

# Kapitel 14: Threads

- Einführung
- Klasse Thread und Interface Runnable
- Methode join und Parallelisierung von Algorithmen
- Synchronisierung mit synchronized
- Erzeuger/Verbraucher-Problem und die Methoden wait, notify und notifyAll
- Zustände eines Java-Threads
- Thread-sichere Typen in der Java API

# Einführendes Beispiel

- Definition einer Thread-Klasse
- Jeder Thread durchläuft den in der **run-Methode** definierten Code.
- Es werden 5 Thread-Objekte definiert, die mit **start()** **nebenläufig** gestartet werden.
- Der **start-Aufruf** eines Threads bewirkt seinen **run-Aufruf**.
- Auch die main-Methode läuft als eigener Thread.
- Damit laufen 6 Threads nebenläufig.

```
class MyThread extends Thread {  
    public MyThread(String name) {  
        super(name);  
    }  
    @Override  
    public void run() {  
        for (int i = 0; i < 100; i++)  
            System.out.println(this.getName() + ": " + i);  
    }  
}
```

```
class ThreadApplication {  
    public static void main(String[] args) {  
        for (int i = 0; i < 5; i++) {  
            Thread t = new MyThread("MyThread " + i);  
            t.start();  
        }  
        System.out.println("Main ist fertig");  
    }  
}
```

```
MyThread 0: 0  
MyThread 3: 0  
MyThread 3: 1  
MyThread 3: 2  
MyThread 3: 3  
...  
MyThread 3: 24  
MyThread 3: 25  
MyThread 2: 0  
MyThread 2: 1  
MyThread 2: 2  
...  
MyThread 0: 4  
MyThread 0: 5  
MyThread 0: 6  
Main ist fertig  
MyThread 0: 7  
MyThread 0: 8  
...
```

Konsolen-  
ausgabe

# Threads und Nebenläufigkeit

---

- Ein **Thread** ist eine Folge von Anweisungen, die nebenläufig ausgeführt werden können.
- **Nebenläufigkeit (concurrency)** bedeutet:
  - (echte) **Parallelität**:  
die Threads laufen auf verschiedenen Prozessoren gleichzeitig ab.
  - **Pseudo-Parallelität**:  
die Threads laufen auf genau einem Prozessor ab, wobei die Threads mit einer hohen Taktrate ständig gewechselt werden.  
Es wird eine Gleichzeitigkeit vorgetäuscht.
- Jeder Thread besitzt einen **eigenen Laufzeitkeller** (Stack) für Methodenaufrufe und Speicherung lokaler Variablen.
- Wichtig: die Threads können Zugriff auf gemeinsame Daten haben.  
Dazu muss der Zugriff geeignet synchronisiert werden (später).

# Erzeugung von Threads durch Erweiterung der Klasse Thread

---

- Die Klasse `Thread` aus `java.lang` wird erweitert, indem die Methode `run()` überschrieben wird.
- Der Aufruf der Methode `start()` der Klasse `Thread` bewirkt, dass die Java Virtual Machine die `run`-Methode als Thread nebenläufig ausführt.

```
class MyThread extends Thread {  
    @Override  
    public void run() {  
        // mein Code: ...  
    }  
}
```

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

# Erzeugung von Threads durch Implementierung des Interface Runnable

---

- Das **Interface Runnable** aus `java.lang` enthält nur die Methode `run()`. Runnable ist ein **funktionales Interface**.
- Das Interface Runnable wird durch eine eigene Runnable-Klasse implementiert.
- Ein Thread lässt sich dann mit Hilfe eines **Thread-Konstruktors** definieren, indem ein Objekt der Runnable-Klasse als Parameter übergeben wird.
- Das Thread-Objekt wird dann mit der Methode `start()` gestartet.

```
@FunctionalInterface
interface Runnable {
    void run();
}
```

```
class MyRunnable implements Runnable {
    public void run() {
        // mein Code: ...
    }
}
```

```
class ThreadApplication {
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start();
    }
}
```

# Runnable-Objekte als Lambda-Ausdrücke

---

- Da Runnable ein funktionales Interface ist, dürfen Lambda-Ausdrücke als Runnable-Objekte verwendet werden.
- Damit ist eine prägnante Schreibweise möglich:

```
Runnable myRun = () -> System.out.println("myRun läuft");  
Thread t = new Thread(myRun);  
t.start();
```

- Noch kürzer:

```
new Thread( () -> System.out.println("myRun läuft") ).start();
```

# Kapitel 14: Threads

- Einführung
- Klasse Thread und Interface Runnable
- Methode join und Parallelisierung von Algorithmen
- Synchronisierung mit synchronized
- Erzeuger/Verbraucher-Problem und die Methoden wait, notify und notifyAll
- Zustände eines Java-Threads
- Thread-sichere Typen in der Java API

# Mit join auf Beendigung von Threads warten

---

- Mit der Methode `join()` der Klasse `Thread` wird solange gewartet, bis der Thread zu Ende gelaufen ist.
- `join` kann eine `InterruptedException` werfen.

```
public void main(String[ ] args) throws InterruptedException {  
    Thread t = new Thread(...);  
    t.start();    // Starte Thread t  
  
    // irgendwelche Berechnungen des main-Threads:  
    // ...  
  
    t.join();    // warte, bis Thread t zu Ende gelaufen ist  
  
    // weitere Berechnungen des main-Threads:  
    // ...  
  
}
```

- Mit dem start-join-Konzept lassen sich sehr einfach Daten-parallele Algorithmen realisieren (d.h. Daten lassen sich in unabhängige Teile zerlegen und nebenläufig bearbeiten).



