Programmiermethodik

Threads

SS 2002

Thomas Kühne

kuehne@informatik.tu-darmstadt.de

http://www.informatik.uni-mannheim.de/informatik/softwaretechnik

Warum Threads?

Bessere Ressourcenausnutzung

- Beschleunigung durch Parallelisierung
  - » Multiprozessorenrechner
  - » Vektorrechner
- Vermeidung von Leerlauf
  - » Multi-User
  - » Multi-Tasking
- Simulation von Agenten
  - » aktive Objekte

Parallelisierung

Beispiel

- Matrixmultiplikation
  - » Statt $O(n^2)$ Skalarprodukte nur noch O(1) Skalarprodukte falls ausreichend Prozessoren zur Verfügung stehen

Threadprogrammierung unabhängig von der Anzahl der Prozessoren

Grenzen der Beschleunigung

- Bei nur 10% (p-%) sequentiellen Anteil ergibt sich maximal eine 10-fache (100/p-fache) Beschleunigung

Vermeidung von Leerlauf

Zeitscheibenverwaltung

- cooperative multitasking (z.B., Win3.1)
  - » jeder "Task" gibt die Kontrolle an einen "Task-Manager" ab
  - » Wechsel ist aktiv und freiwillig
- preemptive multitasking (z.B., Linux)
  - » "Tasks" werden automatisch unterbrochen
  - » faire Zeitverteilung garantiert

Threadscheduling ggf. abhängig von der VM (Win ≠ Linux)

Simulation von Agenten

Natürliche Nebenläufigkeit

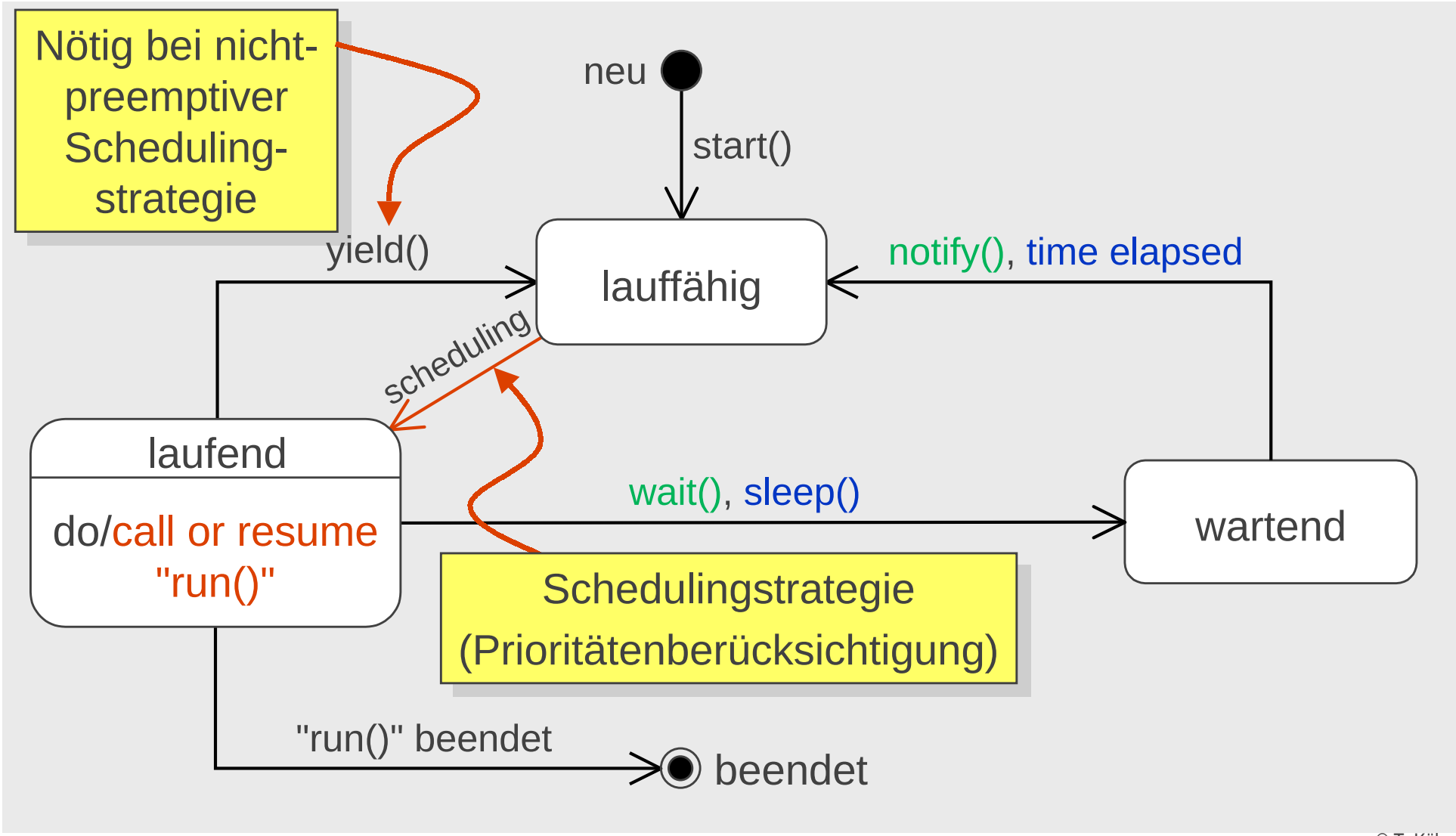
- Aktives Objekt, mehrfach instanziiert — Thread pro Objekt
  - » z.B., Billiardkugeln 
  - » keine eigene Zeitscheibenverwaltung nötig 
- Warten ohne zu blockieren — wait / notify
  - » z.B., I/O-Aktivitäten, die automatisch in Aktion treten sobald Daten verfügbar sind
- Periodische Vorgänge — sleep(time)
  - » einfache Intervallsteuerung

