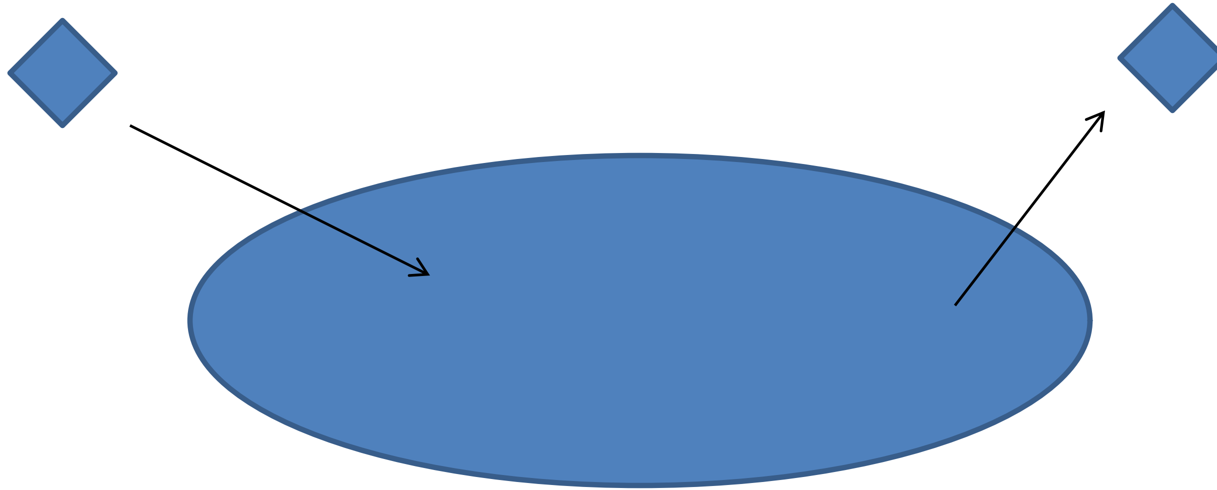
Concurrent Queues and the ABA Problem

Franziska Sauka
Elisabeth Engel

Agenda

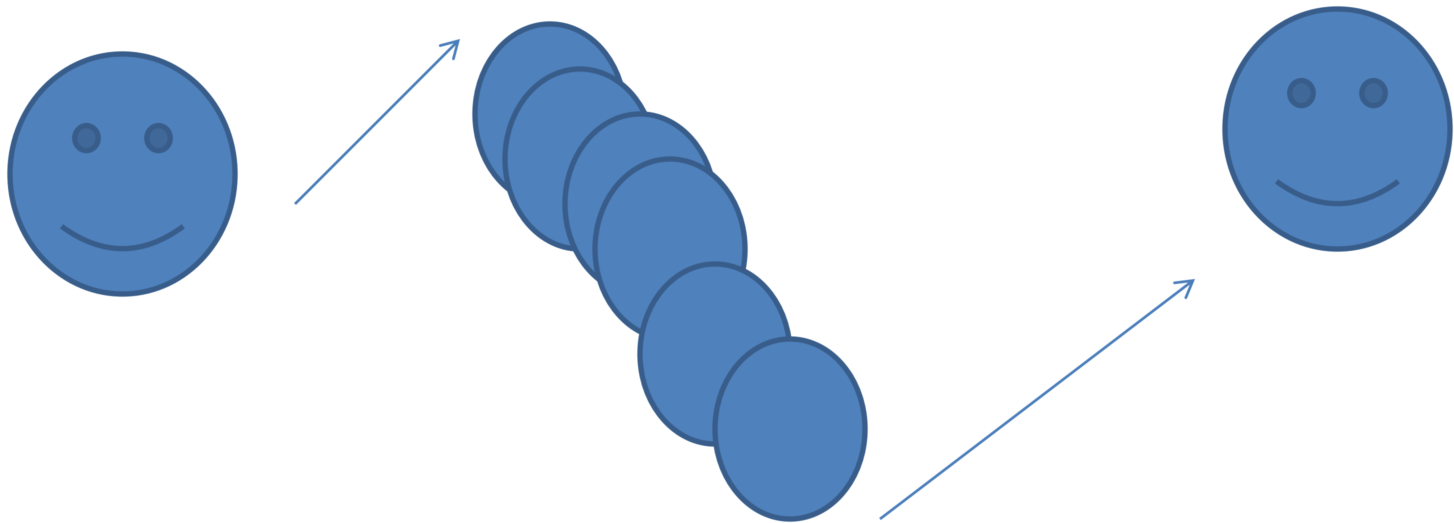
- Pools und das Erzeuger-Verbraucher Problem
- Queues
  - Unbounded Total Queue
  - Bounded Partial Queue
  - Unbounded Lock-free Queue
- Knoten Recycling und das ABA Problem
- Naive Synchronous Queue
- Dual Data Structure

Pools



- **Ähnlich wie Set-Klasse**, aber kein contains() und Elemente können mehrfach vorkommen
- **set() und get()-Methode**

Erzeuger-Verbraucher Problem



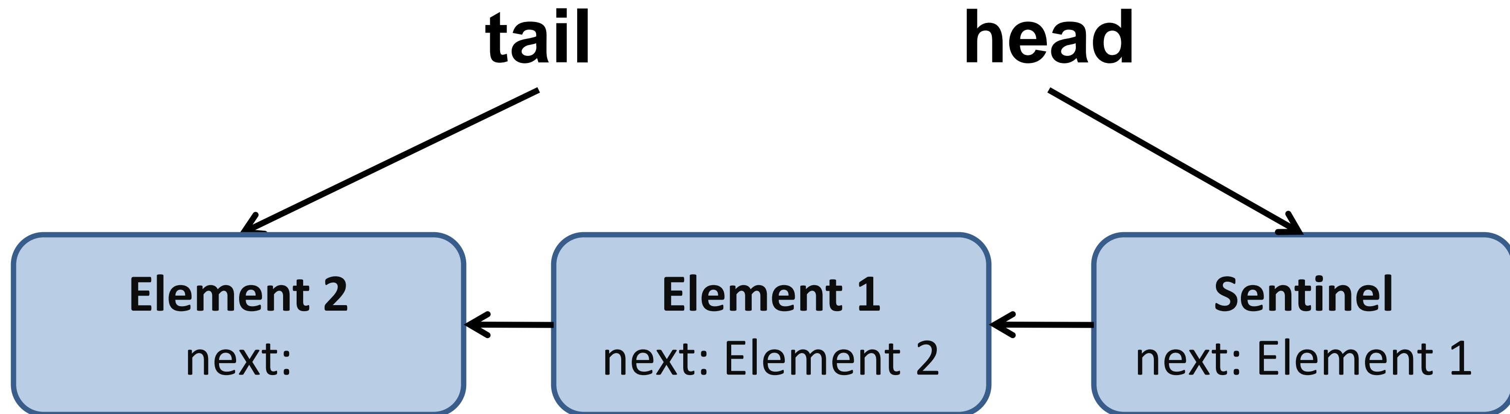
- **Problem:** Erzeuger **schneller** als Verbraucher