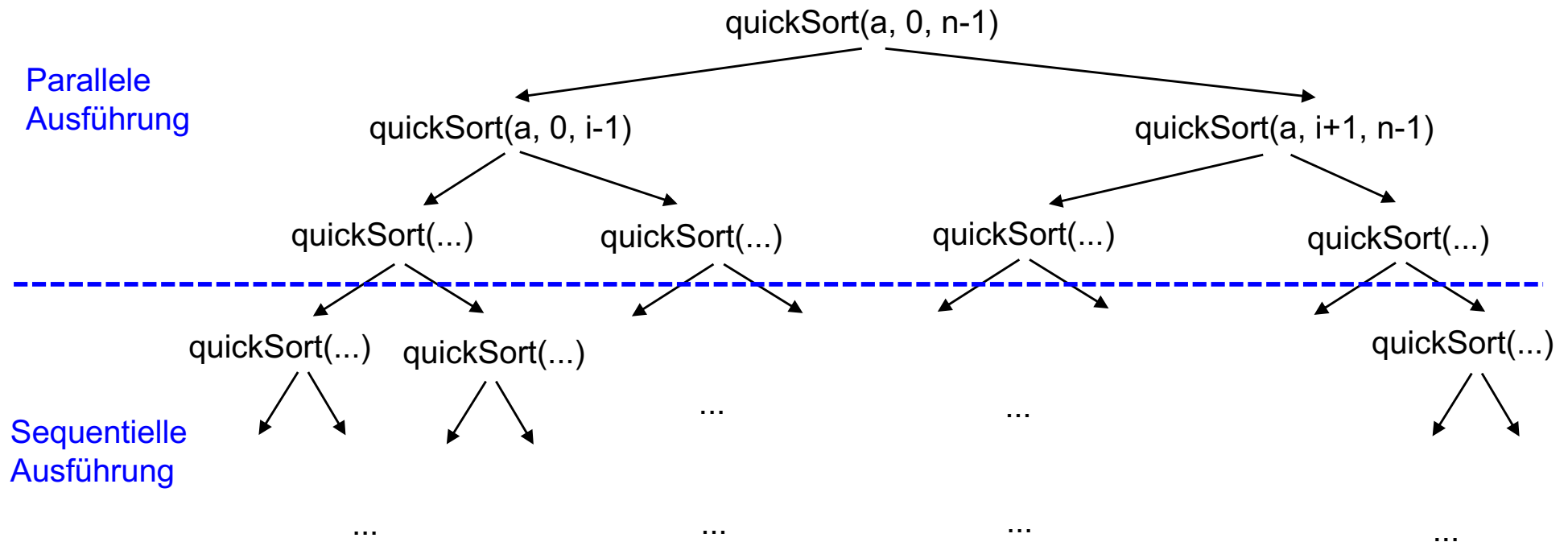
# Beispiel: paralleles Befüllen eines Felds

```
class RandomizeArrayThread extends Thread {  
    private final double[] a;  
    private final int li;  
    private final int re;  
  
    public RandomizeArrayThread (double[] a, int li, int re) {  
        this.li = li;  
        this.re = re;  
        this.a = a;  
    }  
  
    @Override  
    public void run() {  
        for (int i = li; i < re; i++)  
            a[i] = Math.random();  
    }  
}
```

```
public class JoinApplication {  
    public static void main(String[] args)  
        throws InterruptedException {  
  
        int N = 1000;  
        double[] a = new double[N];  
  
        Thread t1 = new RandomizeArrayThread(a, 0, N/2);  
        Thread t2 = new RandomizeArrayThread(a, N/2, N);  
        t1.start();  
        t2.start();  
  
        t1.join();    // Warte bis t1 zu Ende  
        t2.join();    // Warte bis t2 zu Ende  
  
        System.out.println("Alles fertig");  
    }  
}
```

- Die run-Methode befüllt ein Feld a von a[li] bis a[re-1] mit zufälligen Zahlen.
- a, li und re werden als Parameter beim Konstruktor übergeben.
- Der main-Thread startet zwei parallele Threads t1 und t2, die zwei unabhängige Teile des Felds a mit zufälligen Zahlen initialisieren.
- Danach wartet der main-Thread, bis beide Threads t1 und t2 zu Ende gelaufen sind.

# Beispiel: paralleles QuickSort (1)



- Nur QuickSort-Aufrufe bis zur Rekursionstiefe  $d = 2$  einschl. sollen parallel ausgeführt werden.

# Beispiel: paralleles QuickSort (2)

```
public static void sort(int[ ] a) {  
    int maxDepth = 2;  
    Thread sortThread = new QuickSortThread (a, 0, a.length-1, maxDepth );  
    sortThread.start();  
  
    try {  
        sortThread.join();  
    } catch (InterruptedException e) { }  
}
```

Übergeordnete Sortiermethode startet einen Thread und wartet auf sein Ende.

```
class QuickSortThread extends Thread {  
    private int a[ ];  
    private int li;  
    private int re;  
    private int maxDepth ; // Rek.Tiefe, bis zu der parallelisiert wird.  
  
    public QuickSortThread (int[ ] a, int li, int re, int maxDepth ) {  
        this.a = a;  
        this.li = li;  
        this.re = re;  
        this.maxDepth = maxDepth ;  
    }  
  
    public void run() { ... } // nächste Seite  
}
```

Das Runnable-Objekt wird mit den QuickSort-Parametern initialisiert.

