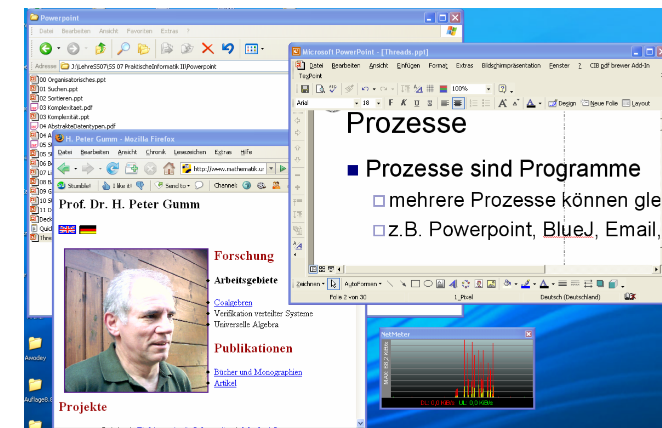
Threads

Prozesse, Parallelität, Nebenläufigkeit, Threads, Erzeugung, Ausführung, Kommunikation, Interferenz, Kritischer Bereich, Deadlock, Synchronisation.

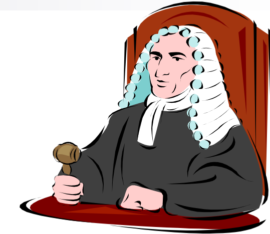
Prozesse

- Prozesse sind Programme
  - mehrere Prozesse können gleichzeitig laufen
  - z.B. Powerpoint, Firefox, Explorer, Netmeter, etc
- Prozesse können aus Unterprozessen bestehen
  - File chooser
  - Renderer
- Prozesshierarchie
  - Unterprozesse
  - Threads

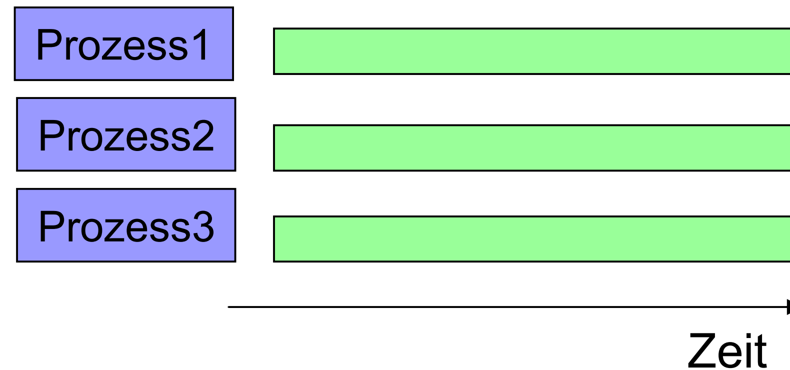


© H. Peter Gumm, Philipps-Universität Marburg

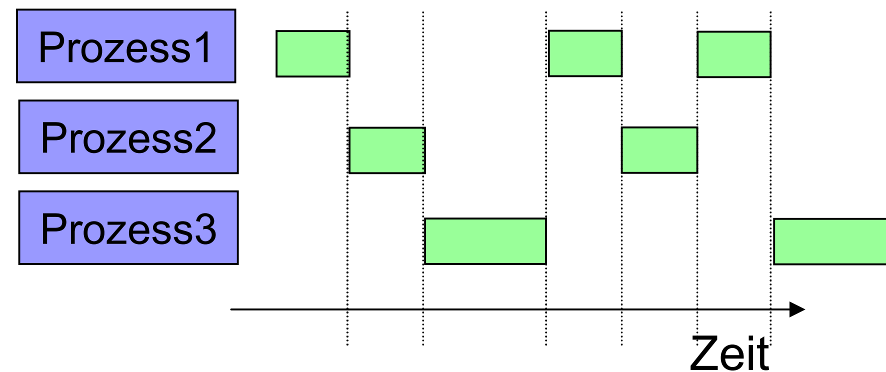
Prozessausführung



- Parallel – auf mehrere CPUs
  - gleichzeitige Ausführung
  - falls mehrere Prozessoren vorhanden
  - jede CPU hat einen Prozess
    - nur im Idealfall zu verwirklichen