Pool Eigenschaften

- **Bounded und unbounded** → Kapazität
- **Total** → warten nicht, sondern wirft Exception
- **Partial** → warten
- **Synchronous** → 1:1, jeder Erzeuger bekommt genau einen Verbraucher

- **Fairness** (z.B. first-in-first-out)

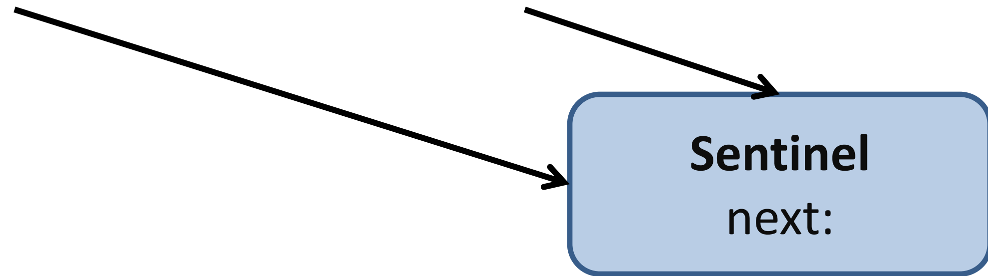
Concurrent Queues



- First-in-first-out Pool
- **Geordnete Folge von Knoten**
- **Zugriff von verschiedenen Seiten**

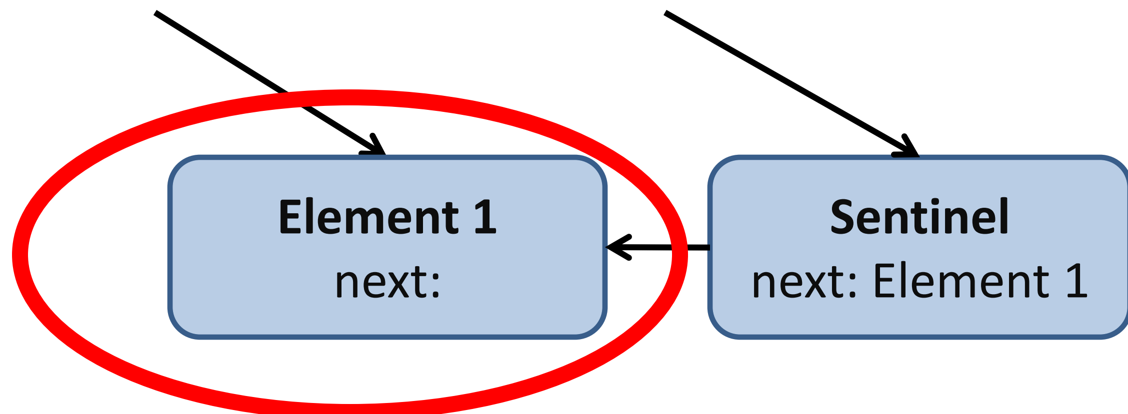
Concurrent Queues

Am Anfang: **tail** **head**



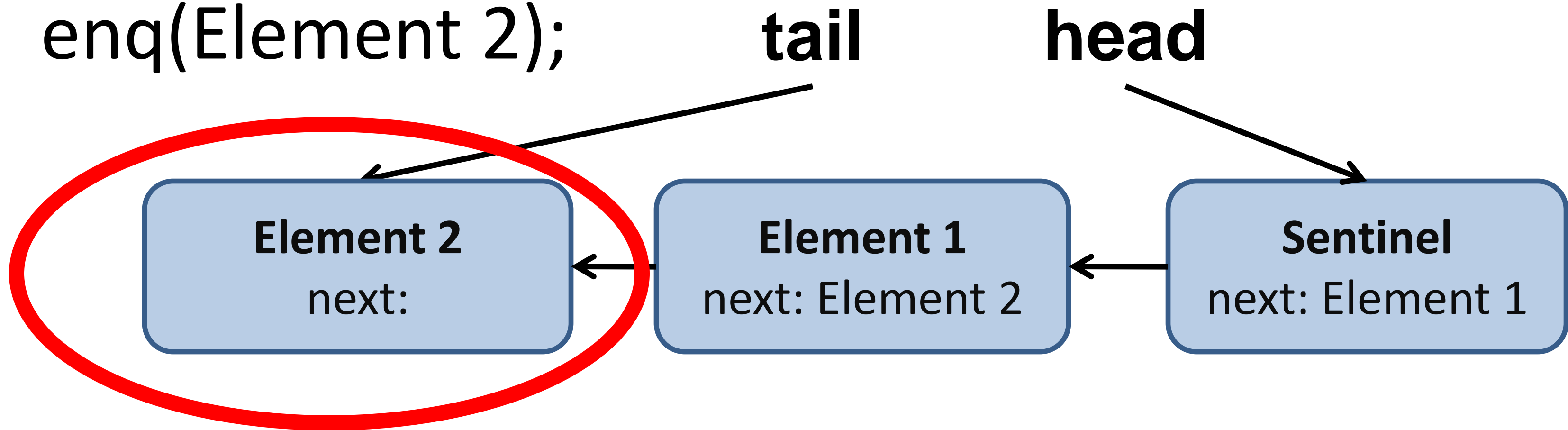
enq(Element 1);

