# Nebenläufige Programmierung I

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer und Axel Rauschmayer

# Ziele

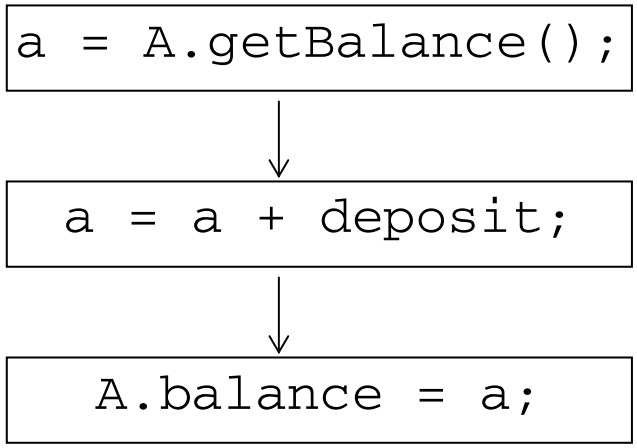
- Grundlegende Begriffe der nebenläufigen Programmierung verstehen lernen
- Nebenläufige Programme in Java schreiben lernen

# Nebenläufige und verteilte Systeme

- Ein **System** ist von seiner Umgebung abgegrenzte Anordnung von Komponenten.
- Können in einem System mehrere Aktivitäten gleichzeitig stattfinden, spricht man von einem **nebenläufigen (parallel ablaufenden) System**.
- Ist das System aus räumlich verteilten Komponenten aufgebaut, spricht man von einem **verteilten System**.
- Programme, wie wir sie bisher geschrieben haben, arbeiten **sequentiell**.
- In einem **nebenläufigen System** laufen mehrere Kontrollflüsse gleichzeitig ab und können miteinander kommunizieren.

# Beispiel: Banktransaktion „single-threaded“

Der Stand eines Konto wird abgefragt, eine Summe „deposit“ eingezahlt, die Kontostandartvariable um einen Betrag „deposit“ erhöht und der neue Wert als neuer Kontostand eingetragen:




// erfragte Kontostand a von Konte A

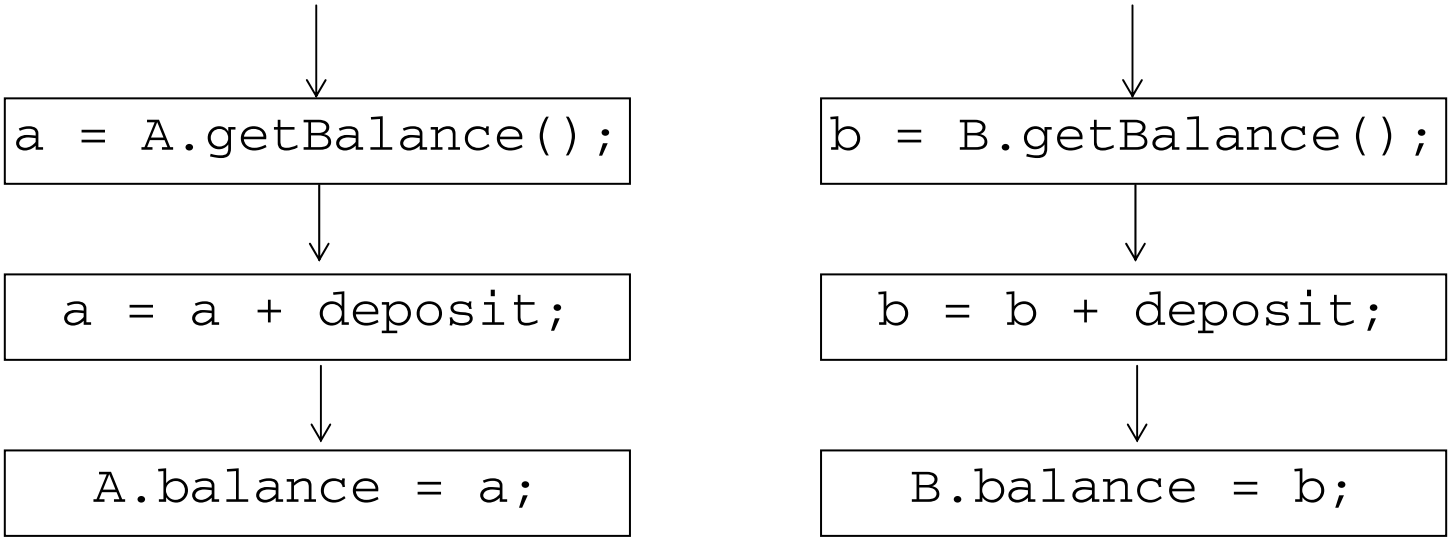
// erhöhe a

// setze Kontostand auf neuen Wert von a

**Bemerkung**: Auch reale Bankautomaten und Computerprogramme laufen in solchen Sequenzen ab, die man „Thread“ (Kontrollfluss) nennt. Das obige Programm ist ein „**single-threaded**“ Programm.

