;
  }
  public void run() {
    for(int i = 0; i < 100000; i++) {
      System.out.println(id + ": " + i);
    }
    finished = true;
  }
  public boolean finished() {
    return finished;
  }
}
```

Volatile variables

Wir können eine Variable **volatile** deklarieren.

Damit sagen wir, daß auf diese Variable von mehreren Threads zugegriffen werden kann.

Folgendes Programmstück kann vom Compiler optimiert werden, da sich i niemals ändert.

```
int i = 1;
while(i == 1) {}
```

Die Schleife kann aber beendet werden, wenn i von *einem anderen Thread* verändert wird.

Ein weiteres Problem sind Caches, die den Speicher von verschiedenen Prozessoren verschieden aussehen lassen.

Lösung: **volatile int** $i = 1;$

"Busy waiting" (oder "polling"):

```
static void test1() {
  List<ZahlenZeiger> zeigerList = new ArrayList<ZahlenZeiger>();
  for(int i = 1; i <= 10; i++) {
    ZahlenZeiger z = new ZahlenZeiger(i);
    zeigerList.add(z);
    Thread thread = new Thread(z);
    thread.start();
  }
  boolean stillRunning = true;
  while(stillRunning) {
    stillRunning = false;
    for(ZahlenZeiger z : zeigerList) {
      if(!z.finished()) {stillRunning = true;}
    }
  }
}
```