# Beispiel: paralleles QuickSort (3)

```
public void run() {  
    if (li >= re) return;  
    int i = partition3Median(a, li, re);  
    if (maxDepth <= 0) {  
        quickSort(a, li, i-1);  
        quickSort(a, i+1, re);  
    } else {  
        Thread tli = null;  
        Thread tre = null;  
        if (li < i - 1) {  
            tli = new QuickSortThread(a, li, i-1, maxDepth-1);  
            tli.start();  
        }  
        if (i + 1 < re) {  
            tre = new QuickSortThread(a, i+1, re, maxDepth-1);  
            tre.start();  
        }  
        if (tli != null)  
            try {tli.join();} catch (InterruptedException e) {}  
        if (tre != null)  
            try {tre.join();} catch (InterruptedException e) {}  
    }  
}
```

Partitionierung mit  
3-Median-Strategie.

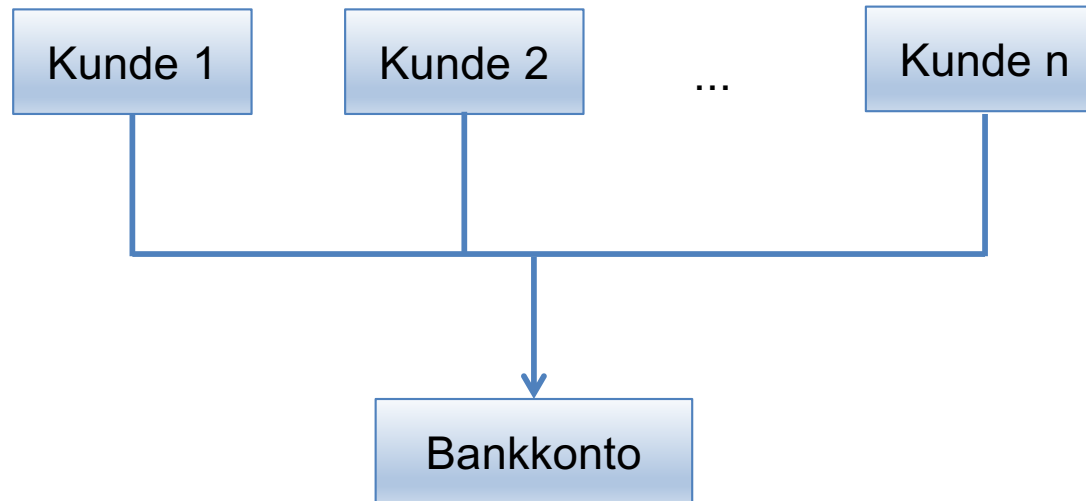
Sequentielles QuickSort.

Paralleles QuickSort.

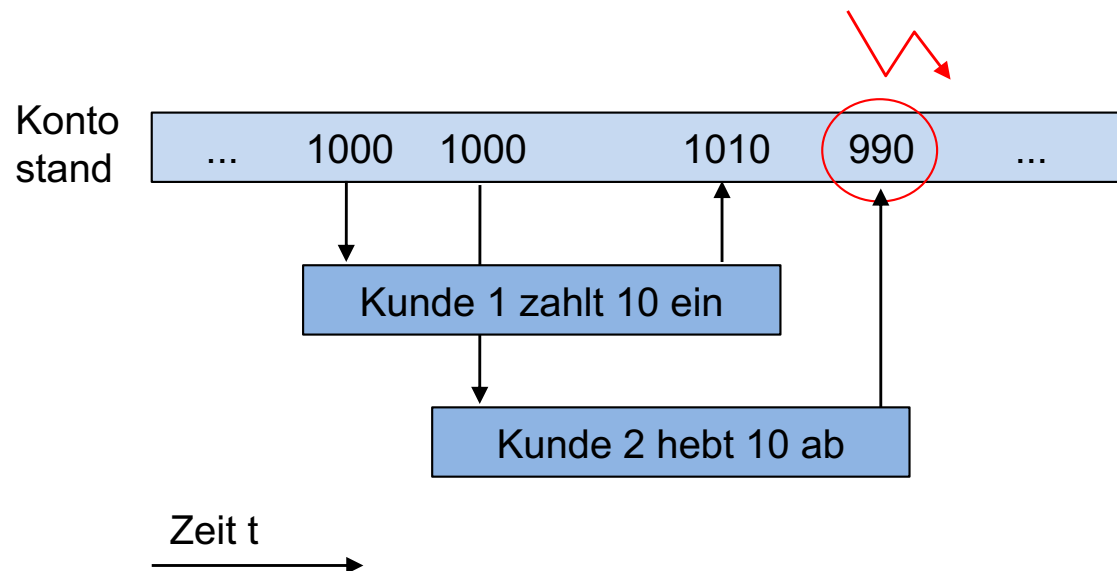
# Kapitel 14: Threads

- Einführung
- Klasse Thread und Interface Runnable
- Methode join und Parallelisierung von Algorithmen
- Synchronisierung mit synchronized
- Erzeuger/Verbraucher-Problem und die Methoden wait, notify und notifyAll
- Zustände eines Java-Threads
- Thread-sichere Typen in der Java API

# Problem bei nebenläufigem Zugriff auf gemeinsame Daten



Verschiedene Kunden greifen auf ein gemeinsames Konto zu.