# Beispiel: Banktransaktion „single-threaded“

In einer realen Bank können Kontoeinzahlungen parallel laufen:

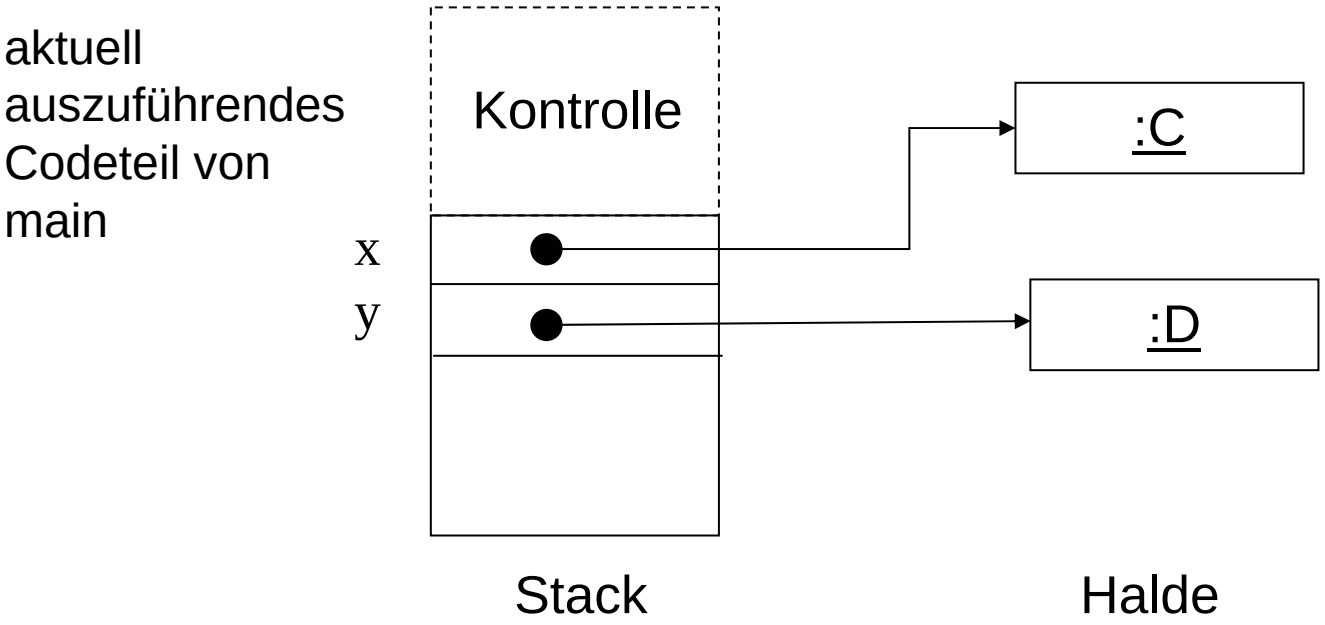



**Bemerkung**: Innerhalb eines Computers wird dies „multi-threading“ genannt. Ein Thread kann eine Aufgabe unabhängig von anderen Threads erfüllen. Ebenso wie zwei Bankautomaten die gleichen Konten benutzen können, können Threads Objekte gemeinsam benutzen.

# Thread

- Aus Betriebssystemsicht wird ein **Prozess** verstanden als eine **abstrakte Maschine, die eine Berechnung ausführt**, d.h. ein Prozess ist ein Programm zusammen mit den Speicherbereichen, in denen es abläuft.
- Ein **Thread** („Faden“) ist ein **„leichtgewichtiger Prozess“,** der einen eigenen Keller, aber keine eigene Halde besitzt.
- Er ist Teil eines Programms, das unabhängig von anderen Teilen des Programms ausgeführt werden kann, und repräsentiert eine einzige Sequenz von Anweisungen, d.h. einen **sequentiellen Kontrollfluss**, der **nebenläufig zu anderen Threads** ausgeführt werden kann und der Daten der **Halde** (Instanzvariablen, statische Variablen und Elemente von Reihungen) **gemeinsam mit anderen Threads benutzt**.