tail **head**

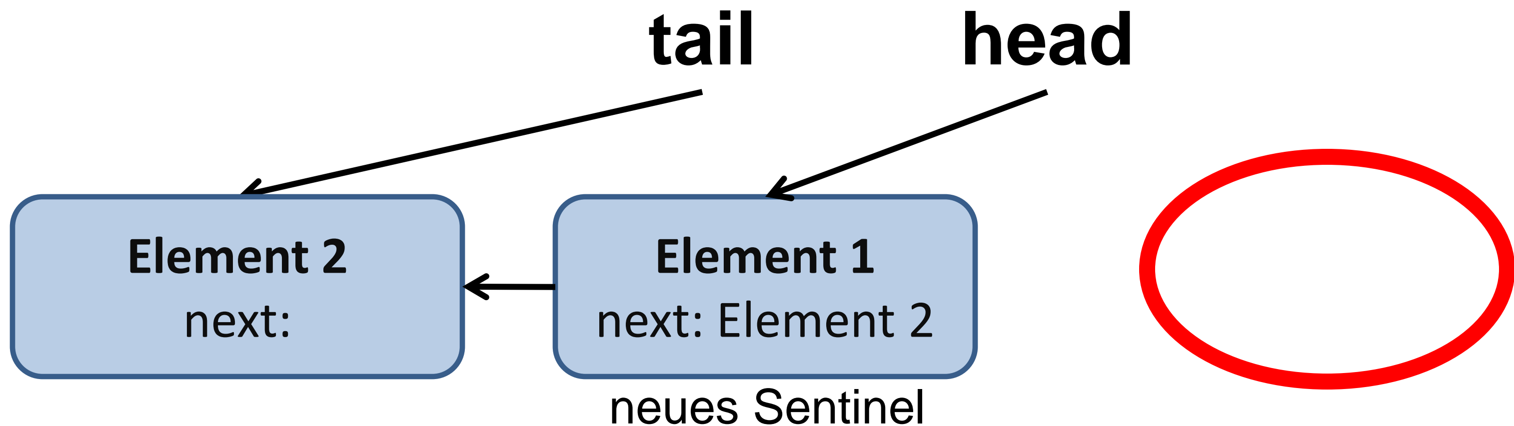


Concurrent Queues

enq(Element 2);

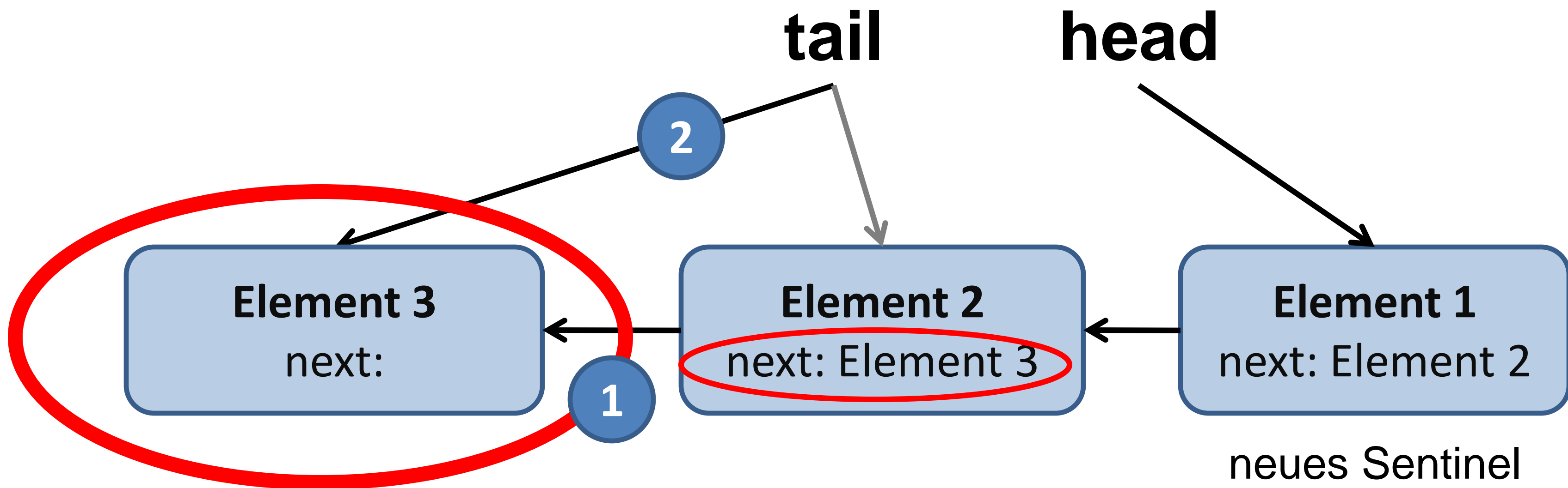


deq();



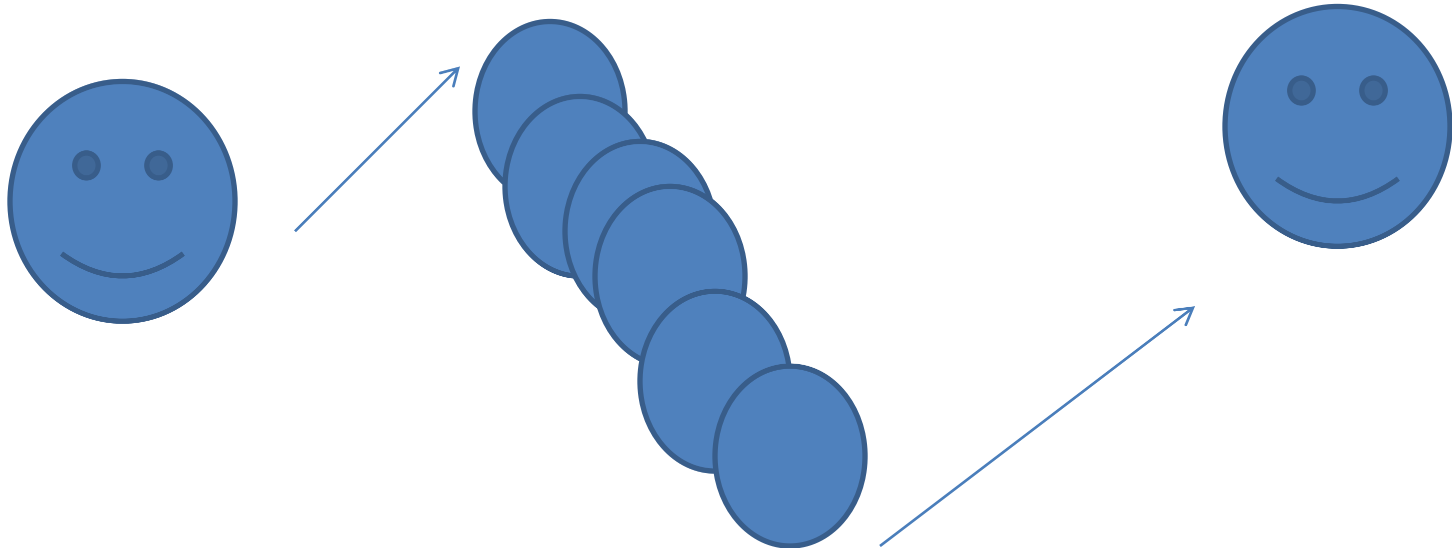
Concurrent Queues

- enq(Element 3);



- Nebenläufigkeit → Head und Tail zeigen nicht zwangsläufig auf das richtige Element