Nebenläufiger Zugriff auf dasselbe Konto kann zu Inkonsistenzen führen

# Problem bei nebenläufigem Zugriff: Beispiel in Java

```
class BankAccount {  
    private int balance;  
    public BankAccount(int initialBalance) {balance = initialBalance; }  
    public int getBalance() {return balance; }  
    public void deposit(int amount) {balance += amount; }  
}
```

Bankkonto mit Startguthaben  
balance = initialBalance.

```
class Customer extends Thread{  
    private BankAccount account;  
    private int amount;  
    public Customer(BankAccount a, int d) { account = a; amount = d; }  
    public void run() {  
        for (int i = 0; i < 1000; i++) account.deposit(amount);  
    }  
}
```

Kunde führt 1000 Buchungen  
durch.

```
public static void main(...) throws InterruptedException {  
    BankAccount a = new BankAccount(1000);  
    Thread kunde1 = new Customer(a, +10);  
    Thread kunde2 = new Customer(a, -10);  
    kunde1.start(); kunde2.start();  
    kunde1.join(); kunde2.join();  
    System.out.println(a.getBalance());  
}
```

Bankkonto mit Startguthaben  
balance = 1000 definieren.

Es werden 2 Kunden gestartet.  
Kunde 1 zahlt 1000-mal 10 ein.  
Kunde 2 hebt 1000-mal 10 ab.

**Kontostand hat nicht immer den  
erwarteten Wert balance = 1000!**

# Synchronisierung mit synchronized-Methode

- Bei Eintritt in eine **synchronized-Methode** wird das Objekt **gesperrt** und bei Austritt wieder **freigegeben** (**locking Mechanismus**)
- Zu einem Zeitpunkt darf daher höchstens ein Thread auf ein gemeinsames Objekt mit einer **synchronized-Methode** zugreifen.
- Der Thread, der ein gesperrtes Objekt bearbeiten möchte, wird **blockiert**, bis das Objekt wieder freigegeben wird.
- Beachte: auf verschiedene Objekte darf gleichzeitig zugegriffen werden.

```
class GemeinsameDaten {  
    ...  
    public synchronized ... zugriff1(...) { ... }  
    public synchronized ... zugriff2(...) { ... }  
    ...  
}
```

```
GemeinsameDaten data = new GemeinsameDaten();
```

Thread 1 greift auf data zu

Thread 2 greift auf data zu

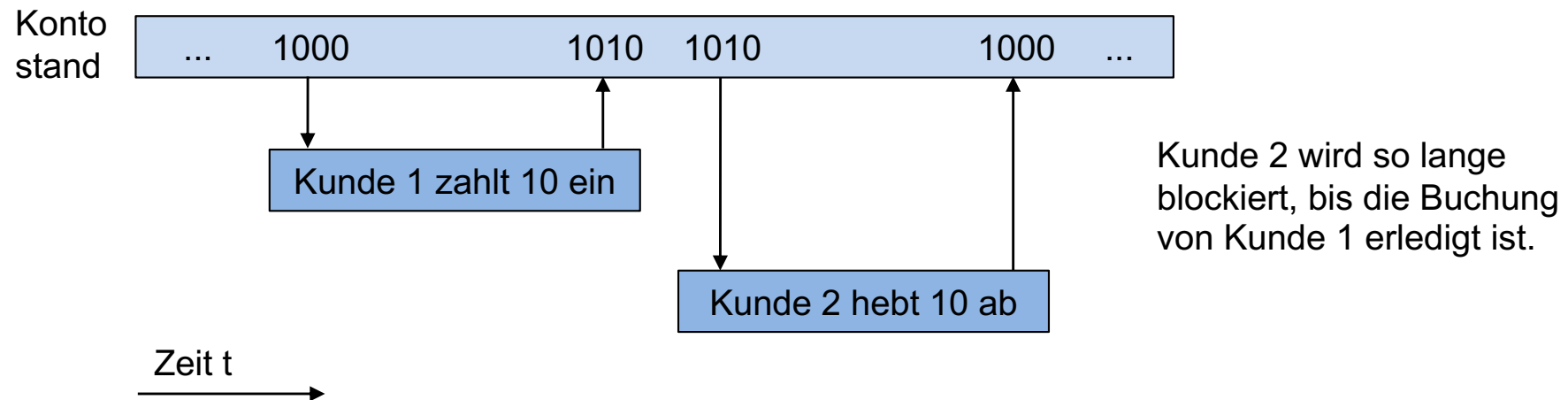
Zeit t  
→

Threads greifen auf  
gemeinsame Daten  
nicht gleichzeitig zu!



