

Algorithmen & Datenstrukturen

Thread Steuerung

Kooperation von Threads

Roger Diehl



Inhalt

- Warten auf Bedingungen
- Bounded Buffer
- Verlorene Signale
- Zusammenfassung

Lernziele

- Sie wissen was Guarded Blocks sind.
- Sie können das nebenläufige Warten auf Bedingungen in Java anwenden.
- Sie kennen den vollständigen Lebenszyklus für Java Threads und können bestimmen, wann ein Thread sich in welchem Zustand befindet.
- Sie wissen wie ein Semaphor funktioniert.
- Sie kennen die Regeln, die mit der Anwendung von **wait**, **notify** und **notifyAll** beachtet werden müssen.
- Sie können die Code-Beispiele mit **wait**, **notify** und **notifyAll** nachvollziehen, modifizieren und erstellen.

Warten auf Bedingungen

Warten auf Bedingungen

- Wenn der Zugang zu einem kritischen Abschnitt von bestimmten Bedingungen oder Zuständen abhängt, so reicht das Konzept der einfachen Synchronisation (synchronized) nicht aus.
- Dieses Konzept wird **guarded blocks** genannt (1999, Doug Lea, Concurrent Programming in Java, Chapter 3.2 ff).
- In der Praxis heisst das, es muss nebenläufig auf eine bestimmte Bedingung oder einen Zustand gewartet werden.
- Eine Möglichkeit wäre busy waiting. Dies ist aber nicht effizient.
- **Idee:** Die Threads warten an einem Monitorobjekt, bis dieses ein Signal erhält, um einen oder mehrere Threads frei zu schalten.

Guarded Blocks - Prüfung

- Der Guarded Block (bewachter Block) beginnt mit dem Prüfen einer Bedingung, die wahr sein muss, bevor der Thread im Block fortfahren kann.

Quelle: <http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html>

```
public synchronized void guardedJoy() {  
    while (!joy) {  
        try {  
            this.wait();  
        } catch (InterruptedException e) {  
            /* handling... */  
        }  
    }  
    System.out.println("Joy have been achieved!");  
}
```

Prüfen, ob Bedingung zutrifft...
...wenn nicht, warten!

Wichtig!

Das Prüfen der Bedingung für die `wait`-Methode ist immer mit einer `while`-Anweisung auszuführen! Denn die Bedingung könnte sich nebenläufig geändert haben, wenn der Thread wieder aufgeweckt wird.

wait

Methoden der Klasse `Object`. Threads müssen den Lock des Objekts besitzen an dem `wait` aufgerufen wird.

```
public final void wait() throws InterruptedException
public final void wait(long timeout)
                        throws InterruptedException
```

- Bei Aufruf von `wait` wird der aufrufende Thread in einen Wartezustand versetzt, und **gleichzeitig wird der Lock auf diesen Abschnitt freigegeben.**
- Es führen genau drei Wege aus dem Warte-Zustand wieder heraus:
 - 1) Ein anderer Thread signalisiert den Zustand Wechsel mittels `notify` bzw. `notifyAll`.
 - 2) Die im Argument angegebene Zeit (timeout) ist abgelaufen.
 - 3) Ein anderer Thread ruft die Methode `interrupt` des wartenden Threads auf.

Guarded Blocks – Benachrichtigung

- Nach dem Aufruf von **wait**, muss ein anderer Thread die wartenden Threads benachrichtigen, dass der Guarded Block nun frei ist.

Quelle: <http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html>

```
public synchronized void notifyJoy() {  
    joy = true;  
    this.notifyAll();  
}
```

← Bedingung setzen...

← ...wartende Threads benachrichtigen

Wichtig!

Die Benachrichtigung der **notify/notifyAll**-Methoden wird nicht gespeichert, wenn der wait-Pool leer ist, d.h. wenn Threads erst **notify/notifyAll** und danach **wait** aufrufen, bleiben sie im wait-Pool.

➔ Deadlock Gefahr.

notify / notifyAll

Methoden der Klasse **Object**. Threads müssen den Lock des Objekts besitzen an dem **notify** oder **notifyAll** aufgerufen wird.