- Nebenläufig – auf einer CPU
  - Gleichzeitigkeit wird vorgetäuscht
  - Prozesse werden reihum ausgeführt
    - jeweils immer nur ein Stückchen
    - dann kommt der nächste dran
  - Prozesswechsel



© H. Peter Gumm, Philipps-Universität Marburg

Prozesse - Programme

- Prozesse sind laufende Programme
  - Programmcode
  - Speicher für Variablen, Daten
  - Programmzähler
  - Stack, Heap
- Prozesswechsel aufwendig
  - Prozess auslagern
    - Werte für Variablen, Daten merken
    - Programmzähler merken
    - Stack, Heap merken
  - Ausgelagerten Prozess laden
    - Werte der Variablen, Daten
    - Programmzähler einstellen
    - Stack, Heap einlagern
    - Los gehts

```
1  public class Banking {
2    int konto = 0;
3
4    public void test(){
5      Produzent otto = new Produzent();
6      Konsument eva = new Konsument();
7      Konsument anna = new Konsument();
8      otto.start();
9      eva.start();
10     anna.start();
11   }
```

```
13       public void run(){
14           while(true){
15               if (konto < 100){
16                          konto += 10;
17                   }
18               assert konto <= 100 : "Zu
```

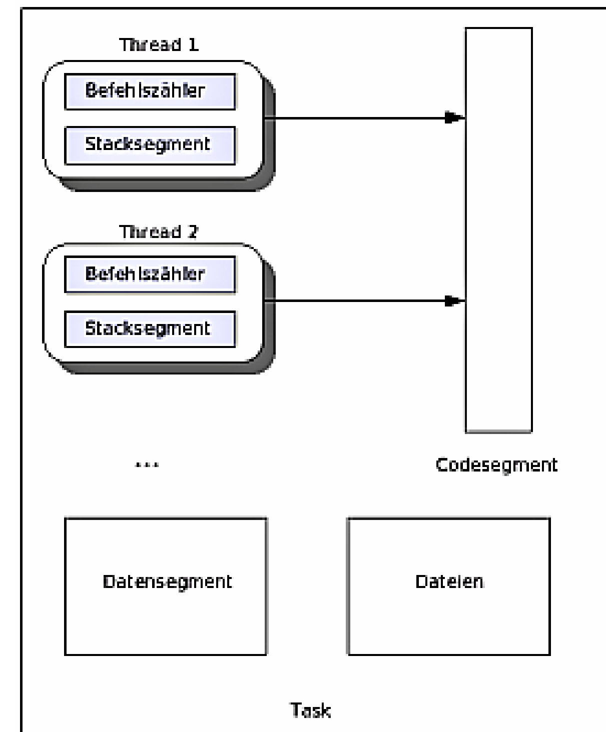
```
13     public void run(){
14         while(true){
15             if (konto < 100){
16                     konto += 10;
17
18
19         }
20       }
```

```
23   class Konsument extends Thread{
24     public void run(){
25         while(true){
26             if (konto >= 10){
27                  konto -= 10;
28             }
29             assert konto >= 0 : "Konto in
30   }}}
```