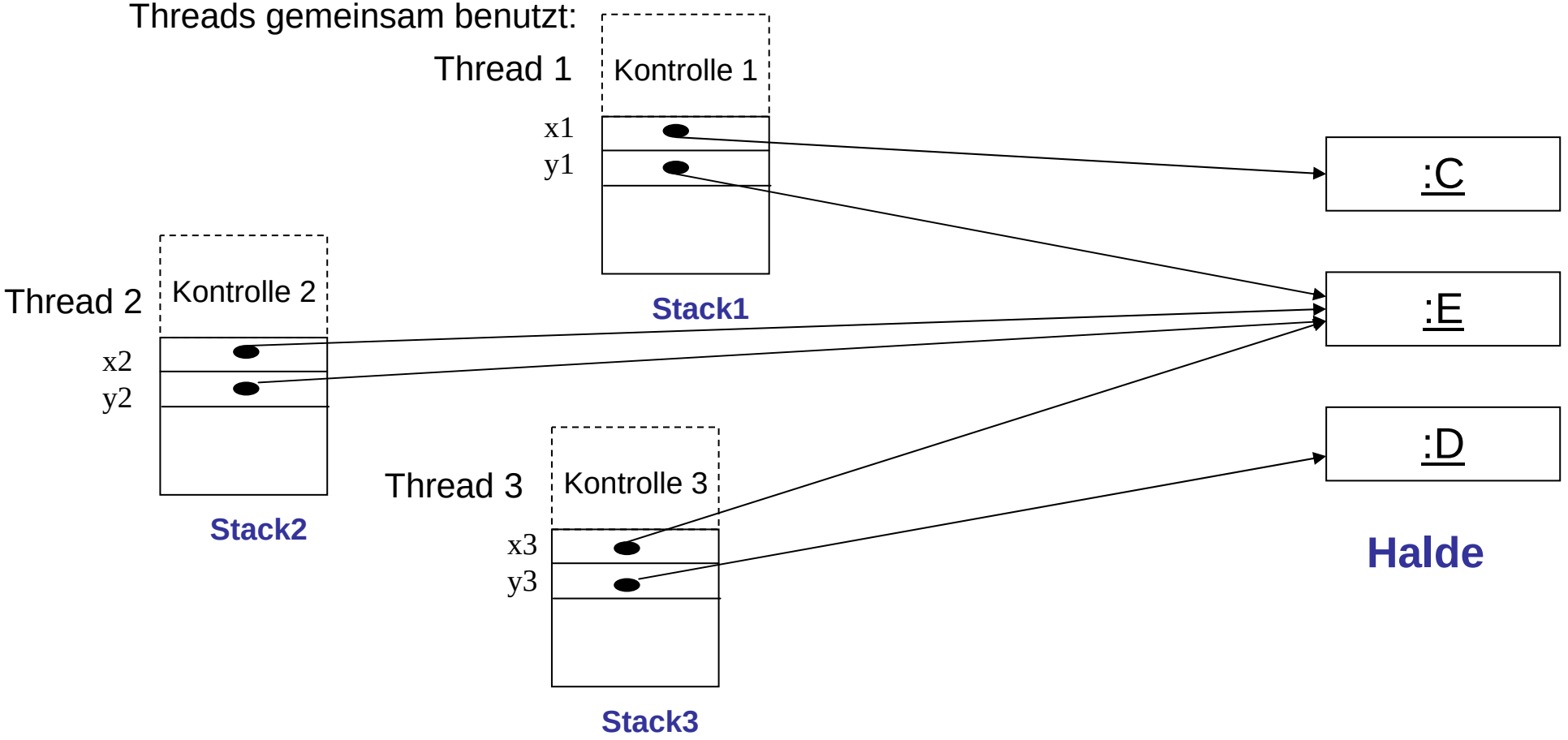
# Sequentielles Speichermodell von Java




Hier entspricht die Ausführung von main einem (single-)Thread.

# Speichermodell für nebenläufige Programme

Ein Thread ist ein „leichtgewichtiger“ Prozess, der die Halde mit anderen Threads gemeinsam benutzt:



# Variablen in nebenläufigen Prozessen

- **Lokale Variable, formale Parameter und Parameter von Ausnahmebehandlern:**
  - sind **lokal** für jeden Thread;
  - haben **keinen Einfluss auf andere Threads**.
- **Instanzvariable, statische Variable, Elemente von Reihungen:**
  - residieren in einem **globalen Speicher**.
  - Änderungen können die **Werte anderer Threads beeinflussen**.

# Scheduling

- Wenn ein Programm mehrere Threads enthält, wird ihre Ausführung durch das Java-Laufzeitsystem geregelt.
- Der „Scheduler“ unter Windows NT implementiert eine „Zeitschrittstrategie“, bei der jeder Thread eine gewisse Prozessorzeit zugeteilt bekommt und dann unterbrochen und in einer Warteschlange hintenangestellt wird.
- Ältere Implementierungen haben ein solches Verfahren nicht. Ein arbeitender Thread wird nicht unterbrochen und behält den Prozessor, bis er terminiert oder sich selbst in einen Wartezustand bringt.
- Die Strategien der Implementierungen können also verschieden sein. Deshalb darf man sich im Programm **nicht** auf ein bestimmtes Verfahren verlassen.

# Zustände eines Threads

**rechnend:** Der Thread wird im Prozessor ausgeführt.

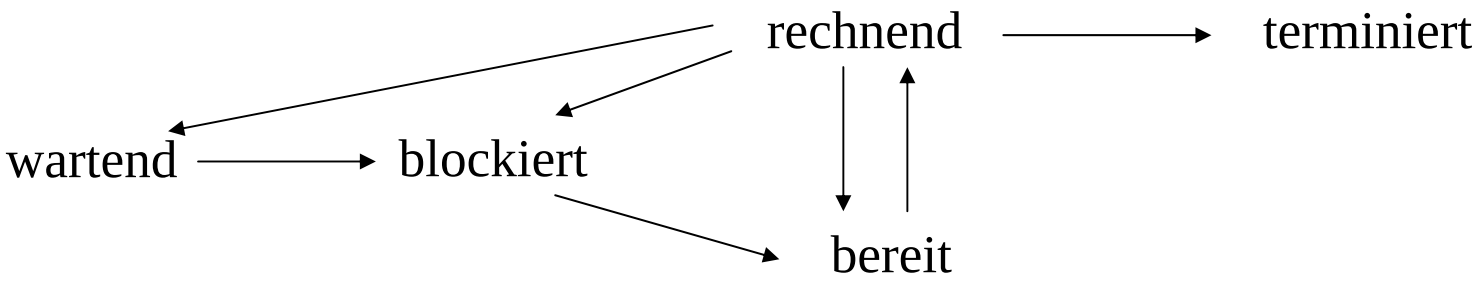
**bereit:** Der Thread ist rechenbereit, hat aber nicht Zugriff auf den Prozessor.

