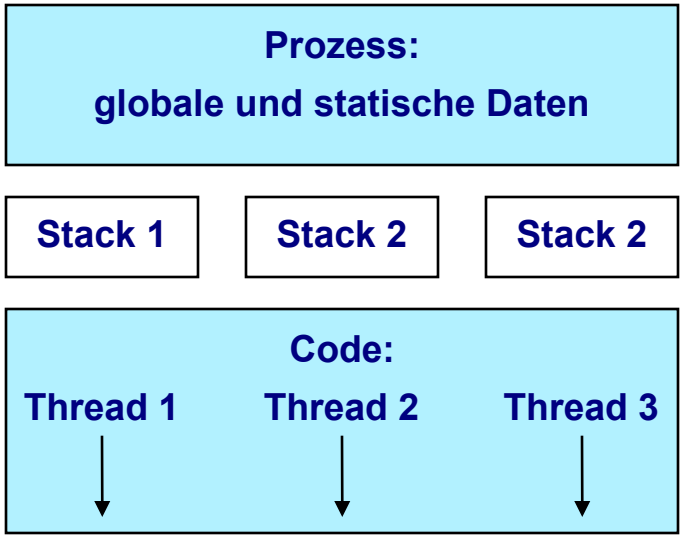
Threads

Stephan Karrer

Thread-Konzept



Vorteile:

- effizienter Kontextwechsel
- gut für Serverprozesse
- gut abbildbar auf symmetrische Multiprozessor-Architektur
- effiziente Interthread-Kommunikation

Nachteile:

- weniger robust als Multiprozessansatz
- Synchronisation ist eigenes Thema
- Abbildung von "User Level Threads" auf "Kernel Threads" kann auch suboptimal sein
- löst auch nicht die Frage "Was ist parallelisierbar?"

Threads in Java: Erweiterung der Klasse Thread

```
public class MyThread extends Thread {
    public void run() {
        for (int i=1; i<=1000; i++) {
            System.out.println(this.getName()+" : "+i);
    }
}
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        t1.start(); t2.start();
    }
}
```

- Die run-Methode kapselt in Java den Thread-Code.
- Die Klasse Thread stellt eine leere run-Methode zur Verfügung.
- Mittels der geerbten start-Methode der Klasse Thread wird ein neuer Thread gestartet (erzeugt) und die run-Methode aufgerufen.
- Die start-Methode kann nur einmal je Threadobjekt benutzt werden.

Threads in Java: Verwendung der Schnittstelle Runnable

```
public class MyThread implements Runnable {
   public void run() {
     for (int i=1; i<=1000; i++) {
    System.out.println(Thread.currentThread().getName()+" : "+i);
   }
}
   public static void main(String[] args) {
    Thread t1 = new Thread(new MyThread());
    Thread t2 = new Thread(new MyThread());
    t1.start(); t2.start();
    }
}
```

- Sofern die Erweiterung der Klasse Thread nicht geeignet, kann dem Konstruktor der Thread-Klasse ein Objekt vom Typ Runnable übergeben werden.
- Das Interface Runnable schreibt die run-Methode vor.
- Mittels der start-Methode der Klasse Thread wird ein neuer Thread gestartet (erzeugt) und die run-Methode aufgerufen.

Threads in Java: Runnable als Lambda (ab Java 8)

```
public class MyThread_Lambda {

public static void main(String[] args) {
    Runnable r = () -> { for (int i = 1; i <= 1000; i++) {
    System.out.println(Thread.currentThread().getName() + " : " + i);
             } };
    Thread t1 = new Thread(r);
    Thread t2 = new Thread(r);
    t1.start();
    t2.start();
    }
}
```

- Da Runnable ein funktionales Interface ist, kann eine Implementierung ab Java 8 selbstverständlich per Lambda-Schreibweise erfolgen.

Einige Attribute eines Threads

```
public class ThreadAttributes {
    public static void main(String[] args) {
        Thread current = Thread.currentThread();
        System.out.println(current.getId());   //long id
        System.out.println(current.getName());
        System.out.println(current.getPriority());
        System.out.println(current.getState());
    }
}
```

- Der Name eines Threads muss nicht eindeutig sein und kann per Konstruktor oder Setter (vor dem Start) gesetzt werden
- Die Id wird vom System vergeben und ist eindeutig
- Java unterscheidet 10 Prioritätsstufen (1 niedrigste, 10 höchste), realisiert als int-Konstanten. Diese kann vor dem Start des Thread vergeben werden. Ob das für die eigentliche Bestriebssystemplattform Bedeutung hat, ist offen.