Threadausführung

Ausführungsaspekte

- Typischerweise gibt es weniger Prozessoren als Threads
  - » ein Prozessor muß mehrere Threads bearbeiten
- Manche Threads müssen auf Ereignisse warten, um fortfahren zu können
  - » andere Threads sollen davon unberührt bleiben
- Einige Threads sollen weniger Rechenleistung beanspruche als andere
  - » Prioritätenvergabe

Thread-Zustände



Beispiel: Konkurierrende Zähler

Thread Definition

Definiert Methoden wie start(), usw.

```
class Zaehler extends Thread  // Erbe von Thread
{
    public void run()
    {
        for (int i = 1; i < 10; i++)  // Zahlen von 1-9 ausgeben
        {
            String name=Thread.currentThread().getName();
            System.out.println(name + ": " + i);
        }
    }
    ...
```

Wird in Thread() definiert und hier redefiniert

Beispiel: Konkurierrende Zähler

Klient

```
public static void main(String[] args)   {
    Thread  t1 = new Zaehler(),
            t2 = new Zaehler();

    t1.setName("Anna");
    t2.setName("Bert");

    t1.start();
    t2.start();
  }
}
```

Thread erzeugen

Namen vergeben

Thread starten;
(Methode start() ist geerbt)
Aufruf kehrt unmittelbar wieder zurück

Bevorzugung eines Threads

```
    public static void main(String[] args)   {
        Thread  t1 = new Zaehler(),
                t2 = new Zaehler();

        t1.setName("Anna");
        t2.setName("Bert");

        t1.setPriority(Thread.MAX_PRIORITY);

        t1.start();
        t2.start();
    }
}
```

Priorität vergeben.
Ohne Angabe wird eine mittlere Priorität vergeben (5), bzw., die Priorität des Elternthreads geeerbt.