**blockiert:** Der Thread braucht ein Betriebsmittel, das temporär nicht vorhanden ist

**wartend:** Der Thread hat ein Betriebsmittel aufgegeben und wartet.

**terminiert:** Der Thread ist beendet (in Java: dead).

In Java werden die Zustände „rechnend“ und „bereit“ zu einem Zustand „runnable“ und die Zustände „blockiert“ und „wartend“ zu „not runnable“ zusammengefasst.



# Die Klasse java.lang.Thread

Ein **Thread in Java** wird repräsentiert durch eine **Instanz der Klasse Thread** (oder einer Subklasse von Thread).

**Konstruktoren:**

```
Thread ()                 //erzeugt einen neuen Thread
Thread (String name)      //erzeugt einen neuen Thread mit Namen name
Thread (Runnable r)       //erzeugt einen neuen Thread, der die run-Methode von r ausführt
```

**Methoden:**

```
static Thread currentThread()      //l iefert eine Referenz auf das
                                   // gerade ausgeführte Thread-Objekt
String getName()                   // liefert den Namen des Thread
void run ()                        // beschreibt den Kontrollfluss; soll nicht direkt aufgerufen werden,
                                   //da dann nur der Rumpf ausgefuehrt, aber kein Thread gestartet wird
void start ()                      // startet einen Thread und ruft run() auf
static void sleep(long s)          // unterbricht den gerade ausgeführten Thread um s msec
static void yield ()               // versetzt den gerade ausgeführten Thread in den Zustand “bereit”
void interrupt()                   //unterbricht den Thread
```

# Erzeugung von Threads

2 Techniken:

- als Objekt einer Erbenklasse von Thread; in dieser wird `start()` geerbt und `run()` überschrieben.
- durch Implementierung der Schnittstelle `Runnable`; einem neuen Thread wird diese Implementierung übergeben.

# Erzeugen von Threads in Erben von Thread

**Schema:**

```
class C extends Thread
{...
   public void run()
   {<<bestimmt den Kontrollfluss des Thread>>
   }
   public static void main (String[] args)
   { C t = new C();
     t.start(); ...
   }
```

ruft `run()` auf

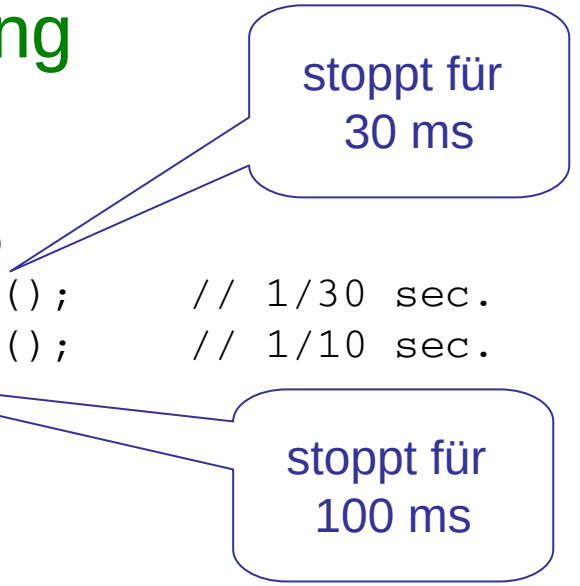
# Beispiel: PingPong

```
public class PingPong extends Thread
{ private String word;          // das zu druckende Wort
  private int delay;            // der Unterbrechungszeitraum

  public PingPong(String whatToSay, int delayTime)
  { word = whatToSay;
    delay = delayTime;
  }
  public void run()
  { try
    { while (true)              // unendliche Schleife
        {       System.out.println(word + " ");
                sleep(delay); //auf das n"achste Mal warten
        }
    }
    catch(InterruptedException e)
    {  System.out.println("Fertig "+ Thread.currentThread().getName());
    }
  }
```

# Beispiel: PingPong

```
        public static void main(String[] args)
        {       new PingPong("ping", 33).start();     // 1/30 sec.
                new PingPong("PONG",100).start();     // 1/10 sec.
        }
}
```




**Bemerkung:**

- Die `run`-Methode von Ping-Pong terminiert nicht und muss deshalb explizit abgebrochen werden (z.B. mit der `InterruptedException` von `sleep()`.
- Mögliche Ausgabe

```
ping PONG ping ping PONG ping ping ping PONG...
```

# Threads unterbrechen

- Die Methode `interrupt()` unterbricht den aktuellen Thread;
- Befindet sich der Thread “im Schlaf” (aufgrund von `sleep`), so wird die Ausnahme `InterruptedException` ausgelöst.
- Man kann dies dazu nutzen, den Thread auf Wunsch von außen vernünftig zu beenden.
- **Beispiel:** Mehrere Threads suchen in unterschiedlichen Datenbanken. Hat einer das Gesuchte gefunden, so kann er die anderen unterbrechen und auffordern, nach ordentlichem Verlassen der Datenbank zu terminieren.
- Die Klasse WatchDog definiert einen Thread der zur Unterbrechung von PingPong-Threads verwendet werden kann (siehe nächste Folie)

# Threads unterbrechen: Beispiel

```
public static void main(String[] args) {
  Thread t1 = new PingPong("ping", 33); // 1/30 sec.
  Thread t2 = new PingPong("PONG", 100); // 1/10 sec.
  t1.start(); t2.start();
  new WatchDog(t1).start();
  new WatchDog(t2).start();}
```