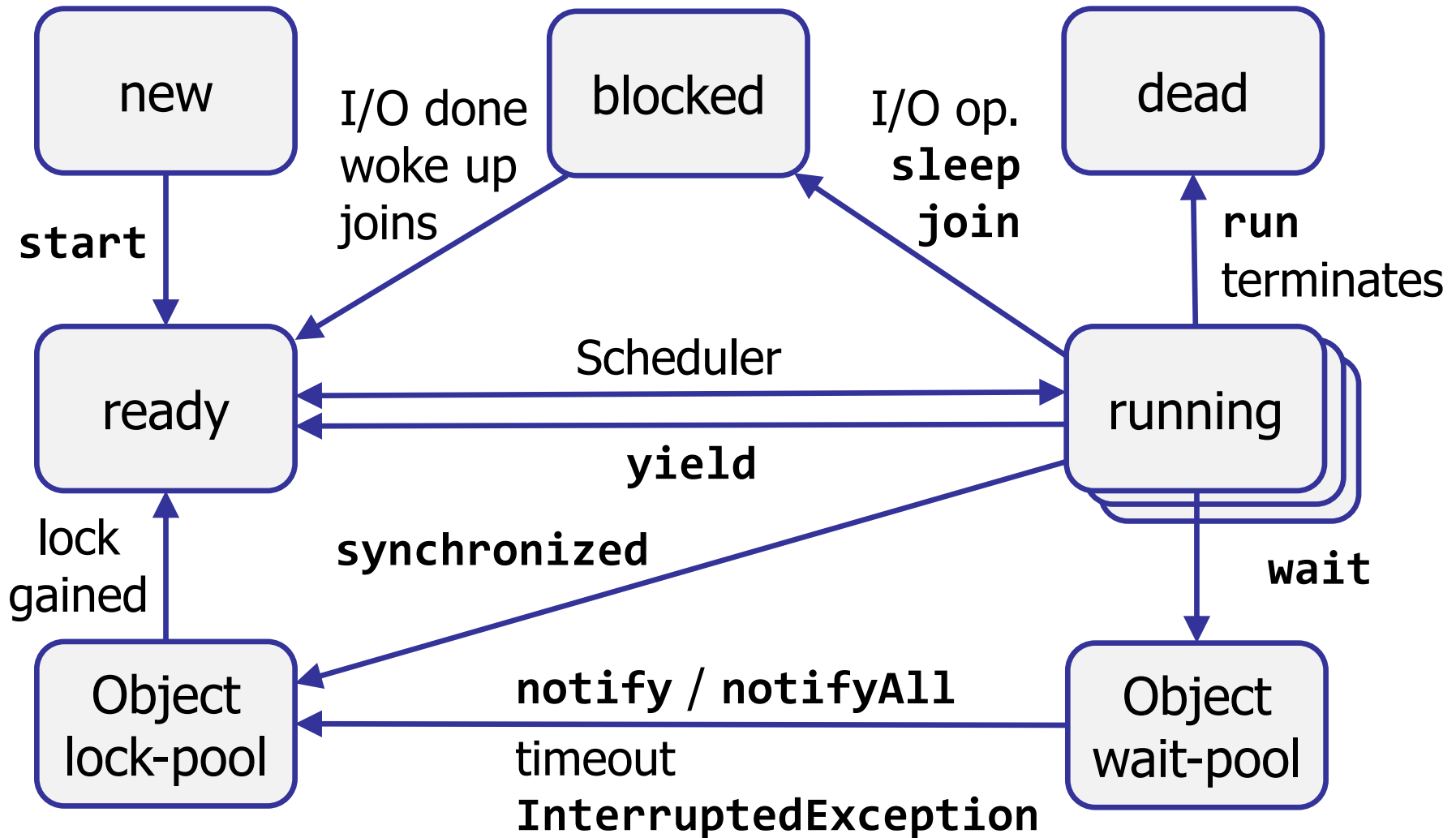
public final void notify()

- **notify** weckt genau einen Thread im Warte-Zustand auf. Falls mehrere Threads warten, ist nicht vorhersehbar oder bestimmbar, welcher Thread aufgeweckt wird.
- der Thread wartet noch einmal, bis er den Lock für den **synchronized** Abschnitt erhält. Erst dann wird er wieder "ready", d.h. bereit zur erneuten Ausführung.

public final void notifyAll()

- **notifyAll** weckt alle an diesem Objekt wartenden Threads auf.

