Freie Universität Berlin

Algorithmen und Programmierung IV: Nichtsequentielle Programmierung

Robert Tolksdorf

Freie Universität Berlin

Überblick

Überblick

- Interaktion über Objekte
  - Sperrsynchronisation
  - Kritische Abschnitte
  - Monitore
  - Nachrüsten von Ausschluss
  - Leser/Schreiber-Ausschluss
  - Sperroperationen

3.1.2 Monitore
(Dijkstra, Brinch Hansen, Hoare, 1972-1974)

Freie Universität Berlin

- Def.:
  Ein Monitor ist ein Objekt mit
  - vollständiger Datenabstraktion
  - vollständigem wechselseitigem Ausschluss zwischen allen seinen Operationen

- Ideale Syntax (nicht Java!) :
  - Ersetzung von class durch MONITOR oder SYNCHRONIZED CLASS
  - Vorteil: Sicherheit – Invariante bleibt garantiert erhalten!
  - Nachteil: evtl. unnötig restriktiv

3.1.2 Monitore

- Java praktiziert einen Kompromiss: synchronized ist als Modifier einer Methode in einer Klasse verwendbar:

- Syntax in Java:
  synchronized otherModifiers MethodDeclaration

- Semantik bei Objektmethoden:
  - als wäre der Rumpf ein kritischer Abschnitt mit synchronized(this)

- Semantik bei Klassenmethoden (static):
  - als wäre der Rumpf ein kritischer Abschnitt mit synchronized(className.class)

volatile

- Volatile-Deklaration

```
volatile long l;
```

- Gleiches Datum durch Ausschluss geschützt:

```
class VolatileLong {
private long v;
final synchronized void set(long l) {v=l;}
final synchronized long get() {return v;}
final synchronized void inc() {v++;}
}
do=new VolatileLong();
```

volatile

l	lobject
gleich bezüglich Ordnung, Sichtbarkeit	
l++; nicht atomar	lo.incr(); atomar
wenig Overhead per Zugriff	mehr Aufwand per Zugriff
mehr Overhead als synchronisierte Zugriffsfolge	Ein Block mit vielen Zugriffen mit einer Synchronisation

- volatile dann, wenn
  - Synchronisation nicht aus anderen Gründen gebraucht wird (Atomizität mehrerer Zugriffe, Konsistenz mit anderen Feldern, Schreibaktionen von einander unabhängig etc.)
  - Ein Schreiber, viele nebenläufige Leseprozesse
  - „Leichte Synchronisation“ konsistente Sichtbarkeit erfordert

3.1.2 Monitore

- Beispiel 1: tabellierte Abbildung

- Spezifikation:

```
interface Map<Key,Data> {
        // table of (Key,Data) pairs
void enter(Key k, Data d);
Data lookup(Key k);
void remove(Key k);
int  count();
}
```

- (realistisch: Ausnahmen berücksichtigen!)