Unbounded Total Queue



- **Keine Kapazitätsbegrenzung**
- **Kein Warten** → deq() liefert event. eine EmptyException
- Enq und deq arbeiten an den beiden Enden → dadurch ist **paralleles Arbeiten möglich**

Unbounded Total Queue

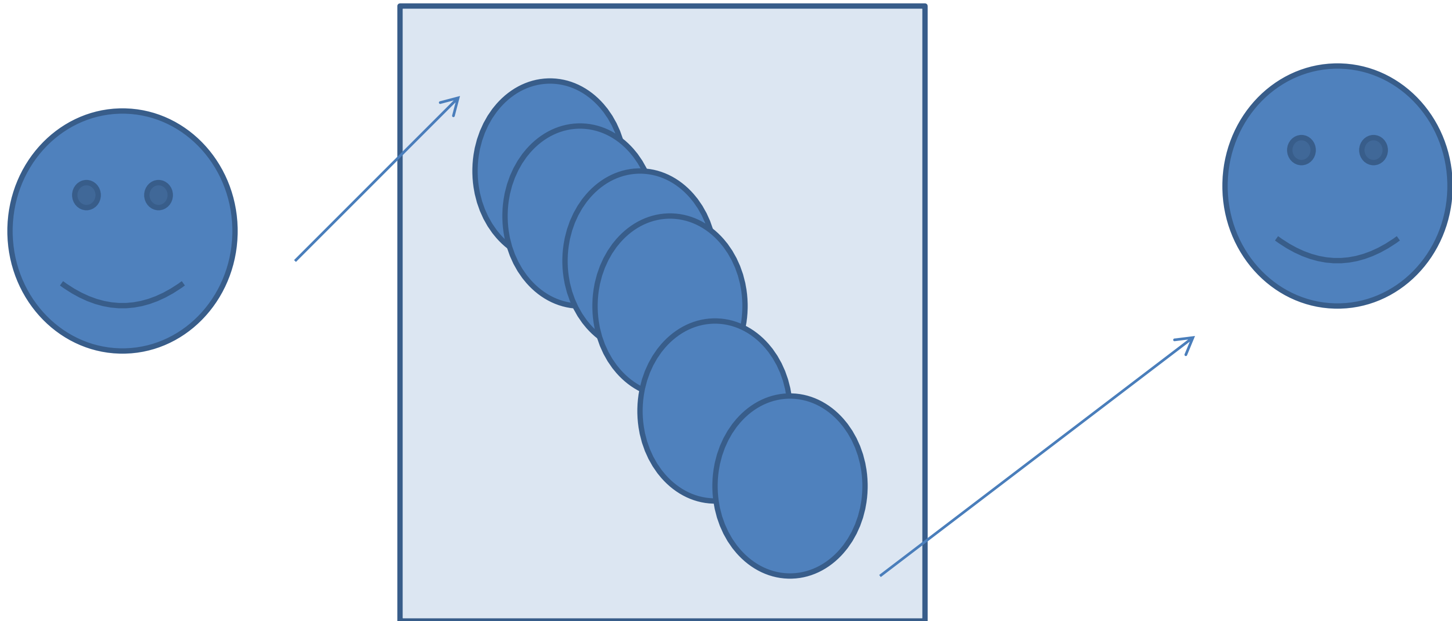
```
1   import java.util.concurrent.locks.ReentrantLock;
2   import queue.EmptyException;
3
4   public class UnboundedQueue<T> {
5
6     ReentrantLock enqLock, deqLock;
7     Node head; // Anfang
8     Node tail; // Ende
9
10    public UnboundedQueue() {
11      head = new Node(null); // Sentinel
12      tail = head; // beide zeigen darauf
13      enqLock = new ReentrantLock();
14      deqLock = new ReentrantLock();
15    }
16
```

```
44
45    protected class Node {
46
47      public T value; // Inhalt
48      public Node next; // Verbindung zum Nachfolger
49
50      public Node(T x) {
51        value = x; // Inhalt setzen
52        next = null; // Nachfolger ist null
53      }
54    }
    }
```

Unbounded Total Queue

```
17    public void enq(T x) {
18      if (x == null) throw new NullPointerException();
19      enqLock.lock(); // Lock setzen
20      try {
21        Node e = new Node(x); // neuer Knoten
22        tail.next = e; // verbinden
23        tail = e; // Ende umbiegen
24      } finally {
25        enqLock.unlock(); // Lock entfernen
26      }
27    }
28
29    public T deq() throws EmptyException {
30      T result;
31      deqLock.lock(); // Lock setzen
32      try {
33        if (head.next == null) {
34          throw new EmptyException();
35        }
36        result = head.next.value; // verbinden
37        head = head.next; // Head umbiegen
38      } finally {
39        deqLock.unlock();
40      }
41      return result; // Lock entfernen
42    }
43
```

Bounded Partial Queue



- **Bounded** → begrenzte Kapazität
- **Partial** → warten

Bounded Partial Queue

```java
1   import java.util.concurrent.atomic.AtomicInteger;
2   import java.util.concurrent.locks.Condition;
3   import java.util.concurrent.locks.ReentrantLock;
4
5   public class BoundedQueue<T> {
6
7     ReentrantLock enqLock, deqLock;
8     Condition notEmptyCondition, notFullCondition;
9     AtomicInteger size; // aktuelle Größe
10    Entry head; // Anfang
11    Entry tail; // Ende
12    int capacity; // Kapazität -> obere Grenze
13
14    public BoundedQueue(int capacity) {
15      this.capacity = capacity;
16      this.head = new Entry(null); // Sentinel
17      this.tail = head; // beide zeigen darauf
18      this.size = new AtomicInteger(capacity);
19      this.enqLock = new ReentrantLock();
20      this.notFullCondition = enqLock.newCondition();
21      this.deqLock = new ReentrantLock();
22      this.notEmptyCondition = deqLock.newCondition();
23    }
24
```

```java
84
85    protected class Entry {
86
87      public T value; // Inhalt
88      public Entry next; // Verbindung zum Nachfolger
89
90      public Entry(T x) {
91        value = x; // Inhalt setzen
92        next = null; // Nachfolger ist null
93      }
    }
  }
```