Der vollständige Lebenszyklus

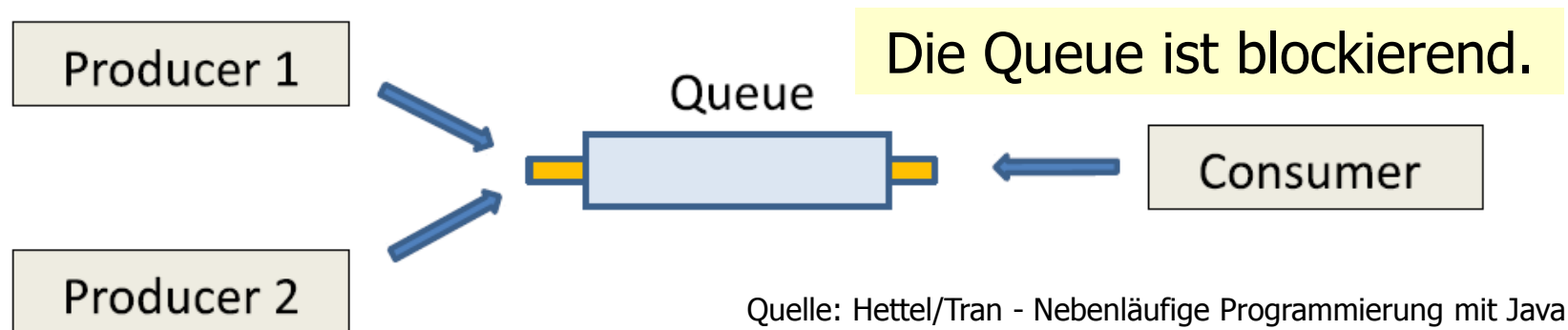


Jedes Objekt hat genau einen Object lock-pool und genau einen Object wait-pool.

Bounded Buffer

Bounded Buffer

- Eine der häufigsten Aufgabenstellungen in der nebenläufigen Programmierung ist, dass ***n*** Produzenten Daten erzeugen, die an ***m*** Konsumenten weitergegeben werden sollen.
- Zur Weitergabe der Daten dient ein Puffer, der in der Lage sein soll, ***k*** Daten zu speichern.
 - Dabei soll $m \geq 1$, $n \geq 1$ und $k \geq 1$ sein.
 - Falls der Puffer voll ist, sollen die Produzenten **warten**, bis wieder Platz ist.
 - Falls der Puffer leer ist, sollen die Konsumenten **warten**, bis wieder Daten vorhanden sind.



Quelle: Hettel/Tran - Nebenläufige Programmierung mit Java

Ein Beispiel zur Motivation

Implementation einer sehr einfachen Blocking Queue (gefunden im Forum java-samples.com)


- Nur ein Wert (**value**) kann geschrieben und gelesen werden
- Falls der Wert noch nicht geschrieben ist, wartet der Lesevorgang
- Falls der Wert noch nicht gelesen ist, wartet der Schreibvorgang

```
/**
 * Sehr einfache Blocking Queue.
 */
public final class SimpleQueue {

    private int value;
    private boolean valueSet = false;

    // ...
}
```

Flag, das anzeigt, ob der Wert geschrieben wurde.



Einfache Blocking Queue

```
//...
```

```
public synchronized int get()  
    throws InterruptedException {  
    if (!valueSet) {  
        this.wait();  
    }  
    valueSet = false;  
    this.notify();  
    return value;  
}
```

Falls der Wert noch nicht gespeichert wurde - warten.