# Beispiel mit synchronized in Java

```
class BankAccount {  
    private int balance = 1000;  
    public BankAccount(int initialBalance) {balance = initialBalance; }  
    public synchronized int getBalance() {return balance; }  
    public synchronized void deposit(int amount) {balance += amount  
}
```



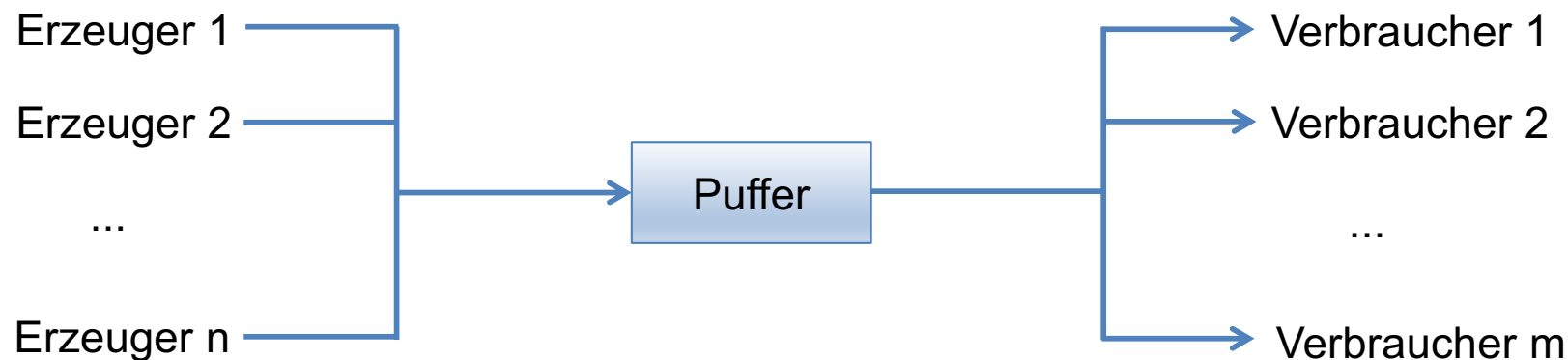
# Kapitel 14: Threads

- Einführung
- Klasse Thread und Interface Runnable
- Methode join und Parallelisierung von Algorithmen
- Synchronisierung mit synchronized
- Erzeuger/Verbraucher-Problem und die Methoden wait, notify und notifyAll
- Zustände eines Java-Threads
- Thread-sichere Typen in der Java API

# Erzeuger/Verbraucher-Problem

---

- Es gibt verschiedene **Erzeuger-Threads**, die Daten erzeugen und in einen Puffer (z.B. eine Queue) schreiben.
- Es gibt verschiedene **Verbraucher-Threads**, die Daten vom Puffer holen und verarbeiten.
- Zugriff auf Puffer muss **synchronisiert** werden.
- Verbraucher-Threads müssen **warten**, falls Puffer leer ist.
- Falls Erzeuger-Threads Daten im Puffer ablegt, dann müssen wartende Verbraucher **benachrichtigt** und aktiviert werden.
- Zusätzlich kann der Puffer begrenzte Kapazität haben, so dass auch Erzeuger eventuell warten müssen und vom Verbraucher benachrichtigt werden müssen.



# Methoden wait, notify, notifyAll

- Mit der Methode `wait` wird ein Thread solange in den **Wartezustand** gesetzt, bis eine Bedingung B erfüllt ist. `wait` erfolgt in einer **Schleife**, da bei Aktivierung des Threads Bedingung erneut geprüft werden muss.
- Mit der Methode `notifyAll` werden **alle wartenden Threads** wieder aktiviert.
- Mit `notify` wird **irgendein wartender Thread** aktiviert.
- `wait` und `notifyAll` (`notify`) sollten in `synchronized`-Methoden aufgerufen werden, da auf gemeinsame Daten zugegriffen wird.
- `wait`, `notify` und `notifyAll` sind in in der **Klasse Object** definiert.
- **Wichtig:** Die hier vorgegebenen Muster für die Benutzung von `wait`, `notify` und `notifyAll` sollten befolgt werden!

```
synchronized void doWhenCondition() {  
    while (! B)  
        wait();  
  
    // Zugriff auf gemeinsame Daten:  
    // ...  
}
```

```
synchronized void changeCondition() {  
    // Zugriff auf gemeinsame Daten:  
    // ...  
  
    // Bedingung B kann sich nun geändert haben.  
    // Daher wartende Threads benachrichtigen,  
    // um Bedingung B neu zu prüfen:  
    notifyAll();    // oder notify();  
}
```

# Beispiel mit Queue (1)

---

- Verschiedene **Erzeuger-Threads** schreiben Daten in eine Queue.
- **Verbraucher-Threads** holen die Daten aus der Queue.
- Verbraucher-Threads müssen **warten (Methode wait)**, falls die Queue leer ist.
- Sobald ein Erzeuger-Thread Daten in die Queue schreibt, wird irgendein Verbraucher mit **notify** aktiviert.