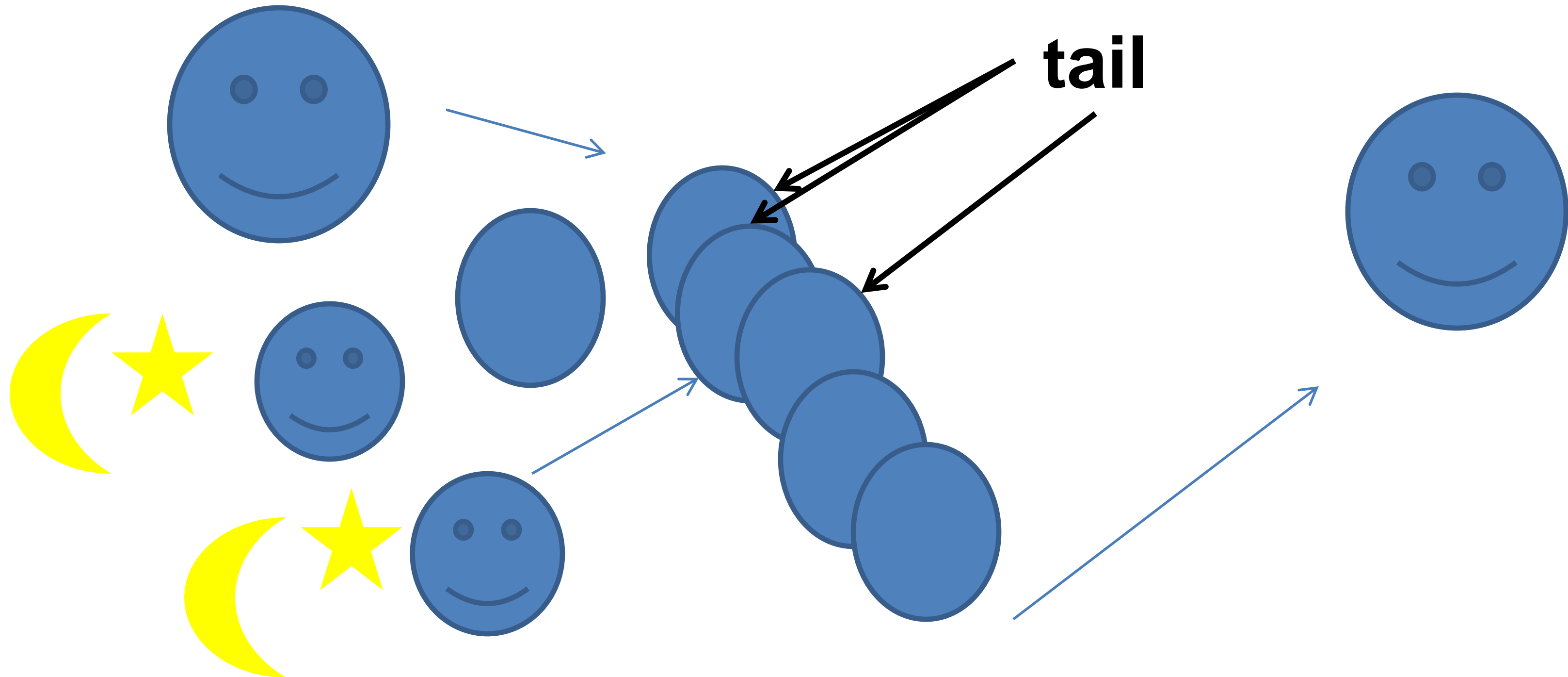
Bounded Partial Queue

```
25
26    public void enq(T x) {
27      if (x == null) throw new NullPointerException();
28      boolean mustWakeDequeuers = false;
29      enqLock.lock();
30      try {
31        while (size.get() == capacity) {
32          try {
33            notFullCondition.await(); // warten
34          } catch (InterruptedException e) {}
35        }
36        Entry e = new Entry(x); // neuer Knoten
37        tail.next = e; // verbinden
38        tail = e; // Ende umbiegen
39        if (size.getAndIncrement() == 0) {
40          mustWakeDequeuers = true;
41        }
42      } finally {
43        enqLock.unlock();
44      }
45      if (mustWakeDequeuers) {
46        deqLock.lock();
47        try {
48          notEmptyCondition.signalAll(); // andere aufwecken
49        } finally {
50          deqLock.unlock();
51        }
52      }
53    }
54
```

Bounded Partial Queue - Fazit

- **Vorteil:** Solide
- **Nachteil:**
  – getAndIncrement() und getAndDecrement() greifen beide auf *size* zu → Bottleneck
- **Lösung:**
  – Zwei Variablen, die nur bei Bedarf angeglichen werden

Unbounded Lock-Free Queue



- **Idee:** schnellere Threads helfen den langsameren

Unbounded Lock-Free Queue

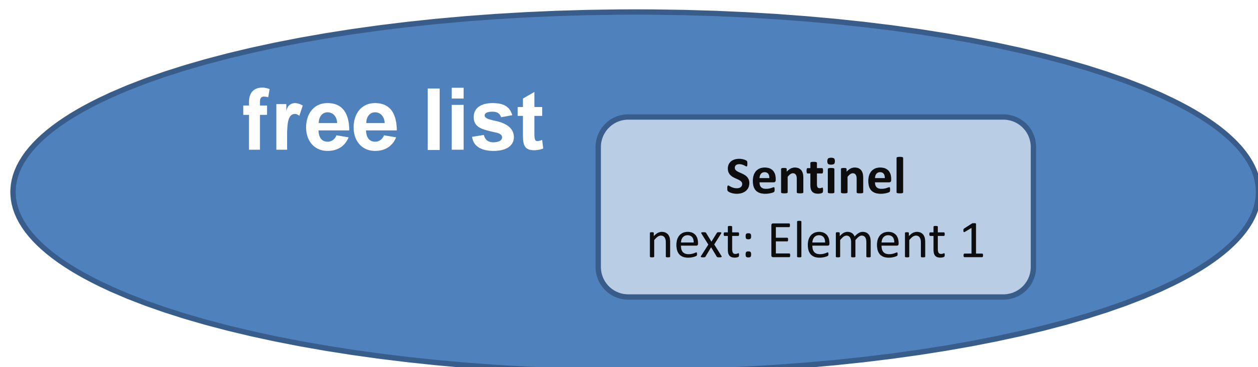
- Jeder Methoden-Aufruf schaut erst, ob er seinem **Vorgänger noch helfen** muss und macht **dann** seine **eigene Arbeit**
- **Besser als Blocking Queues**, da Lock-free

Knoten Recycling

- **Bei vielen Elementen** wäre es gut, alte wiederzuverwenden

  → „free list“ von nicht mehr genutzten Knoten

  → immer wenn ein neuer Knoten benötigt wird, wird **zuerst versucht einen alten zu recyclen**