Wenn der Wert gespeichert wurde - benachrichtigen.

```
public synchronized void put(final int value)  
    throws InterruptedException {  
    if (valueSet) {  
        this.wait();  
    }  
    this.value = value;  
    valueSet = true;  
    this.notify();  
}
```

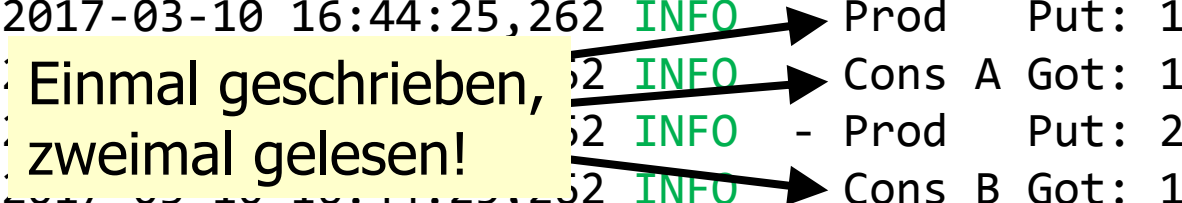
Falls der Wert noch nicht gelesen wurde - warten.

Wenn der Wert gelesen wurde - benachrichtigen.

Demo mit einfacher Blocking Queue

- Bei der Demo mit **einem** Produzenten und **einem** Konsumenten funktioniert die Blocking Queue einwandfrei.
 - Der Produzent liefert immer einen eindeutigen Wert.
- Sobald **zwei oder mehr** Konsumenten lesen wollen, funktioniert die Blocking Queue nicht mehr.

```
2017-03-10 16:44:25,260 INFO - Prod Put: 0
2017-03-10 16:44:25,260 INFO - Cons A Got: 0
2017-03-10 16:44:25,262 INFO - Prod Put: 1
Einmal geschrieben, zweimal gelesen!
2017-03-10 16:44:25,262 INFO - Cons A Got: 1
2017-03-10 16:44:25,262 INFO - Prod Put: 2
2017-03-10 16:44:25,262 INFO - Cons B Got: 1
2017-03-10 16:44:25,262 INFO - Cons A Got: 2
2017-03-10 16:44:25,262 INFO - Prod Put: 3
2017-03-10 16:44:25,262 ERROR - double value: 1
2017-03-10 16:44:25,262 INFO - Cons B Got: 3
...
```



Regel nach dem Warten – Bedingung wieder prüfen!

- Verwenden Sie zum Aufruf der `wait` Methode immer eine Iteration, wenn das Warten aufgrund einer Bedingung passiert. [Bloch 2002: Effektiv Java programmieren, Thema 50]
 - Vielleicht hat ein anderer Thread den Lock erworben und den geschützten Zustand/Bedingung geändert zwischen dem Zeitpunkt, zu dem ein Thread `notify` aufrief, und dem Zeitpunkt, zu dem der wartende Thread aufwachte.
 - Vielleicht hat auch ein anderer Thread `notify` aus Versehen oder mit böser Absicht zu einer Zeit aufgerufen, zu der die Bedingung nicht zutraf.
 - Eventuell ist der benachrichtigende Thread beim Aufwecken wartender Threads zu «grosszügig», wenn z.B. `notifyAll` aufgerufen wird.
 - Gegebenenfalls ist der wartende Thread auch ohne ein `notify` aufgewacht. Dies bezeichnet man als grundloses Aufwachen.

Regeln zur Benachrichtigung – `notify` oder `notifyAll`

- Zur Erinnerung: `notify` weckt einen einzigen wartenden Thread, `notifyAll` weckt alle wartenden Threads.
- Es wird empfohlen immer `notifyAll` verwenden.
 - Dies ist vernünftig, sofern alle `wait` Aufrufe innerhalb von `while` Schleifen stehen.
 - Es führt immer zu richtigen Ergebnissen, weil es garantiert, dass die Threads, die geweckt werden müssen, es auch tatsächlich werden.
- **Aber:** `notifyAll` kann zwar nicht die Richtigkeit, wohl aber die Leistung eines Programms beeinträchtigen.
 - Der Aufwand in bestimmten Fälle muss nicht linear sondern kann quadratisch wachsen.
- Bei Verwendung von `notify` ist sehr darauf achten, dass die Lebendigkeit von Threads erhalten bleibt.