Alternierende Threads

```
public void run()  {
    for (int i = 1; i < 10; i++)  {
        String name=Thread.currentThread().getName();
        System.out.println(name + ": " + i);

        try  {
            Thread.sleep(1000);
        } catch (InterruptedException e)  {
            System.out.println(e); }
    }
}
```

Für eine Sekunde schlafen legen

"Schlaf" kann unterbrochen werden

Durch die "freiwillige Pausen" kommt auch ein weniger stark priorisierter Thread immer wieder zur Anwendung

Runnable Interface

Problem

- Was tun wenn eine Klasse bereits von einer anderen Klasse erbt und deshalb nicht die Klasse Thread beerben kann?
  - » z.B., ScrollingBanner **extends** Applet {
  - » soll aber jede 500ms einen Text um einen Buchstaben verschoben anzeigen

Lösung

- Implementierung des Runnable Interfaces
  - » ScrollingBanner **extends** Applet
    implements Runnable {

Runnable Interface

Thread Definition

Definiert run() Methode

```
class RunnableZaehler implements Runnable
{
    public void run()
    {
        for (int i = 1; i < 10; i++)  // Zahlen von 1-9 ausgeben
        {
            String name=Thread.currentThread().getName();
            System.out.println(name + ": " + i);
        }
    }
    ...
```

Wird in Runnable() deklariert und hier definiert

Runnable Interface

```
public static void main(String[] args)   {
    Runnable  z1 = new RunnableZaehler(),
              z2 = new RunnableZaehler();

    Thread    t1 = new Thread(z1),
              t2 = new Thread(z2);

    t1.setName("Anna");
    t2.setName("Bert");

    t1.start();
    t2.start();
  }
}
```