# Beispiel mit Queue (2)

---

```
class BlockingQueue {  
    private final Queue<Integer> myQueue = new LinkedList<>();  
    public synchronized void add(int x) {  
        myQueue.add(x);  
        notify();  
    }  
    public synchronized int remove() throws InterruptedException {  
        while (myQueue.isEmpty())  
            wait();  
        return myQueue.poll();  
    }  
}
```

Nur Verbraucher-Threads können im Warte-Zustand sein.  
Es genügt, irgendein wartenden Verbraucher-Thread zu aktivieren.  
Daher: notify (und nicht notifyAll)

# Beispiel mit Queue (3)

---

Producer-Thread schreibt 100 Zahlen in die BlockingQueue.

```
class Producer extends Thread {  
    private final BlockingQueue bq;  
    private final int start;  
    public Producer(BlockingQueue bq, int s) {  
        this.bq = bq;  
        this.start = s;  
    }  
    public void run() {  
        for (int i = start; i < start+100; i++)  
            bq.add(i);  
    }  
}
```

Consumer-Thread holt 150 Zahlen aus der BlockingQueue und gibt sie aus.

```
class Consumer extends Thread {  
    private final BlockingQueue bq;  
    private final String name;  
    public Consumer(BlockingQueue bq, String n) {  
        this.bq = bq;  
        this.name = n;  
    }  
    public void run() {  
        for (int i = 0; i < 150; i++)  
            try {  
                System.out.println(name + ": " + bq.remove());  
            } catch (InterruptedException ex) { }  
    }  
}
```

# Beispiel mit Queue (4)

---

```
public static void main(String[ ] args) {  
    BlockingQueue bq = new BlockingQueue();  
    Producer p1 = new Producer(bq, 0);  
    Producer p2 = new Producer(bq, 1000);  
    Producer p3 = new Producer(bq, 1000_000);  
    Consumer c1 = new Consumer(bq, "consumer1");  
    Consumer c2 = new Consumer(bq, "consumer2");  
  
    p1.start();  
    p2.start();  
    p3.start();  
  
    c1.start();  
    c2.start();  
  
}
```

Es werden 3 Producer-Thread gestartet, die insgesamt 300 Zahlen in die BlockingQueue schreiben.

Es werden 2 Consumer-Threads gestartet, die insgesamt 300 Zahlen aus der BlockingQueue holen und ausgeben.



# Beispiel mit kapazitätsbegrenzter Queue (1)

---

- Verschiedene **Erzeuger-Threads** schreiben Daten in eine **kapazitätsbegrenzte Queue**.
- **Verbraucher-Threads** holen die Daten aus der Queue.
- Verbraucher-Threads müssen **warten (Methode wait)**, falls die **Queue leer** ist. Sobald ein Erzeuger-Thread Daten in die Queue schreibt, werden **alle wartenden Threads** mit **notifyAll** aktiviert.
- Erzeuger-Threads müssen **warten (Methode wait)**, falls die **Queue voll** ist. Sobald ein Verbraucher-Thread Daten aus der Queue holt, werden **alle wartenden Threads** mit **notifyAll** aktiviert.



# Beispiel mit kapazitätsbegrenzter Queue (2)

```
class RestrictedCapacityBlockingQueue {  
    private final Queue<Integer> myQueue = new LinkedList<>();  
    private final int cap = 5;  
  
    public synchronized void add(int x) throws InterruptedException {  
        while (myQueue.size() >= cap)  
            wait();  
        myQueue.add(x);  
        System.out.println("added: " + myQueue.size());  
        notifyAll();  
    }  
  
    public synchronized int remove() throws InterruptedException {  
        while (myQueue.isEmpty())  
            wait();  
        int x = myQueue.poll();  
        System.out.println("removed: " + myQueue.size());  
        notifyAll();  
        return x;  
    }  
}
```

Hier muss wenigstens ein Consumer-Thread aktiviert werden.