© H. Peter Gumm, Philipps-Universität Marburg

Threads

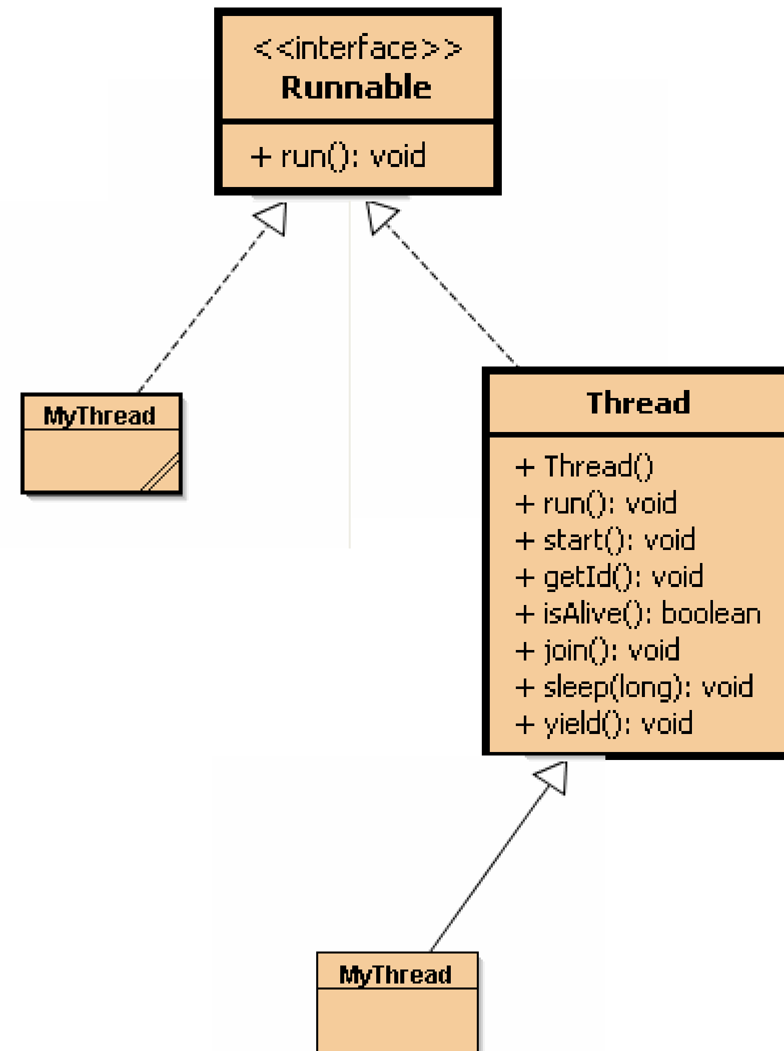
- Leichtgewichtige Prozesse
  - Faden
  - Ausführungsstrang
- teilen sich Daten
  - untereinander
  - mit Vaterprozess
- in Java: Objekte
  - Programmzähler
  - eigenes Stacksegment
  - Zeiger auf Programmcode
- Vorteil:
  - Taskwechsel sehr schnell
- mögliche Probleme:
  - viele Köche
  - kochen am gemeinsamen Brei



Quelle: wikipedia.de

© H. Peter Gumm, Philipps-Universität Marburg

Prozesse in Java



- Schon vorhanden in java.lang :
  - interface Runnable
    - myThread implements Runnable
    - muss nur void run() implementieren
  - class Thread implements Runnable
    - myThread extends Thread
    - sollte void run() überschreiben
- In jedem Fall
  - mit run() erledigt der Thread seine Arbeit
  - Betriebssystem ruft run() auf
  - Vergleiche „Applets“ im letzten Semester

© H. Peter Gumm, Philipps-Universität Marburg

Beispiel: Druckthreads

- Ganz normale Klassen
  - erben von Thread
  - redefinieren void run()

```
public class NumPrinter extends Thread{
  int wieviele;

   public NumPrinter(int wieviele){
     this.wieviele=wieviele;
    }

   public void run(){
       for (int i=0; i < wieviele; i++)
          System.out.print(i);
        }

}// Ende der Klasse NumPrinter
```