lauffähiges Objekt erzeugen

Thread erzeugen

Thread starten

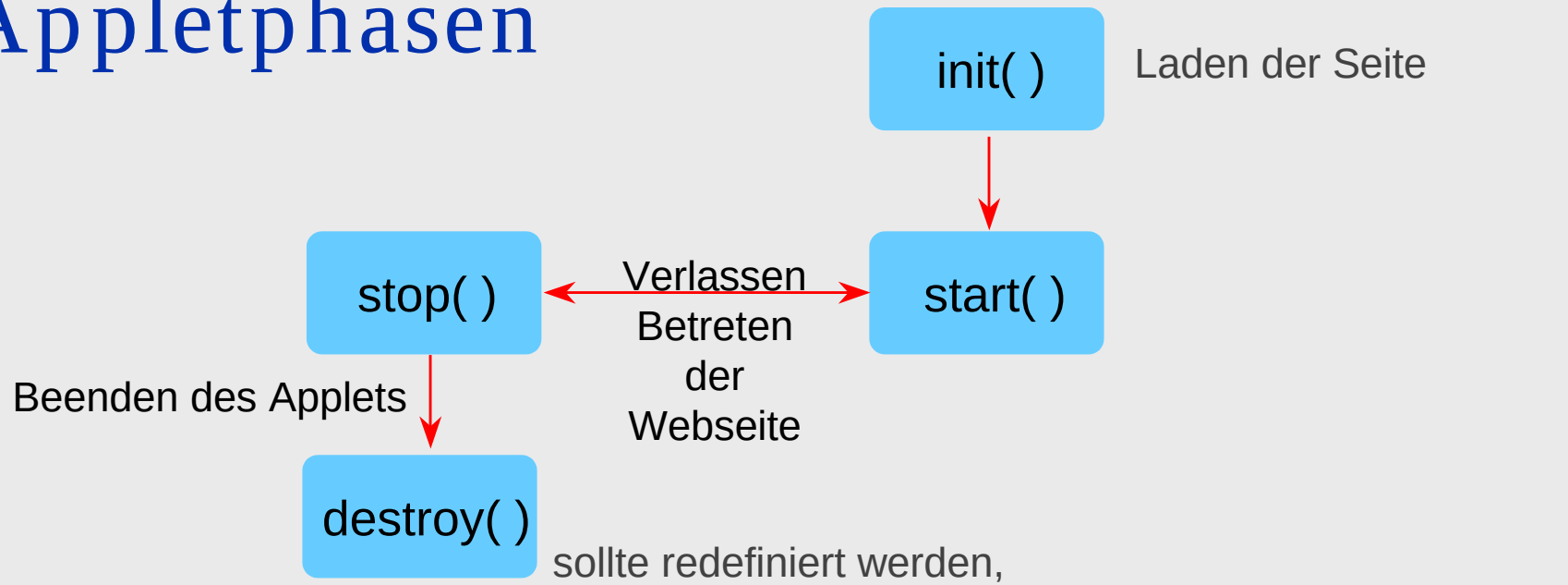
Thread Klasse

- **Konstruktoren**:
  `Thread(String n)`, `Thread(Runnable r, String n)`, und mehr...
- **`void`** `start()`
  initialisiert den Thread; VM ruft irgendwann run( ) auf
- **`void`** `run()`
  enthält den Programmcode, häufig eine “Endlos”-Schleife
- **`void`** `sleep(int milliSeconds)`
  unterbricht den Thread für die angegebene Wartezeit
- **`void`** `setPriority(int prio)`
  setzt die “Wichtigkeit” des Threads: 1 (min.) – 10 (max.)
  für die Zuteilung der Prozessorzeit
- **`void`** `setDaemon(boolean on)`
  Markieren als “Hintergrundprozess”. Laufen nur noch
  Hintegrundprozesse so beendet die Java VM die Ausführung

Beispiel: Aktives Applet

- Applet soll selbsttätig Aktionen ausführen
  - » z.B., ScrollBanner, hier Farbwechsel

Appletphasen



Beispiel: Aktives Applet

Thread Definition

Normales "ausführbares" Objekt

```
class BlinkThread implements Runnable {
    private Applet applet;  // Applet to be manipulated
    private int status;  // memorize current color

    BlinkThread (Applet a) {
        applet=a;
    }

    public void run() {
        for (;;) {  // loop forever
            try {
             applet.setBackground(status == 0 ? Color.green : Color.red);
             status = 1 - status;
             Thread.sleep(1000);
            } catch(InterruptedException ign) { }
        }
    }
```

Beispiel: Aktives Applet

Applet Definition

Erweiterung einer vordefinierten Appletklasse

```
public class BlinkingShapeApplet extends Applet {
    final static int QUAD=0, CIRC=1;  // shape constants
    private BlinkThread bt;  // background changing thread
    private int shape=QUAD;

    public void init() {  // called when applet is loaded
        final Button bq, bk;

        Panel p = new Panel();
        p.add(bq = new Button("Quadrat"));
        p.add(bk = new Button("Kreis"));
        add(p);  // add panel with two buttons

        bt = new BlinkThread (this);  // create runnable object accessing this applet
        new Thread(bt).start();  // create and start new thread (runs forever)
        ...
```

Beispiel: Aktives Applet

Applet Definition

```
.... (init continued)

// define action for button press
bq.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        shape=QUAD;
        bk.setEnabled(true);
        bq.setEnabled(false);
        repaint();
    }
});

// similar for the other button
....
} // end init
```

ActionListener bekannt aus den SWING Beispielen

Beispiel: Aktives Applet

Applet Definition

```
public void paint(Graphics g)
{
   g.setColor(Color.blue);

   if (shape==CIRC)  // depending on shape choice
     g.fillOval(100,100, 200, 200);  // draw circle
   else
     g.fillRect(100,100, 200, 200);  // draw rectangle
}
```

- Ergebnis
  - » unabhängiges Blinken des Hintergrunds (Thread)
  - » ereignisgesteuerte Wahl der Form