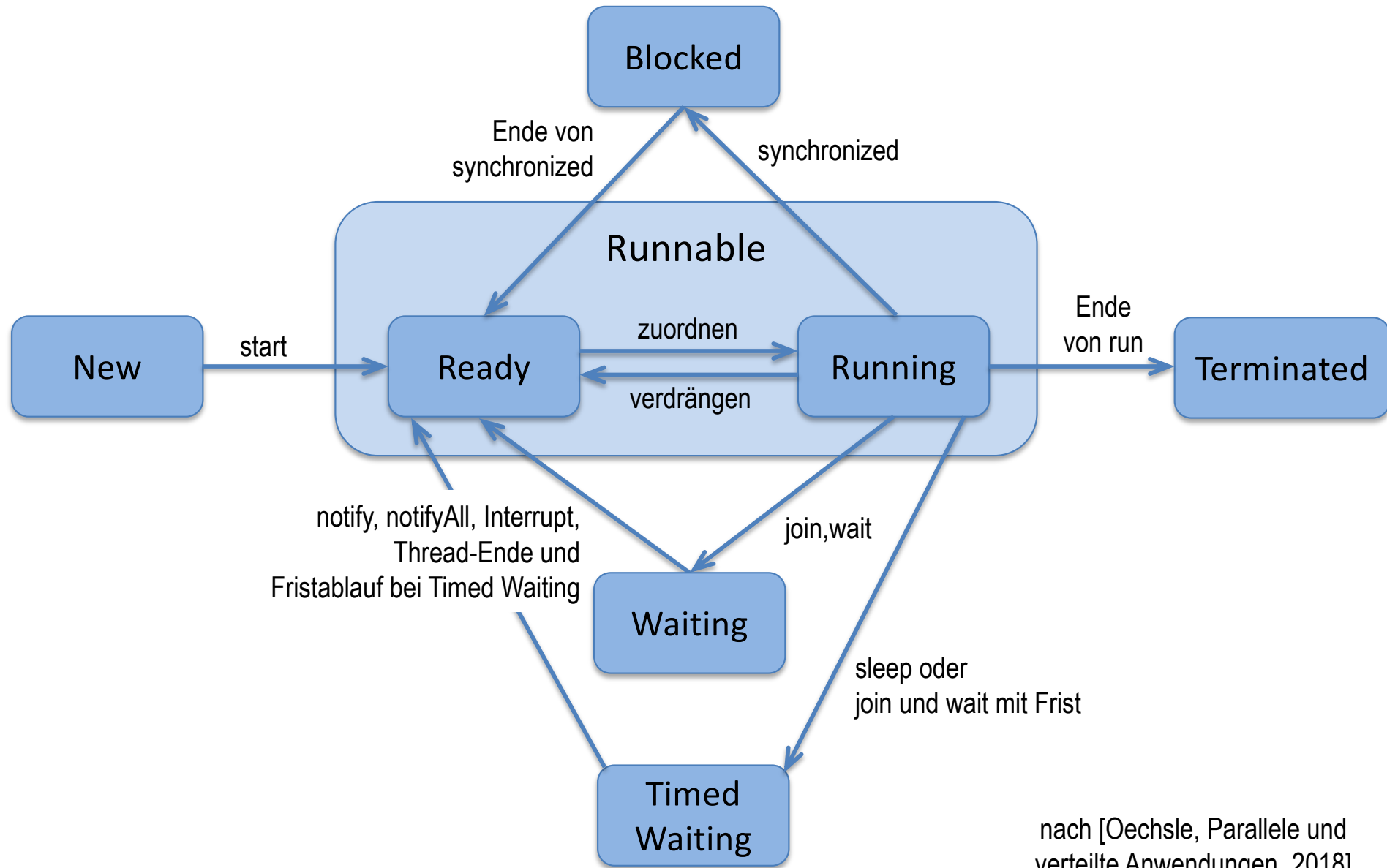
Hier muss wenigstens ein Producer-Thread aktiviert werden.

Da die Aktivierung irgendeines Threads nicht genügen würde, werden alle Threads aktiviert.  
Daher: notifyAll (und nicht notify)

# Kapitel 14: Threads

- Einführung
- Klasse Thread und Interface Runnable
- Methode join und Parallelisierung von Algorithmen
- Synchronisierung mit synchronized
- Erzeuger/Verbraucher-Problem und die Methoden wait, notify und notifyAll
- Zustände eines Java-Threads
- Thread-sichere Typen in der Java API

# Zustände eines Java-Threads



nach [Oechsle, Parallele und verteilte Anwendungen, 2018]

# Kapitel 14: Threads

- Einführung
- Klasse Thread und Interface Runnable
- Methode join und Parallelisierung von Algorithmen
- Synchronisierung mit synchronized
- Erzeuger/Verbraucher-Problem und die Methoden wait, notify und notifyAll
- Zustände eines Java-Threads
- Thread-sichere Typen in der Java API

# Überblick über Thread-sichere Typen

Paket bzw. Klasse	Klasse bzw. Methoden	Beschreibung
java.util.concurrent.atomic	AtomicInteger AtomicIntegerArray ...	Verschiedene gekapselte Basistypen und Felder, die Thread-sicher sind
Collections	synchronizedCollection(c) synchronizedList(l) synchronizedMap(m) synchronizedSet(s) ...	Verschiedene statische Methoden zum Einhüllen von Collection-Typen, so dass Thread-Sicherheit gewährleistet ist.
Collections	unmodifiableCollection(c) unmodifiableList(l) unmodifiableMap(m) unmodifiableSet(s) ...	Verschiedene statische Methoden zum Einhüllen von Collection-Typen, so dass sie immutabel und damit Thread-sicher werden
java.util.concurrent	BlockingQueue ConcurrentMap ...	Verschiedene Thread-sichere Typen

# Beispiel mit AtomicInteger

```
class AtomicInteger {
    AtomicInteger(int initialValue)
    int get() { ... }
    int addAndGet(int delta) { ... }
    boolean compareAndSet(int expect, int update) { ... }
    int accumulateAndGet (int x, IntBinaryOperator f) { ... }
    // ...
}
```

```
public static void main(...) {
    AtomicInteger sum = new AtomicInteger(0);
    class RandomSumThread extends Thread {
        public void run() {
            for (int i = 0; i < 1000; i++) {
                int r = (int) (Math.random()*100);
                sum.accumulateAndGet(r, (x, y) -> x+y);
            }
        }
    }
    RandomSumThread t1 = new RandomSumThread(); t1.start();
    RandomSumThread t2 = new RandomSumThread(); t2.start();
    t1.join(); t2.join();
    System.out.println("Sum = " + sum.get());
}
```

- Atomic-Integer aus dem Paket `java.util.concurrent.atomic` enthält verschiedene Methoden, um einfache int-Werte Thread-sicher und ohne eigene Synchronisation zu manipulieren.
- `accumulateAndGet` aktualisiert den int-Wert `a` des `AtomicInteger`-Objekts durch `f(a,x)`.
- Die beiden Threads `t1` und `t2` erzeugen jeweils 1000 zufällige Zahlen aus `[0,100)` und summieren sie auf die gemeinsame Variable `sum`.