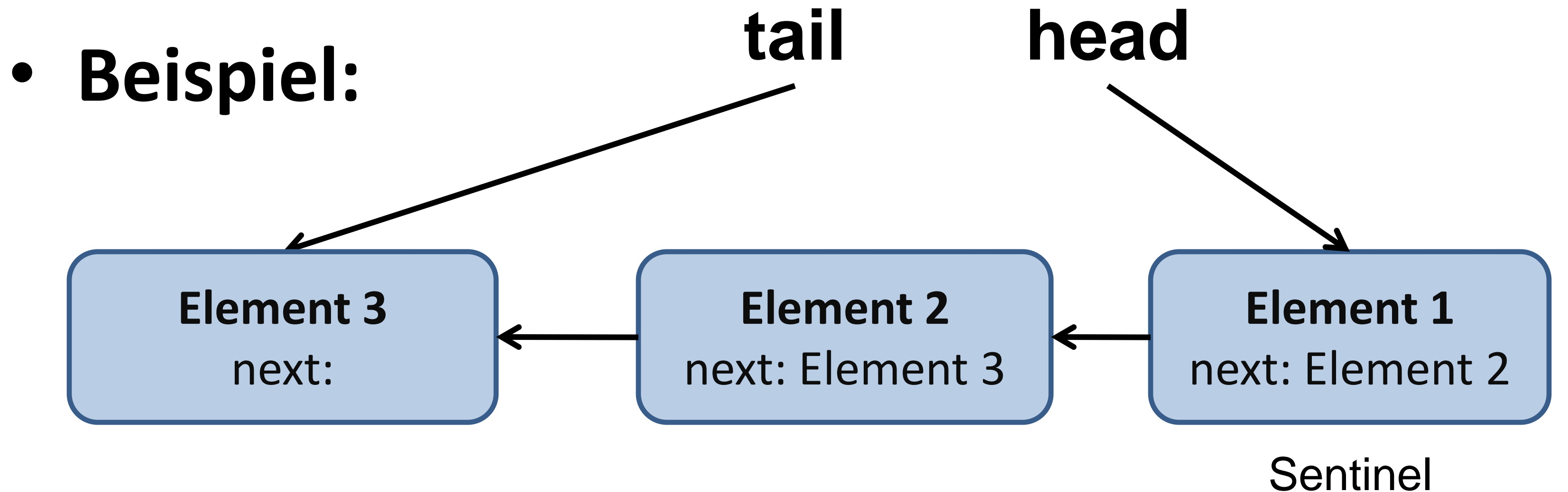


ABA Problem

- **alte Elemente werden neu verbunden**, aber durch **Threadwechsel** kann es sein, dass ein **Thread** sie **behandelt**, **als wären sie noch in ihrer alten Position tätig**

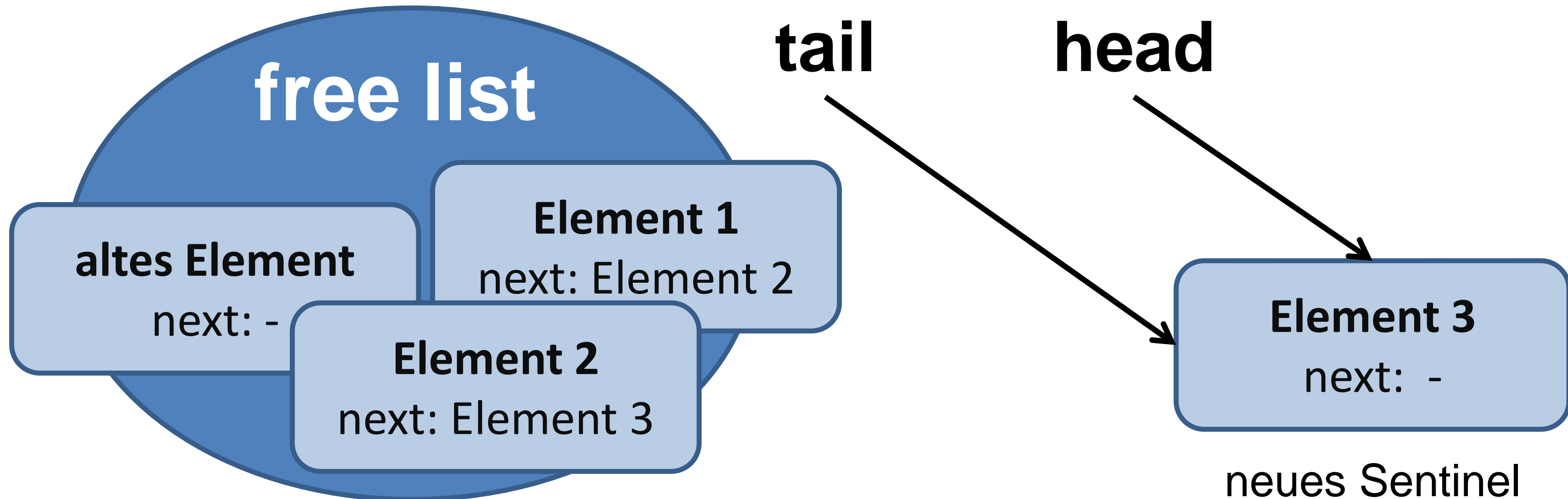
ABA Problem

- **Beispiel:**



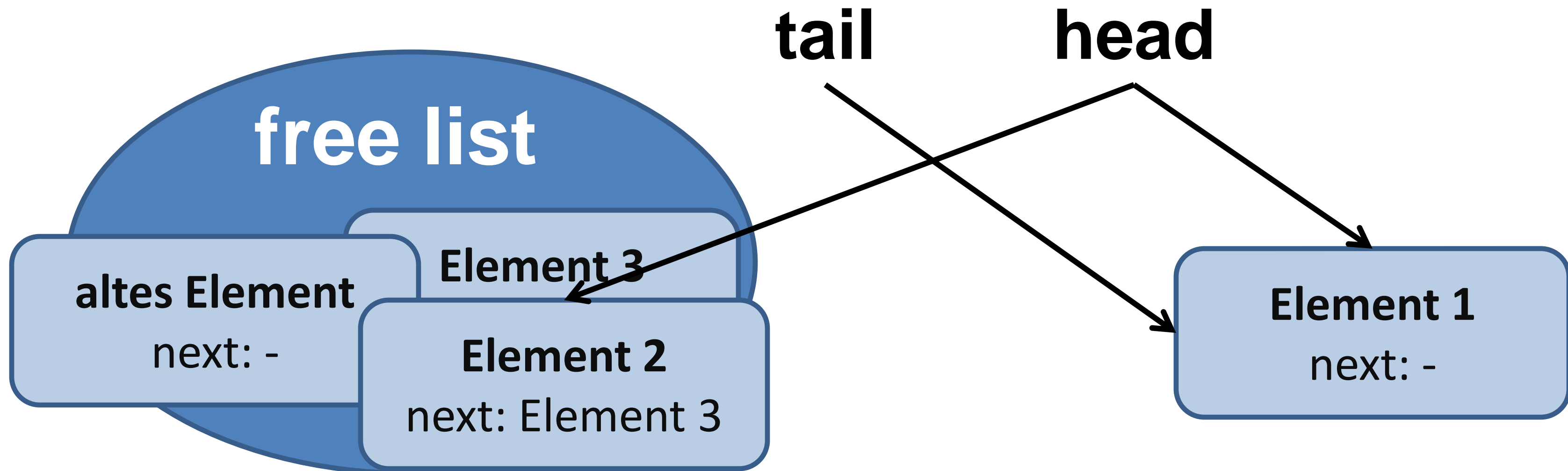
- **Thread A: deq() → will das erste Element**
  → in diesem Fall Element 2
  → um head neu zu setzen: **head.compareAndSet(Element 1, Element 2)**
  → wird aber **!vorher!** unterbrochen
- **Thread B: 2 x deq() → will das erste und zweite Element**
  → in diesem Fall Element 2 und 3