Zustand des Threads

Zustand	Beschreibung
NEW	Thread erzeugt aber noch nicht gestartet
RUNNABLE	Thread ist aktiv
BLOCKED	Wartet an Sperre
WAITING	Wartet auf Benachrichtigung (z.B. auf notify())
TIMED_WAITING	Wartet auf Timeout (z.B. in sleep())
TERMINATED	Beendet

- Die Zustandsbeschreibungen sind als ENUM realisiert
- Mit isalive() kann abgefragt werden, ob der Thread gestartet und noch nicht beendet ist.

Schlafen: Freiwilliges Abgeben des Prozessors

```
//...
public void run() {
    for (int i=1; i<=10; i++) {
        System.out.println(this.getName()+" : "+i);
        try {
        Thread.sleep(1000);   //Millisekunden
        TimeUnit.SECONDS.sleep(2);  //alternativ
        }catch(InterruptedException e){ }
    }
//...
```

- Durch Aufruf der statischen sleep-Methode der Klasse Thread bzw. der sleep-Methode aus dem Enum TimeUnit kann sich der aufrufende Thread für eine Zeitspanne (Nanosekunden werden von vielen Implementierungen ignoriert) suspendieren.
- Ein schlafender Thread benötigt keine Rechenzeit und nach Ablauf der Zeit ist der Thread wieder rechenwillig.
- Aufgrund der internen Zeitauflösung kann es zu Abweichungen kommen.
  - Zur Ablaufsteuerung ist die sleep-Methode nur bedingt geeignet.

Freiwilliges Abgeben der Zeitscheibe

```
//...
public void run() {
    for (int i=1; i<=10; i++) {
        System.out.println(this.getName()+" : "+i);
        yield();
    } }
//...
```

- Durch Aufruf der yield-Methode gibt der Thread temporär den Prozessor ab, bleibt aber rechenwillig unter Beibehaltung der Priorität.

Warten auf Terminierung von Threads

```
public class MyThread extends Thread {
   public void run() {
    for (int i=1; i<=10; i++) {
        System.out.println(this.getName()+" : "+i); }
    }
   public static void main(String[] args) {
    MyThread t = new MyThread();  t.start();
    if (t.isAlive()){
            System.out.println("Erzeugter Thread läuft noch");
        try { t.join(); } catch (InterruptedException e) {}
        System.out.println("Erzeugter Thread ist terminiert"); }
} }
```

- Ein Thread terminiert, wenn sein Code (run- bzw. main-Methode) abgearbeitet ist.
- Mit der isAlive-Methode kann geprüft werden, ob ein Thread bereits terminiert ist.
- Mit der join-Methode kann der Aufrufer auf die Terminierung, optional mit Timeout, warten.

Beenden eines Threads von außen

```
//...
public static void main(String[] args) {
   MyThread t = new MyThread();
   t.start();
   //...
   t.interrupt();
   } }
//...
```

- Mit der interrupt-Methode kann einem Thread von außen der Terminierungswunsch signalisiert werden (es wird ein Interrupt-Flag gesetzt).
- Schläft der Thread, z.B. verursacht durch die sleep- oder join-Methode, wird er aufgeweckt (rechenwillig) und bekommt zur Laufzeit eine InterruptedException.
- Der Code des Thread entscheidet darüber, ob und wann terminiert wird !
- Es existiert auch die deprecated stop-Methode zum zwangsweisen Abbruch
  - deren Verwendung kann zu Inkonsistenzen führen