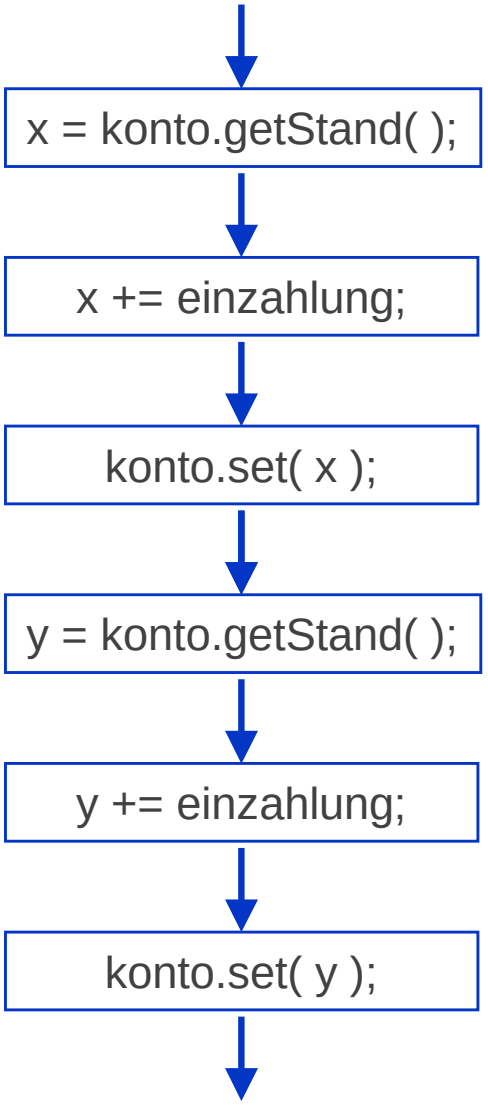
Applet HTML Inhalt

```
<applet code = BlinkingShapeApplet width = 400 height = 400 align = left>
 </applet>
```

Synchronisation: Motivation

Sequentieller Ablauf

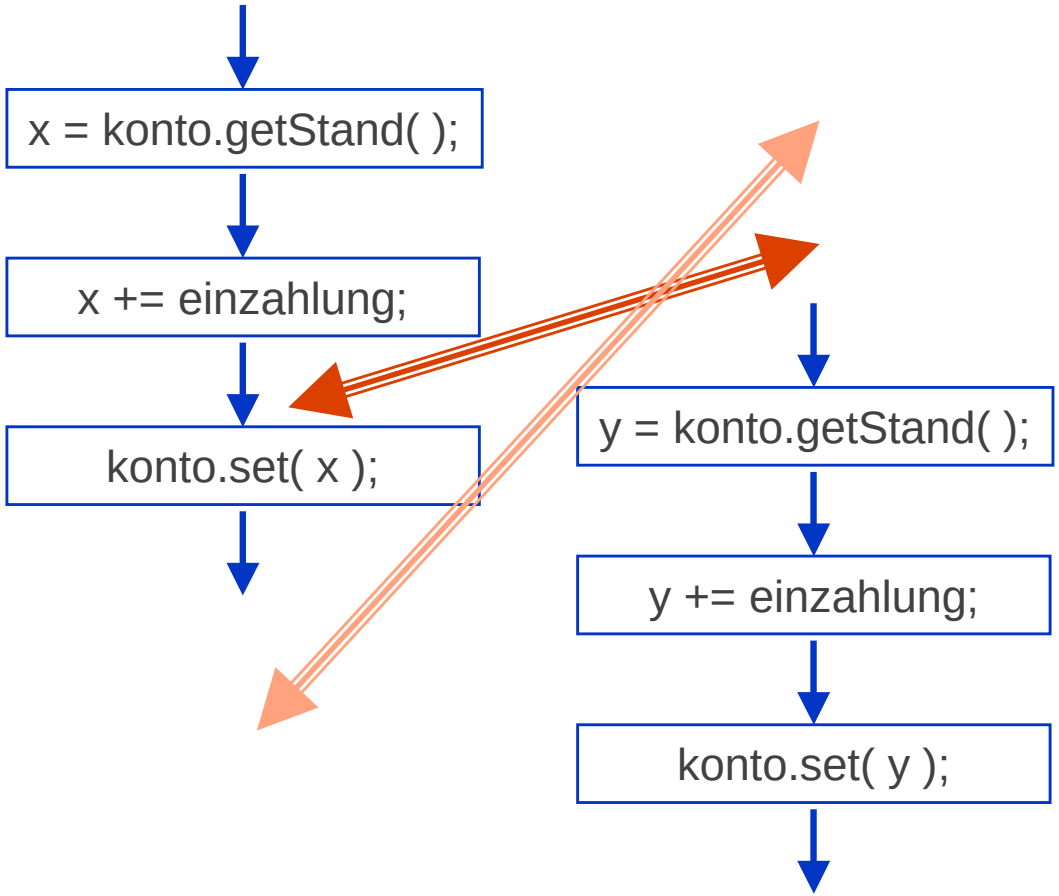
- Beispiel: Kontostandmanipulationen
- Anweisungen werden der Reihe nach ausgeführt.
- Die Einzahlung über y kann erst erfolgen, wenn die Transaktion über x vollständig erledigt ist.