ABA Problem



- **Thread A: weiter unterbrochen**
- **Thread B:**
  - **enq(Element 1)** → Element 1 wird angehängt
  - **deq()** → Element 3 (Sentinel) wird entfernt

ABA Problem



- **Thread A: ist wieder dran**

  → will endlich **head.compareAndSet(Element 1, Element 2)** ausführen

  → Überprüfung Element 1 ist immer noch head

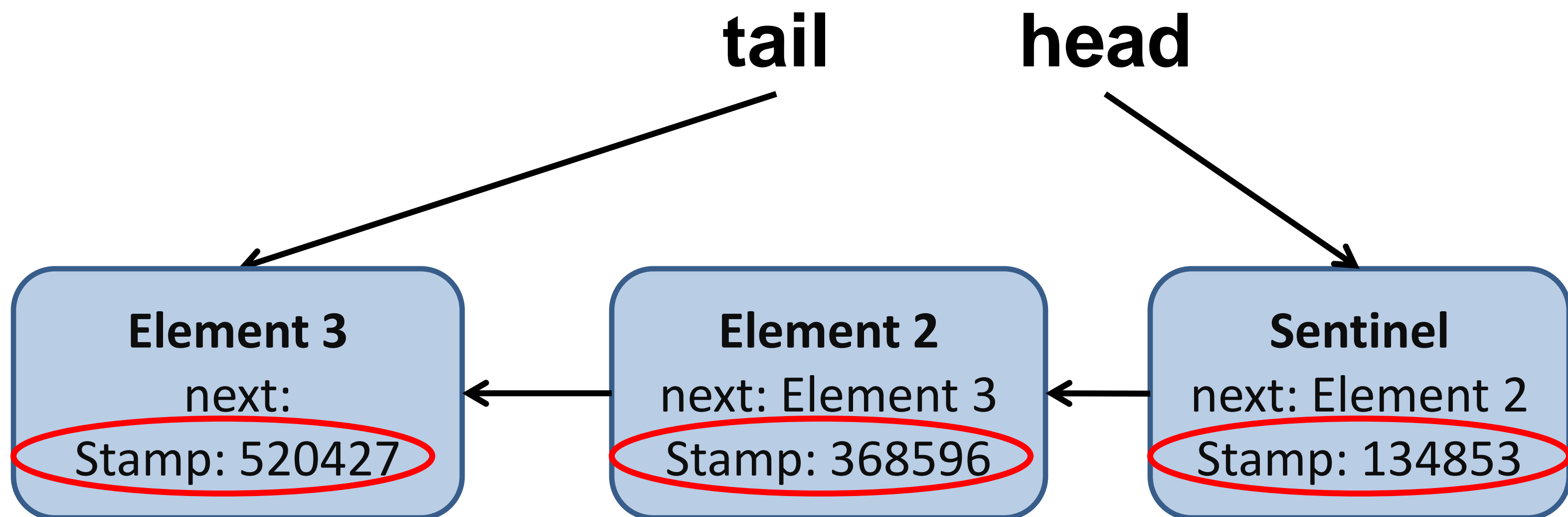
  → neuer head ist Element 2

!!!FEHLER!!! Element 2 ist gar nicht mehr in der Liste

ABA Problem

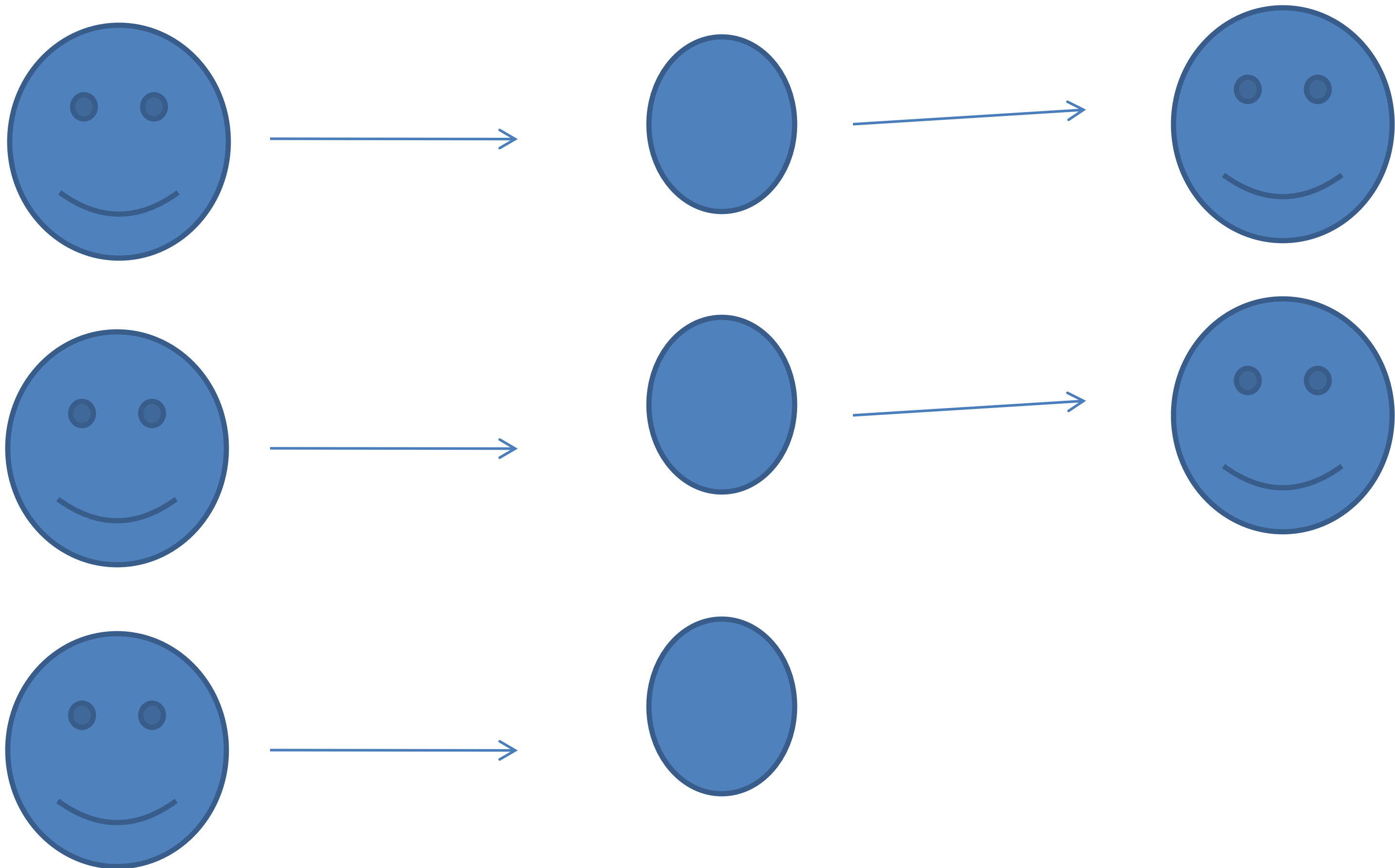
- **einfache Lösung:** jede atomare Referenz bekommt einen **eindeutigen Stempel** (Stamp)