Blocking FIFO Queue

- Implementation eines Ringpuffer über den Threads (Produzenten und Konsumenten) Daten austauschen können.
 - Der Ringpuffer wird durch ein Array von Objects realisiert.
 - Die Variable **count** entspricht der aktuellen Anzahl der Elemente.
 - Die Variable **tail** zeigt auf den ersten freien Platz zum Einfügen.
 - Die Variable **head** entspricht dem Index des nächsten verfügbaren Objekts.

```
/**
 * FIFO-Puffer (First In First Out) mit einer begrenzten Kapazität.
 */
public final class BoundedFIFOQueue<T> {

    private final Object[] data;
    private int head;
    private int tail;
    private int count;
    //...
```

Blocking FIFO Queue – Daten speichern

```
/**
 * Ein Element T speichern. Falls der Puffer voll ist, warten bis
 * ein Platz frei wird.
 * @param elem zu speicherndes Element
 * @throws InterruptedException, wenn das Warten unterbrochen wird
 */
public synchronized void put(final T elem)
    throws InterruptedException {
    while (count == data.length) {
        this.wait();
    }
    count++;
    data[tail] = elem;
    tail = (tail + 1) % data.length;
    if (count == 1) {
        this.notifyAll();
    }
}
```

Falls die Anzahl gespeicherten Elemente die Länge des Array erreicht haben - warten.

Element speichern und alle Attribute nachführen.

Wenn ein Element ins leere Array gespeichert wurde – wartende Thread benachrichtigen.

Warum eine Selektion für die Benachrichtigung?

Blocking FIFO Queue – Daten lesen

```
/**
 * Ein Element T auslesen. Falls der Puffer leer ist, warten bis
 * ein Platz belegt ist.
 * @return ausgelesenes Element
 * @throws InterruptedException, wenn das Warten unterbrochen wird
 */
public synchronized T get() throws InterruptedException {
    while (count == 0) {
        this.wait();
    }
    count--;
    T obj = (T) data[head];
    data[head] = null;
    head = (head + 1) % data.length;
    if (count == data.length - 1) {
        this.notifyAll();
    }
    return obj;
}
```

Falls keine Elemente gespeichert sind - warten.

Element lesen, Array Feld auf **null** setzen und alle Attribute nachführen.

Wenn ein Element aus dem vollen Array gelesen wurde – wartende Thread benachrichtigen.

Demo der Blocking FIFO Queue

- Der Produzent produziert Integer Zahlen und speichert diese in die Queue.
 - Zur Kontrolle berechnet der Produzent die Quersumme der produzierten Zahlen.
- Der Konsument liest Integer Zahlen aus der Queue.
 - Zur Kontrolle berechnet der Konsument die Quersumme der gelesenen Zahlen.

```
2017-03-13 18:33:14,117 INFO - Prod A = 49595820
2017-03-13 18:33:14,120 INFO - Prod B = 19728621
2017-03-13 18:33:14,120 INFO - Prod C = 41619126
2017-03-13 18:33:14,120 INFO - Cons A = 55652692
2017-03-13 18:33:14,121 INFO - Cons B = 55290875
2017-03-13 18:33:14,121 INFO - 110943567 = 110943567
```

Quersummen identisch, d.h. keine Integer Zahl ging verloren

Blocking FIFO Queue – Daten speichern oder nicht...

- Ein Thread kann mit `wait(long timeout)` wieder aktiv werden, wenn das Timeout (in Millisekunden) abgelaufen ist.

```
public synchronized boolean put(final T elem, final long millis)
    throws InterruptedException {
    while (count == data.length) {
        this.wait(millis);
        if (count == data.length) {
            return false;
        }
    }
    count++;
    data[tail] = elem;
    tail = (tail + 1) % data.length;
    if (count == 1) {
        this.notifyAll();
    }
    return true;
}
```

Warten bis der Thread geweckt wird, oder das Timeout eintritt.

Falls das Array immer noch voll ist, war's das Timeout – **false**

Wenn das Element mit Erfolg gespeichert werden konnte – **true**

Regel InterruptedException beim Warten weitergeben

- Wenn die **InterruptedException** innerhalb des «Wartens auf Bedingungen» behandelt wird, kann es zu Fehlverhalten kommen.

```
public final class BadStartingLine {
```

```
    private boolean haltCondition = true;
```

```
    public synchronized void halt() {
```

```
        while (this.haltCondition) {
```

```
            try {
```

```
                this.wait();
```

```
            } catch (InterruptedException e) {
```

```
                Thread.currentThread().interrupt();
```

```
            }
```

```
        }
```

```
    }
```

```
    public synchronized void go() {
```

```
        this.haltCondition = false;
```

```
        this.notifyAll();
```

```
    }
```

```
}
```

Threads werden durch **halt** angehalten.