Synchronisation: Motivation

Paralleler Ablauf

- Beide Aktionen können mit einander interferieren
- Eine Aussage über den endgültigen Kontostand ist unmöglich
- Eine Synchronisation wird notwendig



Synchronisation

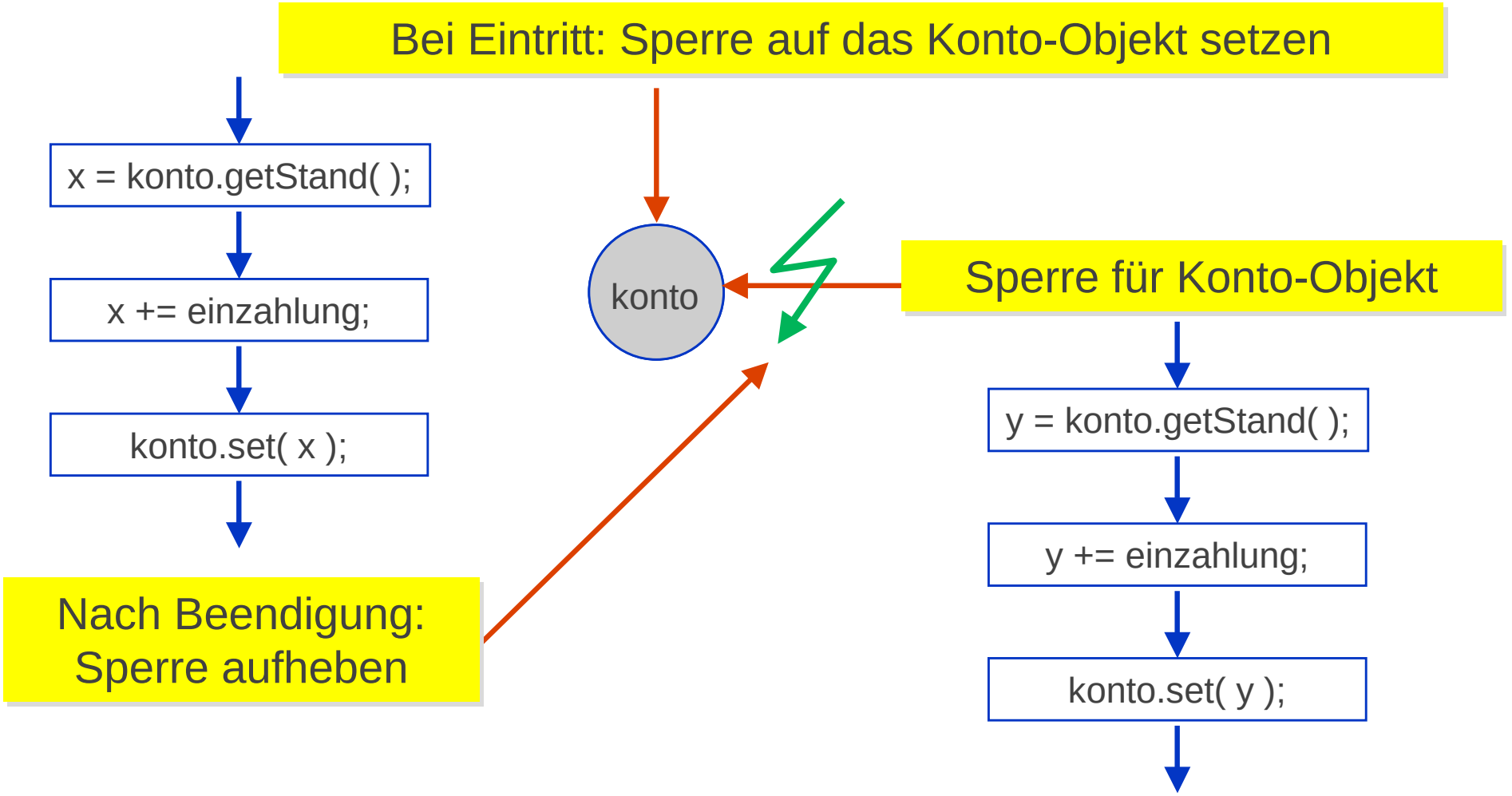
Problem

- Ein Thread wird mitten in der Ausführung einer Methode unterbrochen
  - » das Objekt befindet sich in einem inkonsistenten Zustand
  - » Setzt der unterbrechende Thread auf dem gleichen Objekt auf, dann findet er den inkonsistenten Zustand vor

Lösung

- Definition von kritischen Abschnitten
  - » werden mit Monitoren verwaltet
  - » Unterstützt durch das Schlüssewort synchronized
  - » und die Methoden wait(), notify()

Beispiel Kontoführung



Synchronisation

```
class Konto {
    private double stand;
    public synchronized void einzahlen( double x ) {
        stand += x;
    }
    public synchronized void abheben( double x ) {
        stand -= x;
    }
    public synchronized double stand( ) { return stand;}
}
```

Auswertung des aktuellen Standes und Zuweisung eines Neuen werden nicht unterbrochen

- Monitorzustände: frei oder vergeben (je Objekt)
- nächster Monitorbesitzer (angemeldete, wartende calls) wird zufällig gewählt
- Aktive Methode kann Monitor abgeben (wait()) und andere benachrichtigen (notify())
- Synchronisation auch über kritische Abschnitte kleiner als Methoden (einzelne Anweisungen oder Blöcke)

Kritische Abschnitte

```
class ThreadedClass {
    . . .
    public void wichtigeMethode () {
        . . .
        // unkritische Aktionen
        . . .
        // so jetzt bitte nicht stören:
        synchronized( this ) {
            // die nicht unterbrechbaren Anweisungen
            . . .
        }
        ...
        // unkritische Aktionen
        . . .
    } // wichtigeMethode( )
} // class ThreadedClass
```

nur der kritische Abschnitt wird synchronisiert

man kann auch auf andere Objekte per Monitor zugreifen, wenn sie sich nicht selbst synchronisieren:
synchronized(x) { x.critical() }

Zusammenfassung

Vorteile

- Threads ermöglichen nebenläufige Aktionen
- Einfache Erzeugung und Steuerung

Anforderungen

- Keine Vorher-sagbarkeit über Reihenfolgen mehr
- Synchronisation von Zugriffen nötig
  - » wait(), notify()

Threads für Alles?

The bearing of a child takes nine months, no matter how many women are assigned.

Frederick P. Brooks