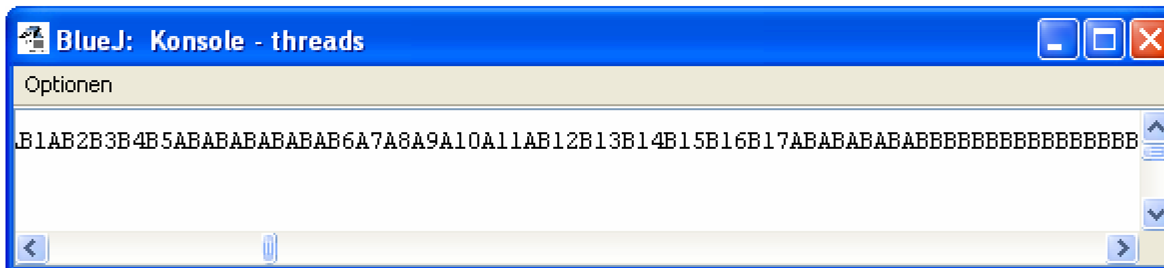
```
public class CharPrinter extends Thread{
  char zeichen;
  int wieOft;

  CharPrinter(char zeichen, int wieOft){
    this.zeichen = zeichen;
    this.wieOft = wieOft;
  }

  public void run(){
    for(int i=0; i< wieOft;i++)
      System.out.print(zeichen);
    }

}// Ende der Klasse CharPrinter
```


© H. Peter Gumm, Philipps-Universität Marburg

Erzeugung von Threads

```
 1 public class DruckTest {
 2
 3   public static void main(String[] args){
 4     CharPrinter printA = new CharPrinter('A',1000);
 5     CharPrinter printB = new CharPrinter('B',1000);
 6     NumPrinter printN = new NumPrinter(1000);
 7
 8     printA.start();
 9     printB.start();
10     printN.start();
11   }
```

- Erzeugen
- Starten



- Prozesswechsel beobachten

© H. Peter Gumm, Philipps-Universität Marburg

Alternative

- Klassen implementieren Runnable
  - d.h. void run()
  - sonst fast alles gleich
- Klassen sind noch keine Threads
  - kennen z.B. noch nicht Methoden
  - start(), join(), …
- wie kann man solche Kreaturen starten ?

```
public class PrintNum implements Runnable{
  int wieviele;

   public PrintNum(int wieviele){
     this.wieviele=wieviele;
    }

   public void run(){
       for (int i=0; i < wieviele; i++)
          System.out.print(i);
         }
}// Ende der Klasse PrintNum
```

```
public class PrintChar implements Runnable{
   char zeichen;
  int wieOft;

  public PrintChar(char zeichen, int wieOft){
    this.zeichen = zeichen;
    this.wieOft = wieOft;
  }

  public void run(){
    for(int i=0; i< wieOft;i++)
      System.out.print(zeichen);
    }
}// Ende der Klasse PrintChar
```


© H. Peter Gumm, Philipps-Universität Marburg

Threads starten

- *dekoriere* Klasse mit Thread
  - new Thread(
    new PrintChar(…));

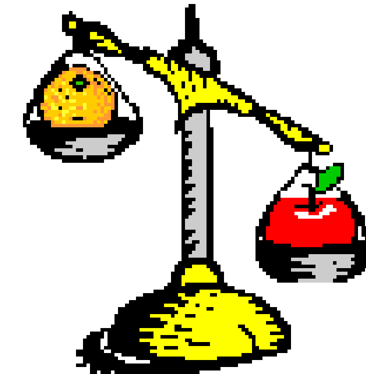
```
public class TestDruck {
    public static void main(String[] args){
      Thread printA = new Thread(new PrintChar('A',1000));
      Thread printB = new Thread(new PrintChar('B',1000));
      Thread printN = new Thread(new PrintNum(1000));

      printA.start();
      printB.start();
      printN.start();
    }
}// Ende der Klasse TestDruck
```

- Threads kennen start()
  - printA.start()
  - etc.

Constructor Summary
Thread()   Allocates a new Thread object.
Thread(Runnable target)   Allocates a new Thread object.
Thread(Runnable target, String name)   Allocates a new Thread object.