# Synchronisierte Collections (1)

---

- Die Klasse Collections enthält verschiedene statische Methoden, um ein Collection-Objekt in eine Thread-sichere Hülle zu packen.

```
List<Integer> intList = new LinkedList<>();  
List<Integer> syncIntList = Collections.synchronizedList(intList);  
Map<String, Integer> telBuch = new TreeMap<>();  
Map<String, Integer> syncTelBuch = Collections.synchronizedMap(telBuch );
```

- Der Zugriff auf das Collection-Objekt ist damit synchronisiert und es kann nebenläufig zugegriffen werden.

```
List<Integer> intList = new LinkedList<>();  
List<Integer> syncIntList = Collections.synchronizedList(intList);  
class RandomThread extends Thread {  
    public void run() {  
        for (int i = 0; i < 1000; i++)  
            syncIntList.add(Math.random());  
    }  
}  
new RandomThread().start();  
new RandomThread().start();
```



# Synchronisierte Collections (2)

- Wird in einem Thread über das Collection-Objekt `c` iteriert und in einem anderen Thread das **Objekt `c` verändert**, kann eine **ConcurrentModificationException** ausgelöst werden.

```
List<Double> dbList = new LinkedList<>();
List<Double> syncDoubleList = Collections.synchronizedList(dbList);

class RandomThread extends Thread {
    public void run() {
        for (int i = 0; i < 1000; i++) {
            syncDoubleList.add(Math.random());
            for (double x : syncDoubleList)
                System.out.println(x);
        }
    }
}

new RandomThread().start();
new RandomThread().start();
```

Vorsicht:  
ConcurrentModificationException!

- Abhilfe: Iterator-Schleife in einem synchronized-Block

[https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Collections.html#synchronizedList\(java.util.List\)](https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Collections.html#synchronizedList(java.util.List))

# Immutable Collections

---

- Die Klasse Collections enthält verschiedene statische Methoden, um ein Collection-Objekt in eine Hülle zu packen, so dass nur lesende Operationen durchgeführt werden können. Container wird damit **immutabel**.
- Es können dann problemlos mehrere Threads lesend auf das Collection-Objekt ohne zusätzliche Synchronisation zugreifen.

```
List<Integer> intList = new LinkedList<>();
intList.add(5);
intList.add(7);
// ...
List<Integer> constList = Collections.unmodifiableList(intList);

Map<String, Integer> telBuch = new TreeMap<>();
telBuch.put("Maier", 1234);
telBuch.put("Anton", 5678);
// ...
Map<String, Integer> constTelBuch = Collections.unmodifiableMap(telBuch);
```

# BlockingQueue aus java.util.concurrent

---

- Das Interface `BlockingQueue` und seine Implementierungen `LinkedBlockingQueue` und `ArrayBlockingQueue` lösen das Erzeuger/Verbraucher-Problem.
- Die Methode `put` hängt ein neues Element an die Schlange an und wartet dabei, solange die Schlange voll ist.
- Die Methode `take` holt das vorderste Element aus der Schlange und wartet dabei, solange die Schlange leer ist.

```
interface BlockingQueue<E> {  
    void put(E e) throws InterruptedException;  
    E take() throws InterruptedException;  
    // ...  
}
```

# Beispiel mit BlockingQueue (1)

Producer-Thread schreibt 100 Zahlen in die BlockingQueue.

```
class Producer extends Thread {  
    private final BlockingQueue<Integer> bq;  
    private final int start;  
    public Producer(BlockingQueue<Integer> bq, int s) {  
        this.bq = bq;  
        this.start = s;  
    }  
    public void run() {  
        for (int i = start; i < start+100; i++)  
            bq.put(i);  
    }  
}
```

Consumer-Thread holt 150 Zahlen aus der BlockingQueue und gibt sie aus.

```
class Consumer extends Thread {  
    private final BlockingQueue<Integer> bq;  
    private final String name;  
    public Consumer(BlockingQueue<Integer> bq, String n) {  
        this.bq = bq;  
        this.name = n;  
    }  
    public void run() {  
        for (int i = 0; i < 150; i++)  
            try {  
                System.out.println(name + ": " + bq.take());  
            } catch (InterruptedException ex) {}  
    }  
}
```

# Beispiel mit BlockingQueue (2)

```
public static void main(String[ ] args) {  
    BlockingQueue<Integer> bq  
        = new LinkedBlockingQueue<>(10);  
  
    Producer p1 = new Producer(bq, 0);  
    Producer p2 = new Producer(bq, 1000);  
    Producer p3 = new Producer(bq, 1000_000);  
  
    Consumer c1 = new Consumer(bq, "consumer1");  
    Consumer c2 = new Consumer(bq, "consumer2");  
  
    p1.start();  
    p2.start();  
    p3.start();  
  
    c1.start();  
    c2.start();  
  
}
```

Es wird eine BlockingQueue definiert, die maximal 10 Elemente aufnehmen kann.

Es werden 3 Producer-Thread gestartet, die insgesamt 300 Zahlen in die BlockingQueue schreiben.

Es werden 2 Consumer-Threads gestartet, die insgesamt 300 Zahlen aus der BlockingQueue holen und ausgeben.