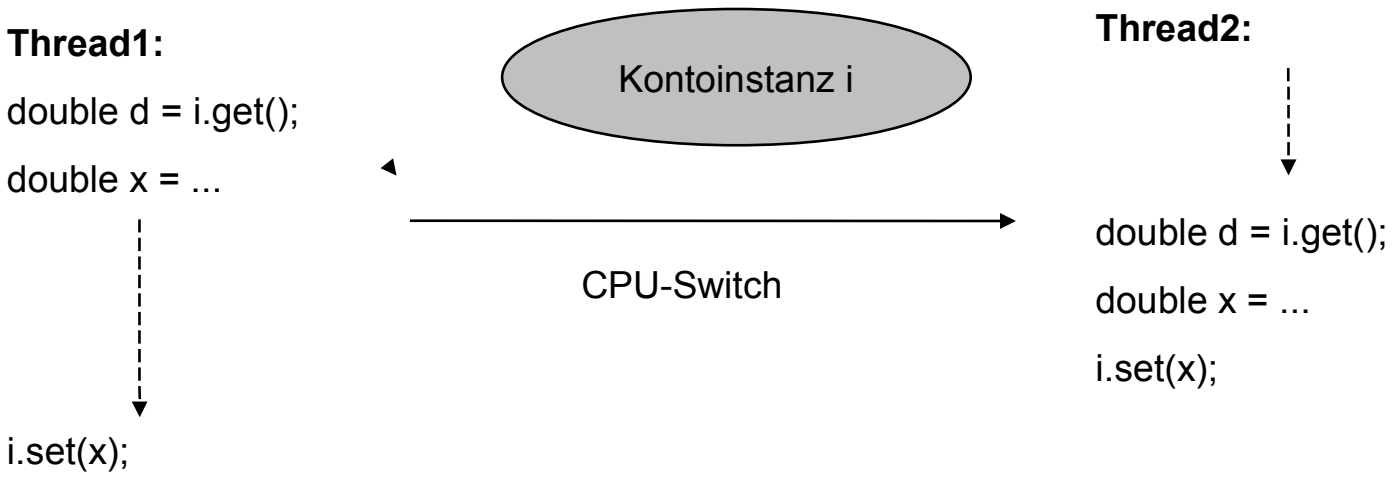
Reagieren eines Threads auf Terminierungswunsch

```
//...
public void run() {
  int i=0;
  try { while(!isInterrupted()){
            System.out.println(this.getName()+" : "+i);
            i++;
            Thread.sleep(10); }
   }catch(InterruptedException e){
            System.out.println("Aufgeweckt!"); }
  System.out.println("Jetzt terminiere ich");
}
//...
```

- Der Thread muss mittels der isInterrupted-Methode prüfen, ob ein Terminierungswunsch vorliegt
- Schläft der Thread, z.B. verursacht durch die sleep- oder join-Methode, wird er aufgeweckt (rechenwillig) und bekommt zur Laufzeit eine InterruptedException.
- Vorsicht: Das Interupt-Flag ist nach Auftreten der Exception wieder zurückgesetzt !!
- Es existiert auch eine static interrupted-Methode, die bei der Prüfung das Flag zurücksetzt.

Zugriff auf gemeinsame Daten

```
public class Konto {
    private double saldo;
    public double get() {return saldo;}
    public void set(double d) {saldo=d;}
}
```



Thread1 überschreibt die Aktualisierung von Thread2

Beispiel: Verwendung eines gemeinsamen statischen Zählers

```
public class MyThread extends Thread {
  static int cnt = 0;
  public void run() {
    while (cnt < 200) {
        System.out.println(getName()+": "+cnt++);
  } }
  public static void main(String[] args) {
    MyThread t1 = new MyThread();
    MyThread t2 = new MyThread();
    t1.start();
    t2.start();
  } }
```

- Auch die Verwendung zusammengesetzter Operatoren wie z.B. "++" ist nicht atomar.
- Das reine Setzen der Basisdatentypen außer long und double (64 Bit) ist atomar.
  - Allerdings kann der Compiler Optimierungen vornehmen (z.B. Halten des Werts im Register)
  - Dies kann duch die Kennzeichnung als volatile verhindert werden!

Beispiel: Verwendung von volatile

```
public class VolatileExample {

   private volatile int i;

   public void setValue(int value){
        i=value;
   }

   public int getValue(){
        return i;
   }
}
```

Kritische Abschnitte sichern mit **synchronized**

```
public class Konto{
    private double saldo
    public synchronized void buche(double betrag){
        saldo = saldo + betrag;
}}
```

```
public class Bucher extends Thread{
    private Konto k;
    public Bucher(Konto k){   this.k=k; }
    public void run(){ //... sonstiger Code
        double betrag = Math.random()*1000-500;
        k.buche(betrag);  //... sonstiger Code
    }
    public static void main(String[] args) {
        Konto k = new Konto();
        Thread b1 = new Bucher(k); Thread b2 = new Bucher(k);
        b1.start(); b2.start();
    }}
```

Wirkung von **synchronized**

- Das Schlüsselwort synchronized sorgt dafür, dass jeweils nur ein Thread den Rumpf der Methode ausführen kann
  - Der Zugriff wird serialisiert
  - Die Ausführung der Methode ist atomar
  - Intern wird beim Betreten eine Sperre gesetzt und beim Verlassen freigegeben
  - Der Sperrmechanismus ist aus Sicht der Applikation atomar
- Jeder weitere Aufruf der Methode durch einen anderen Thread solange die Sperre gesetzt ist, führ zum Blockieren (passives Warten) dieses Threads
  - Mehrere Threads können sich im Zustand des Wartens befinden
  - Bei Freigabe der Sperre werden die Wartenden aufgeweckt, aber nur einer kann als nächstes den kritischen Abschnitt betreten.
  - Sofern die Abarbeitung des kritischen Abschnitts lange dauert, führt das auch zu entsprechend lang dauerndem Warten
- Der Thread, der die Sperre hält, kann die Methode trotz Sperre erneut aufrufen
  - Dadurch ist Rekursion bei synchronized Methoden möglich