Ist der Thread vor dem Eintritt in **halt** interrupted – Endlosschleife!

Alle Threads werden durch **go** geweckt.

Verlorene Signale


Ein Beispiel zur Motivation

- Ein Thread addiert Zahlen und...

```
static final Object lock = new Object();  
static long sum = 0;
```

```
Thread t1 = new Thread(() -> {  
    for (int i = 1; i <= 10000; i++) {  
        sum += i;  
    }  
    LOG.info("calc finished, notifying...");  
    synchronized (lock) {  
        lock.notify();  
    }  
});  
t1.start();
```

Steuerung wartender
Threads durch **notify**.



Codeskizze

Ein Beispiel zur Motivation

- ...ein Thread wartet auf das Resultat und gibt es aus.

```
Thread t2 = new Thread(() -> {  
    LOG.info("wait for result...");  
    synchronized (lock) {  
        try {  
            lock.wait();  
        } catch (InterruptedException e) {  
            return;  
        }  
    }  
    LOG.info("sum = " + sum);  
});  
t2.start();
```



Thread wartet...

Deadlock

Wann und Warum?

Codeskizze

Lost Signals

- Der Aufruf von **notify** und **notifyAll** wird nicht gespeichert!
- Wenn kein Thread wartet, geht das **notify** Signal verloren!
- Es muss eine Chronologie zwischen **wait** und **notify** herrschen.
 - Der Aufruf **wait** muss vor dem **notify** erfolgen.
 - Falls keine Chronologie besteht kann ein Deadlock drohen.
- **Lösung:** Die Aufrufe (Signale) von **wait** und **notify** müssen gespeichert werden.

Semaphor

- Semaphore = Mechanischer Signalgeber im Bahnverkehr
- Von E. Dijkstra (1968) eingeführt (counting semaphores).
- Das Semaphor ist ein Zähler für Passier-Signale.
- Das Semaphor kennt zwei Operationen:
 - P steht für passieren (passieren)
 - V steht für verhogen (erhöhen)
 - Synonyme für P und V sind:
 - up / down
 - wait / signal
 - acquire / release



Semaphor Definition

- Für ein Semaphor s sei
 - p die Anzahl der abgeschlossenen P-Operationen auf s
 - v die Anzahl der abgeschlossenen V-Operationen auf s
 - n ein Initialwert, der n Passiersignale im Semaphor initialisiert.
- Dann gilt die Invarianz: $s \geq 0$ and $s = n + v - p$
- Mit anderen Worten:
 - Operation V kann stets ausgeführt werden,
 - Operation P aber nur dann, wenn $p \leq v + n$.

Implementation eines Semaphors

- Ein nach oben nicht begrenztes Semaphor, d.h. der Zähler kann unendlich wachsen.

```
public final class Semaphore {
```

Codeskizze

```
    private int sema; // Semaphorzähler
```

```
    public Semaphore(int init) {
```

```
        sema = init;
```

```
    }
```

```
    public synchronized void acquire() throws InterruptedException {
```

```
        while (sema == 0) {
```

```
            this.wait();
```

```
        }
```

```
        sema--;
```

```
    }
```

```
    public synchronized void release() {
```

```
        this.notifyAll();
```

```
        sema++;
```

```
    }
```

```
}
```

← P Operation: Falls der Zähler den Zustand 0 hat – Thread wartet.

← V Operation: Threads werden geweckt und Zähler um 1 erhöht.

Beispiel: Addition mit Semaphor

- Ein Thread addiert Zahlen und...

```
static final Semaphore sema = new Semaphore();  
static long sum = 0;
```

Codeskizze

```
Thread t1 = new Thread(() -> {  
    for (int i = 1; i <= 10000; i++) {  
        sum += i;  
    }  
    LOG.info("calc finished, notifying...");  
    sema.release();  
});
```

Steuerung wartender Threads
durch ein Semaphor mit
Anfangszählerstand 0.