© H. Peter Gumm, Philipps-Universität Marburg

Vorteil - Nachteil



- **`MyClass extends Thread{`**
  - einfach
  - alles schon da
  - kann sofort gestartet werden
  - `(new MyClass(…)).start();`
- **`MyClass implements Runnable`**
  - nicht viel schwerer
  - **`MyClass`** ist noch kein Thread
  - aber passendes Argument für Thread-Konstruktor
  - `new Thread( new MyClass(…)).start();`
  - Mehrfachvererbung möglich:
    - `MyClass extends OtherClass implements Runnable`

© H. Peter Gumm, Philipps-Universität Marburg

Threads beenden

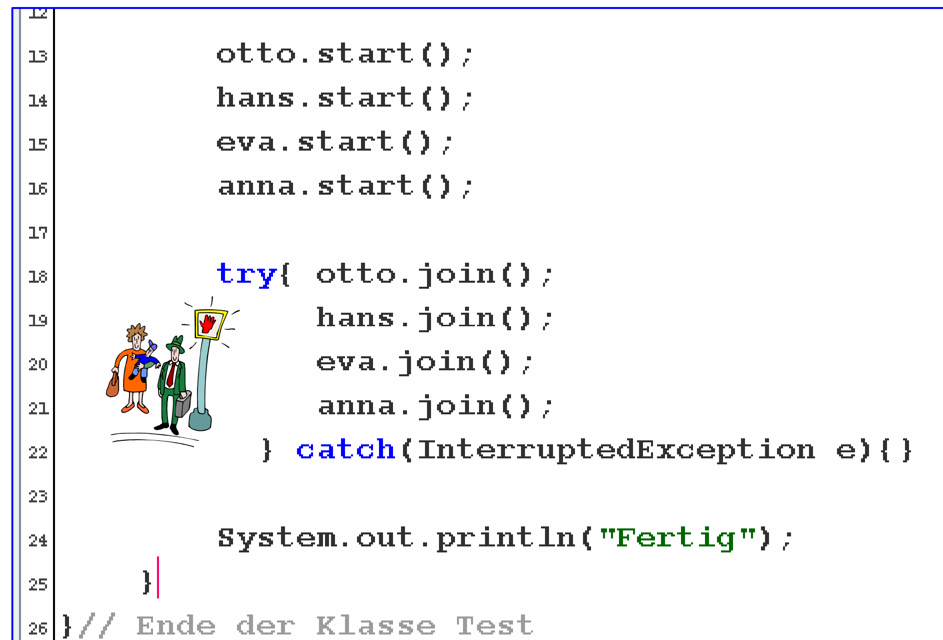
- das Hauptprogramm ist auch nur ein Prozess
- kann vor den Threads fertig werden
- druckt „Fertig“ bevor die Prozesse beendet sind
- man sollte warten können, bis die Prozesse zu Ende sind

```
 8          Buchhalter otto = new Buchhalter("Otto",meins, de
 9          Buchhalter hans = new Buchhalter("Hans",meins, de
10          Buchhalter eva = new Buchhalter("Eva",deins, mei
11          Buchhalter anna = new Buchhalter("Anna",deins, me
12
13          otto.start();
14          hans.start();
15          eva.start();
16          anna.start();
17
18          System.out.println("Fertig");
19      }
20
21  }// Ende der Klasse Test
```

© H. Peter Gumm, Philipps-Universität Marburg

Auf das Ende warten

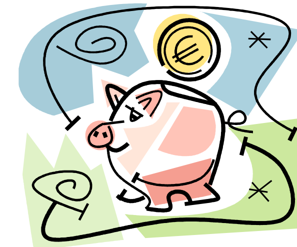
- join()
  - blockiert, bis Prozess zu Ende ist
- sleep(int millis)
  - warte eine bestimmte Zeit
- interrupt()
  - unterbrechen
- isAlive()
  - prüfen ob Prozess noch läuft



```
13          otto.start();
14          hans.start();
15          eva.start();
16          anna.start();
17
18          try{  otto.join();
19                hans.join();
20                eva.join();
21                anna.join();
22          } catch(InterruptedException e){}
23
24          System.out.println("Fertig");
25      }
26  }// Ende der Klasse Test
```