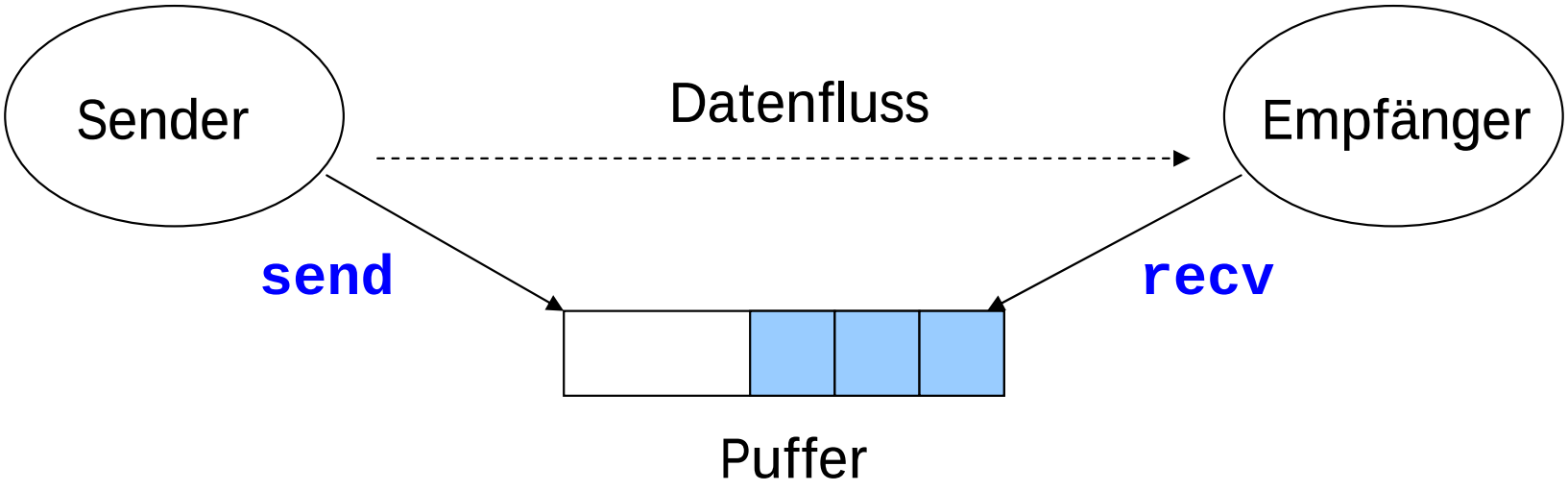
3.1.2 Monitore

- Implementierung:

```
class HashMap<K,D> implements Map<K,D> {
    .....
  public synchronized void enter(K k, D d) {
      .....
  }
  public synchronized D lookup(K k) {
      .....
  }
  public synchronized void remove(K k) {
      .....
  }
  public synchronized int  count() {
      .....
  }
}
```

Beispiel 2:

- Puffer (buffer) =
  Warteschlange (queue)
  von Nachrichten (messages)
  eines Senders (sender, producer)
  an einen Empfänger (receiver, consumer)
  oder auch mehrere Sender/Empfänger



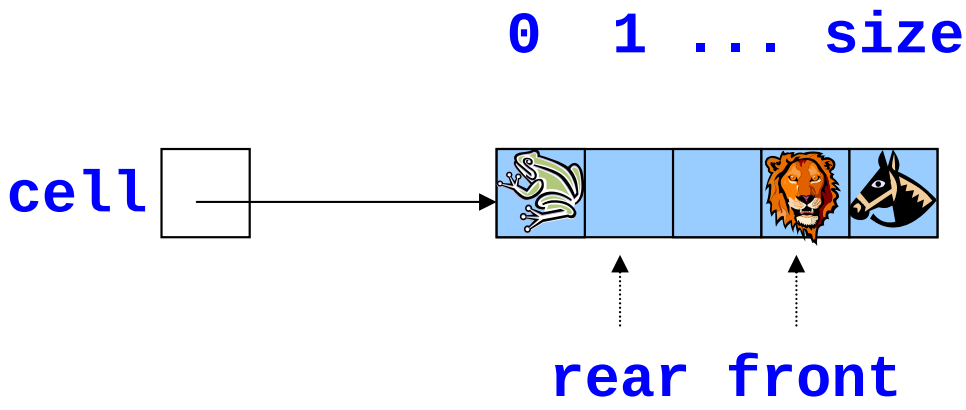
Spezifikation


```
interface Buffer<Message> {
        // like finite-capacity sequential Queue (ALP III!)

void send(Message m) throws Overflow;
        // like enqueue

Message recv() throws Underflow;
        // like dequeue
}
```

Repräsentation als linearer „Ringpuffer“ in Feld:




```
class LinearBuffer<M> implements Buffer<M> {
public LinearBuffer(int size) {
      this.size = size;
      cell = new M[size];}
private int size,count,front,rear;
private M[] cell;  // element container

   // invariant:
   // (front+count)%size == rear  &
   // 0 <= count <= size  & .....
```

Implementierung:

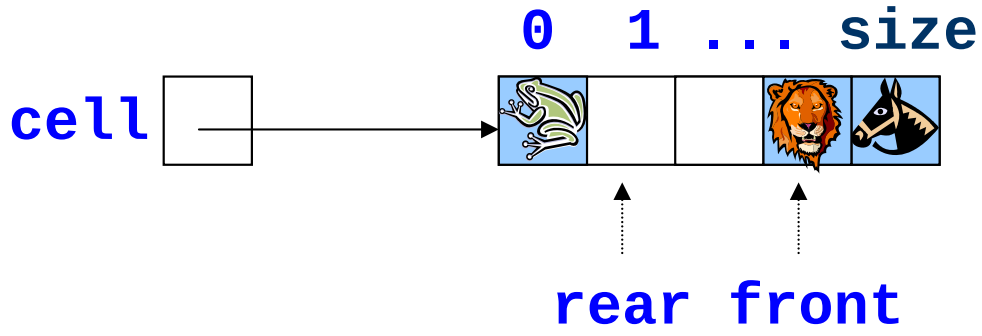
```
  public void send(M m) throws Overflow {
      if(count==size) throw new Overflow();
      cell[rear] = m;
      count++;
      rear = (rear+1)%size;
  }

  public M recv() throws Underflow{
      if(count==0) throw new Underflow();
      M m = cell[front];
      count--;
      front = (front+1)%size;
      return m;
  }
}
```