Wartende Threads benachrichtigen –
Summe kann einmal gelesen werden.

Beispiel: Ausgabe mit Semaphor

- ...ein Thread wartet auf das Resultat und gibt es aus.

```
Thread t2 = new Thread(() -> {  
    LOG.info("wait for result...");  
    try {  
        sema.acquire();  
    } catch (InterruptedException e) {  
        return;  
    }  
    LOG.info("sum = " + sum);  
});
```

Codeskizze

← Thread wartet...

t2.start(); ← Ausgabe-Thread wird als erstes gestartet...

t1.start(); ← Additions-Thread wird als zweites gestartet...

Anwendung von Semaphoren

- Ein Semaphor verwaltet die gemeinsamen Ressourcen nicht selbst.
- Ein Semaphor verwaltet nur den Zugriff auf die Anzahl der aktuell verfügbaren gemeinsamen Ressourcen.
 - Deshalb werden Semaphore hauptsächlich in komplexeren Synchronisationsmechanismen und weniger zur direkten Thread Steuerung eingesetzt.
- Bei der Anwendung von Semaphoren ist darauf zu achten, ob starke oder schwache Semaphore benötigt werden.
 - Ein starkes Semaphor besitzt eine FIFO Queue, damit das Windhund Prinzip (engl. «first come, first served») garantiert ist.
 - Ein schwaches Semaphor besitzt einen Pool und garantiert nicht die chronologisch richtige Abarbeitung der Warteschlange.

Bounded Buffer mit Semaphore

- Für die Implementation benötigt man
 - Thread sichere Queue oder List für die Speicherung der Datenelemente und
 - Semaphore für die Zugriffskontrolle.

```
public final class BoundedBuffer<T> {
```


```
    private final ArrayDeque<T> queue;  
    private final Semaphore putSema;  
    private final Semaphore takeSema;
```

```
    public BoundedBuffer(final int n) {  
        queue = new ArrayDeque<>(n);  
        putSema = new Semaphore(n);  
        takeSema = new Semaphore(0);  
    }
```


```
// ...
```