**wobei**

```
public class WatchDog extends Thread{
  private Thread t;

  public WatchDog(Thread t){ this.t = t;}
  public void run(){
    try {
         sleep(500);
         t.interrupt();
    } catch (InterruptedException e){}
  }}
```

# Erzeugen von Threads mit `Runnable`

- Eine Subklasse von `Thread` kann nur von `Thread` (oder einer Subklasse davon) erben.
- Mit Hilfe der Schnittstelle `Runnable` kann ein Thread auch von anderen Klassen erben:

```
public interface Runnable
{   void run();
}
```

- Mit den Konstruktoren (der Klasse Thread)

```
public Thread (Runnable r)
public Thread (Runnable r, String name)
```

konstruiert man einen neuen Thread, der die `run`-Methode von `r` verwendet, `name` gibt einen Namen für den Thread an.

# Schematisches Aufbau eines Runnable Threads

**Schema:**

```
class ConcreteRunnable implements Runnable {
        ...
        public void run()
        {               //Code für den Kontrollfluss
        }
        public static void main (...)
        {  Runnable r = new ConcreteRunnable(...);
                                        //neues Runnable-Objekt
           Thread t = new Thread (r); // neues Thread-Objekt
           t.start();                 // Aufruf von r.run();
        }
}
```

oder
ConcreteRunnable

# Beispiel: PingPong

```
public class RunPingPong implements Runnable
{ …
  public void run() {
    try { …
    } catch(InterruptedException e){
        System.out.println("Fertig "+ Thread.currentThread().getName());}
  public static void main(String[] args){
    Runnable ping = new RunPingPong("ping", 33);
    RunPingPong pong = new RunPingPong("PONG",100);
    new Thread(ping, „Thread ping“).start();
    new Thread(pong, „Thread pong“).start(); }}
```

**Bemerkung:**

- Die Klasse `Thread` und damit auch die Erbenklasse `RunPingPong` implementieren die Schnittstelle `Runnable`.
- Durch Angabe des Threadnamens im Konstruktor wird dieser an Stelle des Standardnamens bei der Unterbrechung ausgegeben

# Probleme bei nebenläufiger Programmierung

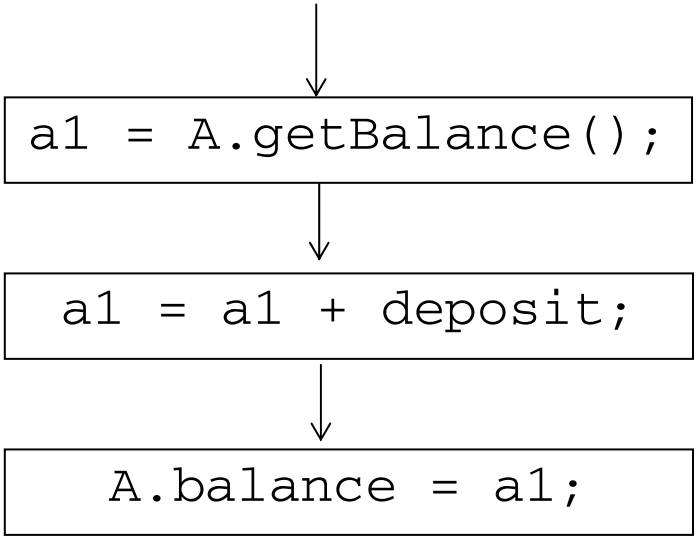
**Beispiel:**

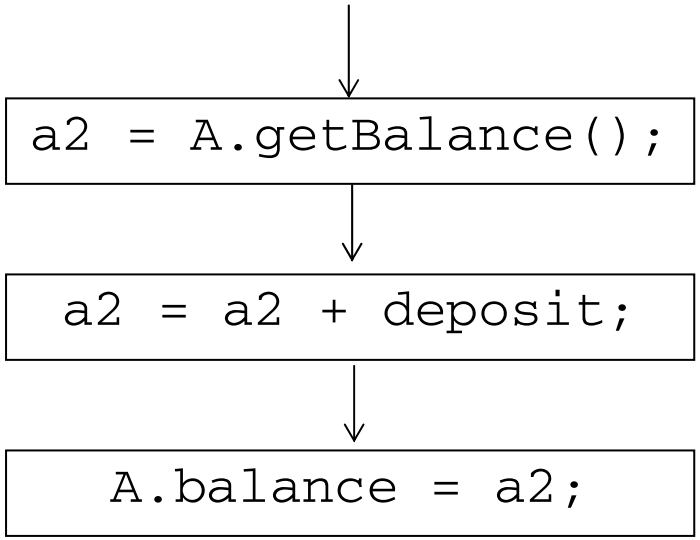
Das folgende (unschöne) Programm für `deposit()` spiegelt das Verhalten eines Bankangestellten wider:

```
void strangeDeposit(double d)
{
   double hilf = getBalance();
   hilf = hilf + d;
   balance = hilf;
}
```

# Probleme bei nebenläufiger Programmierung

Wenn zwei Kunden gleichzeitig auf das gleiche Konto einzahlen wollen, kann der Kontostand von der Reihenfolge der Einzahlung abhängen:




Jeder Bankangestellte geht zum Kontobuch, sieht den Wert nach und geht dann später wieder hin, um den neuen Wert einzutragen. Damit ist der erste Eintrag verloren. ⇒ „Race Condition“, Interferenz zwischen Threads

# Race Condition Beispiel in Java

```
public class BankAccount extends Thread{
  private double balance;
  public BankAccount() { balance = 0; }
  public double getBalance() { return balance; }
  public void slowdeposit(double amount) {
      try{
       System.out.println(currentThread().getName()+" depositing "+amount);
       double newBalance = getBalance() + amount; sleep(100);
       balance = newBalance;
       System.out.println("New balance is: " + newBalance);
      }catch (InterruptedException e){} }}
public class DepositThread extends Thread {
  private BankAccount account; private double amount;
  DepositThread(BankAccount acc, double a) {account = acc; amount = a;}
  public void run() { account.slowdeposit(amount); }
  public static void main(String[] args) {
      BankAccount account = new BankAccount();
      DepositThread th1 = new DepositThread(account, 100);
      DepositThread th2 = new DepositThread(account, 100);
      th1.start();  th2.start();
  }}
```

# Race Condition Beispiel in Java

```
public class BankAccount extends Thread{
  private double balance;
  public BankAccount() { balance = 0; }
  public double getBalance() { return balance; }
  public void slowdeposit(double amount) {
    try{
     System.out.println(currentThread().getName()+" depositing "+amount);
     double newBalance = getBalance() + amount; sleep(100);
     balance = newBalance;
     System.out.println("New balance is: " + newBalance);
    }catch (InterruptedException e){} }}
public class DepositThread extends Thread {
  private BankAccount account; private double amount;
  DepositThread(BankAccount acc, double a) {account = acc; amount = a;}
  public void run() { account.slowdeposit(amount); }
  public static void main(String[] args) {
    BankAccount account = new BankAccount();
    DepositThread th1 = new DepositThread(account, 100);
    DepositThread th2 = new DepositThread(account, 100);
    th1.start();  th2.start();
}}
```

# Race Conditions

**Als Ergebnis erhält man:**

```
Thread-0 depositing 100.0
Thread-1 depositing 100.0
New balance is: 100.0
New balance is: 100.0
```

Eine Einzahlung ging verloren …

Man erhält eine „Race Condition“, d.h. eine Interferenz zwischen Threads

- Eine **Race Condition** ist eine Situation, in der zwei oder mehr Threads gemeinsame Daten schreiben oder lesen und das Endergebnis vom Scheduling der Threads abhängt.
- Race Conditions sind gefürchtet, da sie zu **schwer vorhersagbarem und schwer reproduzierbarem Programmverhalten** führen.
- Eine Race Condition manifestiert sich oft nur sehr selten (etwa 1 Mal pro 1000 Programmabläufe) und ist daher **durch Testen kaum aufzuspüren**.
- Um „Race Conditions“ zu vermeiden, hat man früher eine Bemerkung auf das Konto A geschrieben: „Ich arbeite an dem Konto“. **In Java** verwendet man „**synchronized**“

# Sicherheit durch Synchronisation

- Interaktion zwischen Threads kann zu unerwünschtem Verhalten führen.
- Man unterscheidet zwei Arten von Eigenschaften nebenläufiger Programme:
  - Sicherheit:
    - Eigenschaft, dass nie etwas Unerwünschtes geschieht
  - Lebendigkeit:
    - Eigenschaft, dass immer irgendetwas geschieht
- Sicherheitsfehler führen zu unerwünschtem Laufzeitverhalten
- Lebendigkeitsprobleme führen dazu, dass das Programm anhält, obwohl noch nicht alle Threads terminiert haben.

# Sicherheit und Nebenläufigkeit

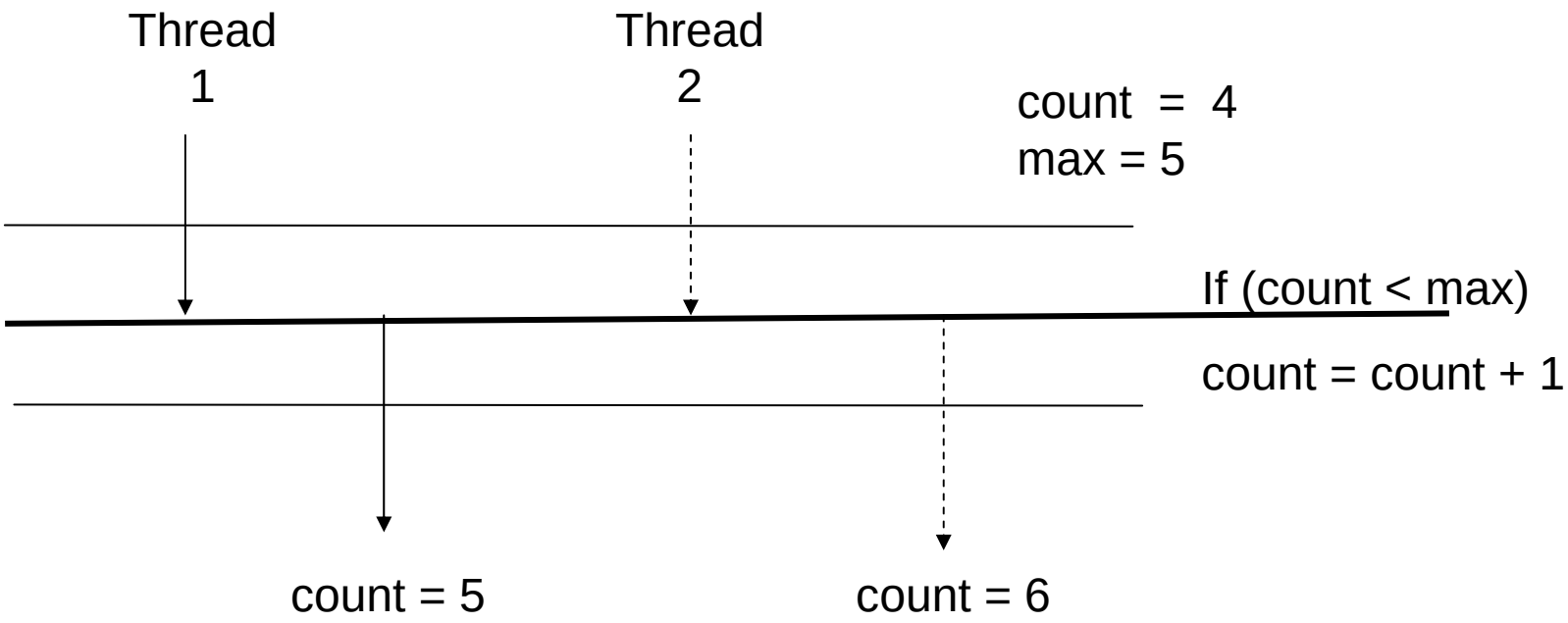
- Sicherheitsprobleme basieren häufig auf
  - **Lese / Schreibkonflikten**