  → keine Verwechslungen mehr



Naive Synchronous Queue

- **Rendevous von Erzeuger und Verbraucher**



Naive Synchronous Queue - Fazit

- **Hohe Synchronisationskosten**
- Wake-Up aller Threads

- **Verbesserung:**
  Condition Objekte → weniger Wake-Ups
- Aber: Block bei jedem Aufruf bleibt
  → Weiterhin teuer

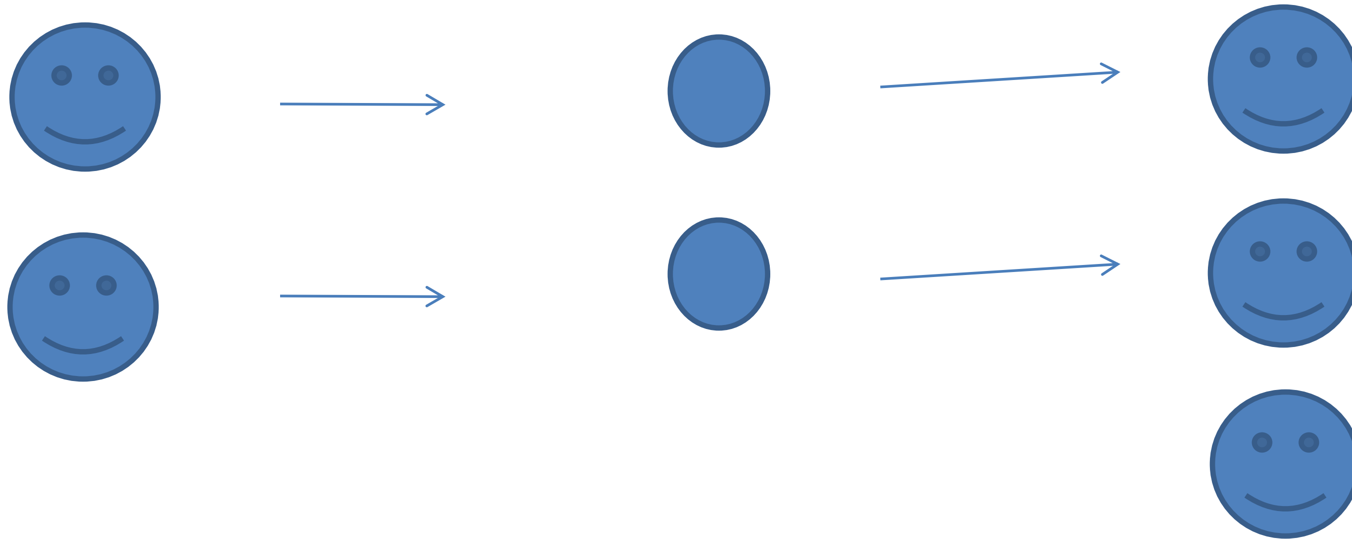
Dual Data Structure

- **Verbesserung der Naive Synchronous Queue**
- **Dequeuer**: setzt eine ***reservation*** in die Queue
  Enqueuer kann diese dann erfüllen (***fullfill***) und
  den Dequeuer benachrichtigen
- Umgekehrt genauso

→ die Queue enthält entweder
enq- oder deq-Reservierungen oder ist leer

→ **Fairness**, da die Reservierungen der Reihe nach
abgearbeitet werden

Dual Data Structures



reservation | reservation | fulfillment

Tag Cloud

Bounded vs. unbounded

Knoten

enq-Methode

Queues

Total

Synchronous

Stamps

Knoten Recycling

Erzeuger-Verbraucher-Problem

Lock-free

reservations & fullfillments

free list

Vorgänger helfen

ABA-Problem

deq-Methode

Partial

Tag Cloud

Bounded vs. unbounded

Knoten

enq-Methode

Queues

Total

Synchronous

Stamps

Knoten Recycling

Erzeuger-Verbraucher-Problem

Lock-free

reservations & fullfillments

free list

Vorgänger helfen

ABA-Problem

deq-Methode

Partial

Quellen

- M. Herlihy and N. Shavit. **The art of multiprocessor programming**. 2008, Morgan Kaufmann Publishers.
- D. Lea. Java community process, JSR 166, **concurrency utilities**. http://gee.cs.oswego.edu/dl/concurrency-interest/index.html, 2003.
- M. M. Michael and M. L. Scott. **Simple, fast, and practical non-blocking and blocking concurrent queue algorithms**. In Proc. of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 267–275, 1996, ACM Press.
- W. N. Scherer III, D. Lea, and M. L. Scott. **Scalable synchronous queues**. In PPoPP ’06: Proc. of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 147–156, NY, USA, 2006, ACM Press.
- T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea and D. Holmes. **Java Concurrency in Practice**, 2005, Addison-Wesley Professional.

Vielen Dank für Eure Aufmerksamkeit!