Implementierung

```
synchronized
  public void send(M m) throws Overflow {
      if(count==size) throw new Overflow();
      cell[rear] = m;
      count++;
      rear = (rear+1)%size;
  }
synchronized
  public M recv() throws Underflow{
      if(count==0) throw new Underflow();
      M m = cell[front];
      count--;
      front = (front+1)%size;
      return m;
  }
}
```

Repräsentation, die mit weniger Ausschluss auskommt ?




```
class LinearBuffer<M> implements Buffer<M> {
  public LinearBuffer(int size) {
       this.size = size;
       cell = new M[size+1];}
  private int size,front,rear;   // no count !
  private M[] cell;  // element container

    // invariant:
    // 0 <= front <= size  &  .....
```

Implementierung:

```
  int count() {
    return (rear-front+size+1)%(size+1);
  }
  public void send(M m) throws Overflow {
      synchronized(this) { // sender exclusion
          if(count()==size) throw new Overflow();
          cell[rear] = m;
          rear = (rear+1)%(size+1);}
  }
  public M recv() throws Underflow{
      synchronized(cell) { // receiver exclusion
          if(count()==0) throw new Underflow();
          M m = cell[front];
          front = (front+1)%(size+1);
          return m; }
  }
}
```

Bemerkungen

1. send und recv operieren stets auf verschiedenen Zellen des Feldes cell – dort kann es also keinen Konflikt zwischen send und recv geben, der zu Nichtdeterminismus führen würde.
2. send-Inkarnation werden gegeneinander ausgeschlossen, ebenso recv-Inkarnationen (wobei die Wahl der Objekte this und cell hier relativ willkürlich ist).
3. Die Tatsache, dass count nicht unteilbar ist, ist unproblematisch. Auch kann count jederzeit von anderen Stellen des Programms her aufgerufen werden.

Achtung:

- Die Lösung ist dennoch *nicht korrekt*,
  - weil beispielsweise bei leerem Puffer ein recv die Effekte der beiden Zuweisungen eines send in verkehrter Reihenfolge beobachten kann und damit aus einer Zelle liest, bevor sie beschrieben wird.
  - Dieses Problem ist Java-spezifisch und tritt bei anderen Sprachen nicht auf.

- Die Verwendung von volatile würde hier nicht helfen, da es nicht auf die Feldelemente bezogen werden kann.

… und drei Übungsfragen:

Freie Universität Berlin

1. Warum kann die Prüfung auf Über/Unterlauf nicht aus den kritischen Abschnitten herausgezogen werden?

2. Kann return m aus dem kritischen Abschnitt herausgezogen werden?

3. Wie muß eine zusätzliche Operation urgent aussehen, die eine dringliche Nachricht vorn im Puffer ablegt?

3.1.3 Nachrüsten von Ausschluss…

Freie Universität Berlin

- … bei Klassen, die für *sequentielle* Benutzung gebaut wurden,
- für Objekte, die *nichtsequentiell* benutzt werden,
- durch Bereitstellung eines Monitors, der als synchronisierte („*thread-safe*“) Version der Klasse eingesetzt werden kann,
- typischerweise mit *gleicher Schnittstelle*.

2 Alternativen:

1. *Vererbung:* Monitor ist *Unterklasse* der Originalklasse
2. *Delegation:* Monitor ist *Adapter* für die Originalklasse

- Beispiel:

```
interface SortedSet { ...
  boolean add     (Object x);
  boolean contains(Object x);
}
class TreeSet implements SortedSet { ...
  public boolean add     (Object x) {.....}
  public boolean contains(Object x) {.....}
}
```

voll synchronisiert, da die Implementierung des Originals i.a. unbekannt ist !

Vererbung und Delegation

Freie Universität Berlin

1.

```
class SyncTreeSet extends TreeSet {...
  public synchronized boolean add(Object x){
                        return super.add(x); }
  public synchronized boolean contains(Object x){
                    return super.contains(x); }
}
```

2.

```
class SyncTreeSet implements SortedSet {...
  private final SortedSet s = new TreeSet();
  public synchronized boolean add(Object x){
                        return s.add(x);     }
  public synchronized boolean contains(Object x){
                    return s.contains(x);     }
}
```

Fabriken (Factories)

Freie Universität Berlin

- Synchronisierte Adapter kann man über die *Fabrik* java.util.Collections erhalten, z.B. mittels

```
public static SortedSet synchronizedSortedSet
                              (SortedSet x) {
    return new SortedSet() {
        final SortedSet s = x;
        ...
    }
}
```

anonyme Klasse analog zu SyncTreeSet

- oder

```
public static List synchronizedList(List x) {
    return new List() {...}
}
```

- usw.