© H. Peter Gumm, Philipps-Universität Marburg

Zugriff auf gemeinsame Daten



- Otto überweist 10 € auf Konto
  - aber nur wenn kontoStand $\leq$ 90.
- Eva und Anna heben immer 10 € ab
  - aber nur wenn **`kontoStand`** $\geq$ 10€
- Eigentlich müsste gelten:
  - 0 $\leq$ **`kontoStand`** $\leq$ 100
- Aber

```
Exception in thread "Thread-3" java.lang.AssertionError: Konto in den Miesen
        at Banking$Konsument.run(Banking.java:29)
```

© H. Peter Gumm, Philipps-Universität Marburg

Fiese Prozesswechsel

- Bevor Konsument abhebt, prüft er, ob genug Geld auf dem Konto.
- Was, wenn zwei Konsumenten (eva, anna) abheben ?
  - Eva prüft:
    - ≥ 10 €
  - Anna prüft
    - ≥ 10 €
  - Anna holt ab
  - Eva holt ab
  - Konto in den Miesen !

```java
int konto = 0;

public void test(){
  Produzent otto = new Produzent();
  Konsument eva = new Konsument();
  Konsument anna = new Konsument();
  otto.start();
  eva.start();
  anna.start();
}
class Produzent extends Thread{
  public void run(){
      while(true){
          if (konto < 100){
                konto += 10;
             }
          assert konto <= 100 : "Zuviel auf dem Kont
       }
     }

   }
class Konsument extends Thread{
  public void run(){
      while(true){
          if (konto >= 10){
               konto = konto - 10;
          }
          assert konto >= 0 : "Konto in den Miesen";
 }}}
}// Ende der Klasse Banking
```


© H. Peter Gumm, Philipps-Universität Marburg

Fiese Prozesswechsel

- Unterbrechungspunkte zwischen Maschinen-Instruktionen
- nicht zwischen Java-Anweisungen

Mögliche Unterbrechungspunkte

```
   load konto
→  cmp 10
→  jmpless end:
→  load konto
→  sub 10
→  sto konto
end:
```

```java
int konto = 0;

public void test(){
  Produzent otto = new Produzent();
  Konsument eva = new Konsument();
  Konsument anna = new Konsument();
  otto.start();
  eva.start();
  anna.start();
}
class Produzent extends Thread{
  public void run(){
      while(true){
          if (konto < 100){
                 konto += 10;
          }
          assert konto <= 100 : "Zuviel auf dem Kont
      }
    }

  }
class Konsument extends Thread{
  public void run(){
      while(true){
          if (konto >= 10){
              konto = konto - 10;
          }
          assert konto >= 0 : "Konto in den Miesen";
}}}
}// Ende der Klasse Banking
```


© H. Peter Gumm, Philipps-Universität Marburg

Nicht Thread-safe

- Zwei Überweisungen von dem gleichen Konto können sich ins Gehege kommen:
  - Buchhalter1:
    - Prüfe, ob genug Geld
  - Buchhalter2:
    - Prüfe, ob genug Geld
  - Buchhalter2:
    - überweise
  - Buchhalter1:
    - überweise

```
1  public class Konto {
2      String inhaber;
3      int stand;
4
5      Konto(String inhaber){ this.inhaber = inhaber;}
6
7      void einzahlen(int betrag){
8          if(betrag > 0) stand += betrag;
9      }
10     void abheben(int betrag){
11         if(stand >= betrag) stand -= betrag;
12     }
13     void überweisen(Konto nach, int betrag){
14        if(stand > betrag && betrag > 0){
15            this.abheben(betrag);
16            nach.einzahlen(betrag);
17        }
18     }
19 }// Ende der Klasse Konto
```