Verwendung von synchronized Blöcken

```
public class Bucher extends Thread {
    private Konto k;
    public Bucher(Konto k){  this.k=k; }
    public void run(){ //... sonstiger Code
        double betrag = Math.random()*1000-500;
        synchronized(k){
            double saldo = k.get();
            saldo = saldo + betrag;
            k.set(saldo);
        }
        //... sonstiger Code
        }

    public static void main(String[] args) {
        Konto k = new Konto();
        Thread b1 = new Bucher(k);
        Thread b2 = new Bucher(k);
        b1.start(); b2.start();
    }}
```

Kritische Abschnitte sichern mit **synchronized:** **So nicht!**

```
public class Bucher extends Thread{
    private Konto k;
    public Bucher(Konto k){   this.k=k; }
    public void run(){ //... sonstiger Code
        double betrag = Math.random()*1000-500;
        buche(betrag);  //... sonstiger Code
        }
    private synchronized void buche(double betrag){
        double saldo = k.get();
        saldo = saldo + betrag;
        k.set(saldo);
    }

    public static void main(String[] args) {
        Konto k = new Konto();
        Thread b1 = new Bucher(k);
        Thread b2 = new Bucher(k);
        b1.start(); b2.start();
    }}
```

synchronized Blöcke

- Die Verwendung von synchronized Blöcken ist das allgemeinere Konzept
  - Eine synchronized Methode kann stets durch einen synchronized Block ersetzt werden
- Bei einem synchronized Block wird ein Objekt (Instanz) benutzt, auf das sich die Sperre bezieht
  - Jedes Objekt in Java kann eine Sperre verwalten, dies ist bereits in der Basisklasse Object realisiert
  - Das für die Sperre benutzte Objekt kann beliebig sein, es muss nur gemeinsamer Zugriff durch die Threads existieren !!
  - In der Regel wird als Objekt eines benutzt, welches im Zusammenhang mit dem kritischen Abschnitt steht
- Bei einer nicht-statischen synchronized Methode bezieht sich die Sperre implizit auf das aktuelle Objekt (this)
- Die Sperren der einzelnen Objektinstanzen sind unabhängig voneinander

Bezug der Sperre auf ein Objekt

```
class MyClass {
  public void m1() { /* bel. Code */ };
  public synchronized void ms1() { /* bel. Code */ };
  public synchronized void ms2() { /* bel. Code */ };
```

- Führt ein Thread gerade die Methode ms1 aus, so kann ein anderer Thread weder die Methode ms1 noch die Methode ms2 ausführen !!
  - Die Sperre bezieht sich auf das aktuelle Objekt, nicht den Code der Methode
- Ein anderer Thread kann aber sehr wohl die Methode m1 ausführen
  - Die Berücksichtigung von Sperren ist freiwilliger Natur

Sperren auf Klassenebene

```
public class MyThread extends Thread {
  static int cnt = 0;
  public void run() {
    while (cnt < 200) {
        synchronized (getClass()) {
          System.out.println(getName() + ": " + cnt++); }
    } }
  public static void main(String[] args) {
    MyThread t1 = new MyThread();
    MyThread t2 = new MyThread();
    t1.start();
    t2.start();
  } }
```

- Beim konkurrierenden Zugriff auf statische Attribute **kann** das Klassenobjekt als Sperrobjekt dienen
- Bei einer statischen synchronized Methode bezieht sich die Sperre auf die Klasse
- Eine Sperre auf der Klasse bedingt keine Sperre auf den Objektinstanzen

Regeln zur Vermeidung inkonsistenter Zustände bei Nebenläufigkeit

Wenn von mehreren Threads auf ein Objekt zugriffen werden kann, wobei mindestens ein Thread den Zustand des Objekts verändert (Attribute schreibt), dann sollten alle Zugriffsmethoden auf dieses Objekt, egal ob lesend oder schreibend, als synchronized gekennzeichnet werden.

Da die Sperren sich auf Objekte beziehen, sollte die Granularität des Sperrmechnismus fein gesteuert werden.