    Der Zugriff eines lesenden Kunden auf ein Objekt während eines noch nicht beendeten Schreibvorgangs kann zum lesen inkorrekter Daten führen

    **Beispiel**: Lesen einer Reihung während alle Feldelemente verdoppelt werden
  - **Schreib/Schreibkonflikte**

    Inkonsistente Zuweisung an Variablen durch nebenläufiges Ausführen von zustandsändernden Methoden (siehe Bankangestellten-Beispiel)

# Beispiel: Schreib/Schreib-Konflikt (ohne Synchronisation)



# Beispiel: Schreib/Schreib-Konflikt (ohne Synchronisation) - Ablauf

- Die lokalen Variablen haben die Werte: count = 4, count = 5
- Thread 1 führt den ersten Teil der Anweisung aus

  if (4 < 5) (also count < max)
- Thread 2 führt den ersten Teil der Anweisung aus

  if (4 < 5) (also count < max)
- Thread 1 führt den 2. Teil der Anweisung aus (count ++)

  Ergebnis: count = 5 (= 4 + 1)
- Thread 2 führt den 2. Teil der Anweisung aus (count ++)

  Ergebnis: count = 6 (= 5 + 1) d.h.
  count>max, obwohl count nicht größer als max hätte sein sollen

# Synchronisation von Methoden

- Im Bankbeispiel kann man die Race Condition dadurch vermeiden, dass man die Methode `slowDeposit` mit dem Schlüsselwort **`synchronized`** kennzeichnet:

```
public synchronized void slowDeposit(double amount) { ... }
```

- **Achtung:** Es ist keine Lösung, `deposit` irgendwie "atomar" zu schreiben:

```
System.out.println("Depositing " + amount + "\n" +
"New balance is: " + balance+=amount):
```

Ein zusammengesetztes Statement wie letzteres wird nämlich in mehrere Bytecode - Befehle übersetzt und diese können auch wieder mit anderen Befehlen verzahnt werden.

# Synchronisation von Methoden

- Jedes Objekt ist mit einem **Monitor** ausgestattet. Dieser Monitor beinhaltet eine Boole'sche Variable und eine Warteschlange für Threads.
- Ruft ein Thread eine `synchronized` Methode auf, so wird zunächst geprüft, ob die assoziierte Boole'sche Variable des Objekts `true` (="frei") ist.
  Falls ja, so wird die Variable auf `false` (="besetzt") gesetzt.
  Falls nein, wird der aufrufende Thread *blockiert* und in die Warteschlange eingereiht.
- Verlässt ein Thread eine `synchronized` Methode, so wird zunächst geprüft, ob sich wartende Threads in der Schlange befinden.
  Falls ja, so darf deren erster die von ihm gewünschte Methode ausführen.
  Falls nein, wird die mit dem Objekt assoziierte Boole'sche Variable auf true (="frei") gesetzt.
- **Vergleich:** Für jedes Objekt gibt es einen Schlüssel. Nur, wer den Schlüssel in der Hand hält, kann bei dem Objekt synchronisierte Methoden aufrufen.
- **Bemerkung**: Das Monitorkonzept wurde von **C.A.R. Hoare** 1978 eingeführt.