© H. Peter Gumm, Philipps-Universität Marburg

synchronized – locks

- nur eine synchronized Methode kann gleichzeitig für ein Objekt der Klasse Konto aktiv sein
- Aufruf einer synchronized Methode fordert ein Schloss für ihr Objekt an
- falls verfügbar wird es erteilt, und die Methode erhält exklusiven Zugriff
  - alle anderen müssen warten
- falls nicht verfügbar muss die Methode warten
- am Ende der Methode wird das Schloss freigegeben

```
public class Konto {
    String inhaber;
    int stand;

    Konto(String inhaber, int anfang){
        this.inhaber = inhaber;
        stand = anfang;
    }

    synchronized void einzahlen(int betrag){
        if (betrag > 0) stand += betrag;
    }
    synchronized void abheben(int betrag){
        if(stand >= betrag) stand -= betrag;
    }
    synchronized void überweisen(Konto nach, int betrag){
        int summe = this.stand + nach.stand;
        if(stand > betrag && betrag > 0){
            this.abheben(betrag);
            nach.einzahlen(betrag);
            assert (this.stand + nach.stand == summe) : "Robbery detect
        }
    }
}// Ende der Klasse Konto
```


© H. Peter Gumm, Philipps-Universität Marburg

Deadlock

- Verklemmung
  - jeder wartet auf den anderen
  - keiner gibt nach
- Szenario bei gleichzeitigem Überweisen:
  - Prozess P will Geld von A nach B überweisen
  - Prozess Q will Geld von B nach A überweisen
- Überweisen von A nach B benötigt zwei Schlüssel
  - L_A für abheben von A
  - L_B zum Einzahlen auf B
- größter anzunehmender Unfall (GAU):
  - Prozess P fordert L_A und L_B und
    - erhält L_A
    - wartet noch auf L_B
  - Prozess Q fordert L_B und L_A und erhält zunächst L_B
    - erhält L_B
    - wartet noch auf L_A
  - sie werden ewig warten – Verklemmung !!

© H. Peter Gumm, Philipps-Universität Marburg

KontenDeadlock

- Buchhalter otto
  - will von kontoA auf kontoB überweisen
  - ruft *kontoA.überweisen* auf.
  - benötigt und erhält lock für kontoA
  - ruft *kontoA.abheben*
  - möchte *kontoB.einzahlen*
  - benötigt lock für kontoB
  - … aber in der Zwischenzeit …
- Buchhalter hans
  - will von kontoB auf kontoA überweisen
  - ruft *kontoB.überweisen* auf
  - benötigt erhält lock für kontoB
  - ruft *kontoA.abheben*
  - möchte *kontoA.einzahlen*
  - benötigt lock für konto A
  - … ist besetzt …
- Deadlock

```
9
10  synchronized void einzahlen(int betrag){
11      if (betrag > 0) stand += betrag;
12  }
13  synchronized void abheben(int betrag){
14      if(stand >= betrag) stand -= betrag;
15  }
16  synchronized void überweisen(Konto nach, int betrag){
17      int summe = this.stand + nach.stand;
18      if(stand > betrag && betrag > 0){
19         this.abheben(betrag);
20         nach.einzahlen(betrag);
21         assert (this.stand + nach.stand == summe) : "Robb
22      }
23  }
```


© H. Peter Gumm, Philipps-Universität Marburg

Einfache Lösung

- Lock auf die ganze Klasse
  - statisches Objektfeld
  - dient nur als Spender eines Locks
  - `public static Object syncObject=new Object();`

- Verbesserungswürdig, weil für eine Überweisung nicht immer die ganze Klasse gesperrt werden sollte

```
    void überweisen(Konto nach, int betrag) {
        synchronized (syncObject) {
            if (stand > betrag && betrag > 0) {
                this.abheben(betrag);
                nach.einzahlen(betrag);
                assert (this.stand + nach.stand == 200) : "Robbery
            }

        }
    }
```


© H. Peter Gumm, Philipps-Universität Marburg