Codeskizze

Kapazität des Puffers,
bzw. der Queue



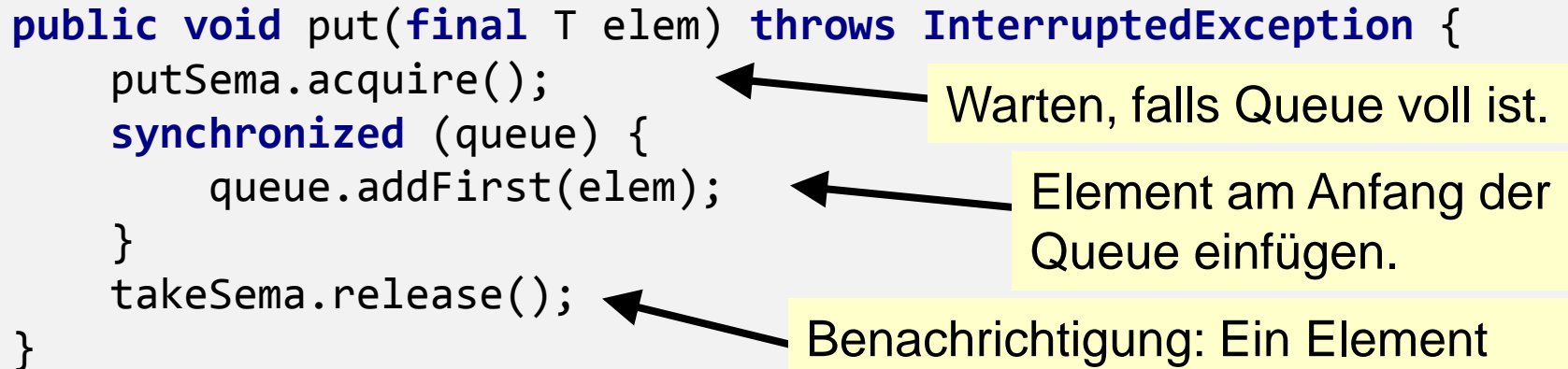
Anfangszustand –
leerer Puffer



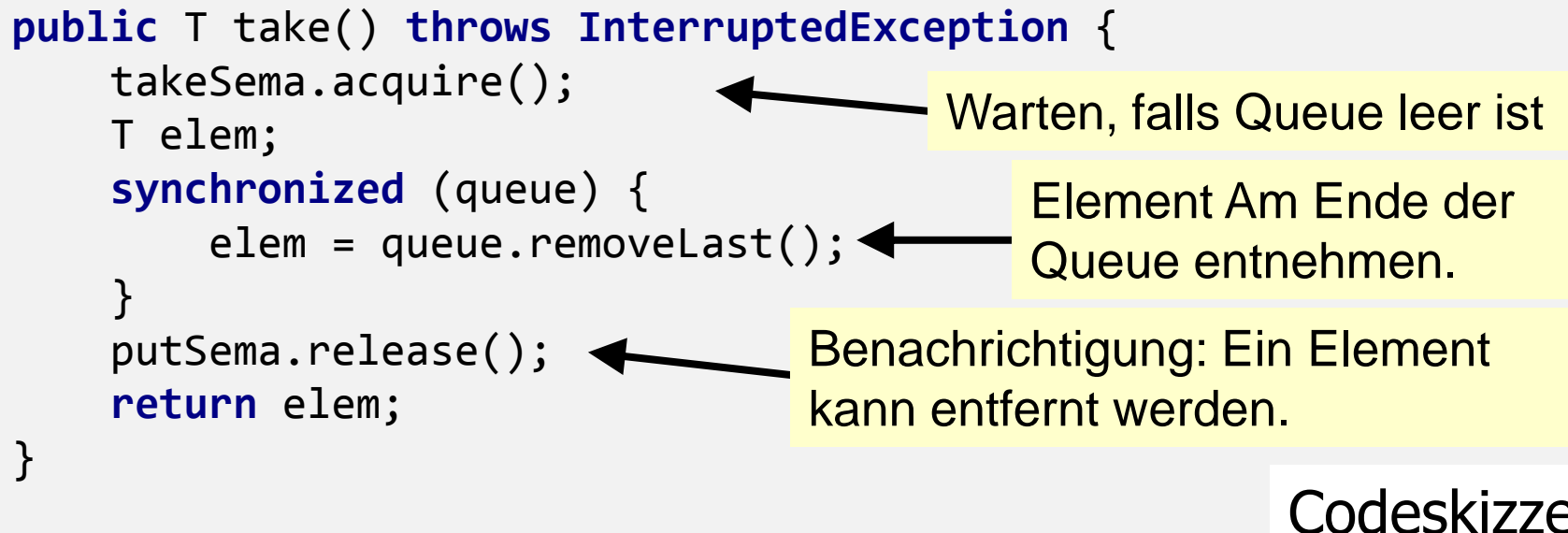
Zugriff auf den Bounded Buffer

- Beim Lesen und Schreiben gilt das FIFO Prinzip.

```
public void put(final T elem) throws InterruptedException {  
    putSema.acquire();  
    synchronized (queue) {  
        queue.addFirst(elem);  
    }  
    takeSema.release();  
}
```



```
public T take() throws InterruptedException {  
    takeSema.acquire();  
    T elem;  
    synchronized (queue) {  
        elem = queue.removeLast();  
    }  
    putSema.release();  
    return elem;  
}
```



Codeskizze

Zusammenfassung

- Mit **wait**, **notify** und **notifyAll** ist das Konzept des Wartens auf Bedingungen in Java umgesetzt. Threads belegen dabei keine Rechnerressourcen.
- Threads können durch **wait** schlafen gelegt werden und schlafende Threads können mit **notifyAll** geweckt werden.
- Die Methode **notify** sollte in der Regel nicht benutzt werden.
- Alle diese Methoden dürfen nur in Zusammenhang mit der durch **synchronized** erlangten Sperre verwendet werden.
- Das Prüfen einer Bedingung in Zusammenhang mit **wait** ist immer mit einer **while**-Anweisung auszuführen.
- Die Benachrichtigung mit **notify** oder **notifyAll** wird nicht gespeichert, wenn kein Thread wartet.

Fragen?