# Synchronisation von Methoden

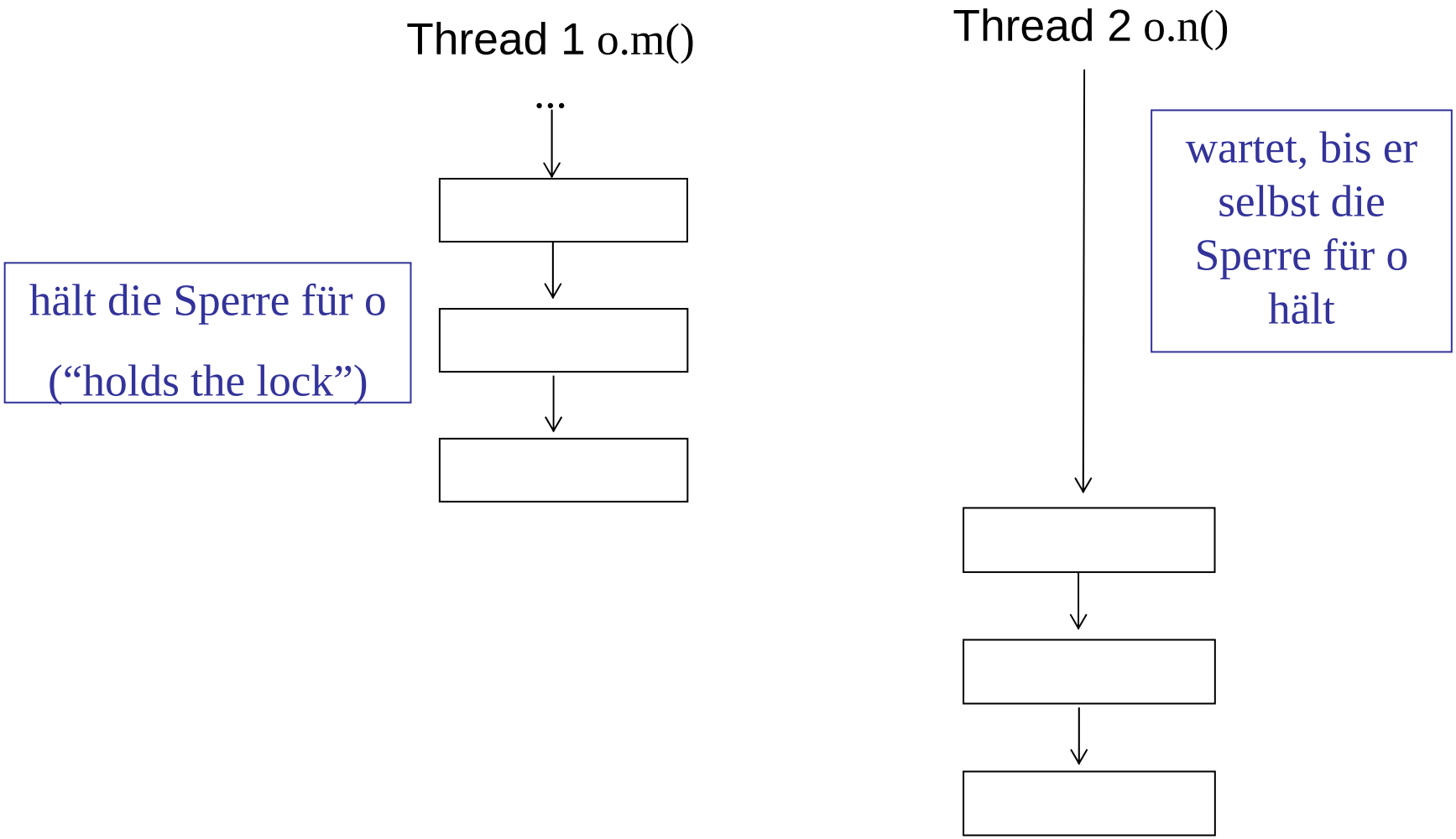
Vermeidung von „Race Conditions“ durch Synchronisation:

```
synchronized  T m(T1x)
```

Nicht synchronisierte Methoden können auf gesperrte Objekte zugreifen

- Wenn ein Thread `o.m(a)` aufruft, wird eine Sperre auf `o` gesetzt.
- Jeder andere Thread, der `o` mit einer synchronized-Methode aufruft, wird blockiert, bis die Sperre (nach Beendigung der Ausführung von `o.m(a))` aufgehoben wird.
- Durch Synchronisation erreicht man folgende **Sicherheitseigenschaft**: Es können niemals zwei Threads gleichzeitig auf einem gemeinsamen Datenbereich zugreifen.

# Beispiel: Wirkung der Synchronisation



# Bemerkungen

- **Wechselseitiger Ausschluss:**

  Man spricht von wechselseitigem Ausschluss, wenn zu jedem Zeitpunkt nie mehr als ein Prozess auf ein Objekt zugreifen kann.

- Statische Methoden können auch synchronisiert werden.

- Wird eine synchronisierte Methode in einem Erben überschrieben, so kann sie, muss aber nicht synchronisiert sein.

# Beispiel: BankAccount

Die folgende Account-Klasse synchronisiert die Operationen und garantiert so, dass während der Ausführung von `slowDeposit` keine andere synchronisierte Operation auf das aktuelle Konto zugreifen kann.

```
public class SyncBankAccount
{ ...
   public synchronized double getBalance()
   { ... }
   public synchronized void slowDeposit(double d)
   {   ...
   }
}
```

# Zusammenfassung

- Java unterstützt Nebenläufigkeit durch „leichtgewichtige“ Prozesse, sogenannte **Threads**, die über die gemeinsame Halde und damit über gemeinsam benutzte Objekte miteinander kommunizieren.
- Nebenläufigkeit wirft **Sicherheits- und Lebendigkeitsprobleme** auf. Gemeinsam benutzte Objekte müssen synchronisiert werden.