Java-Bibliotheken

- Beachte: Java-Bibliotheksklassen
  - sind teilweise synchronisiert (z.B. Vector),
  - *zum großen Teil aber nicht !*
  - *Insbesondere ist clone nicht synchronisiert !*

Collections mit Nebenläufigkeit

Freie Universität Berlin

- Collections ab Java 1.2
  - Schnittstellen List, Set, Map
  - Implementierungen HashMap, Hashtable, TreeMap, WeakHashMap, HashSet, TreeSet, Vector, ArrayList, LinkedList, etc.
  - Nebenläufigkeitsverträglich: Hashtable and Vector
  - Mit Collections.synchronizedMap, synchronizedList, synchronizedSet können die anderen sicher gemacht werden
  - Sperren der Collection normalerweise notwendig während Iteration (ConcurrentModificationException)
  - Schlechte Performance bei nebenläufigem Zugriff
- java.util.concurrent package
  - Führt neue nebenläufigkeitsverträgliche Implementierungen (ConcurrentHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet) ein. Diese sind auf Performance ausgelegt
  - Führt neue Schnittstellen ein:
    - BlockingQueue: Blockierende Entnahme

Collections aus java.util.concurrent

Freie Universität Berlin

- Collections aus java.util benutzen *fail-fast* Iteratoren: Wenn sich Collection während der Iteration verändert, wird ConcurrentModificationException geworfen und die Iteration abgebrochen
  - Problematisch beispielsweise wenn ein Vector eine Menge von Observers verwaltet: Eine laufende Benachrichtigung über eine Änderung würde abgebrochen wenn sich ein weiterer Observer nebenläufig registriert
- Erwünscht: Iteratoren *versuchen* nur, *eine* konsistente Sicht auf die Collection zu liefern
- Collections aus java.util.concurrent benutzen *weakly consistent* Iteratoren:
  - Elemente die nach dem Start der Iteration gelöscht wurden, werden nicht durch next() ausgeliefert
  - Elemente die nach dem Start der Iteration hinzugefügt wurden, werden oder werden nicht durch next() ausgeliefert
  - Kein Element wird doppelt ausgeliefert

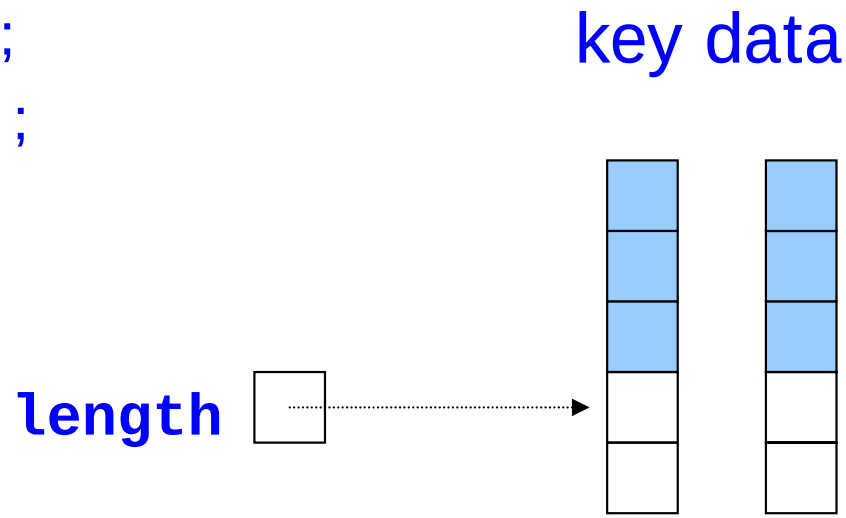
BlockingQueue

- Schnittstelle BlockingQueue (Unterschnittstelle von Collection<E>, Iterable<E>, Queue<E>)
  - Entnahme blockiert bis Element vorhanden
  - Ablegen blockiert bis Platz frei
- Implementierungen:
  - ArrayBlockingQueue
    Implementierung basierend auf einem (begrenzten) Feld
  - LinkedBlockingQueue
    Implementierung basierend auf (unbegrenzt vielen) verketteten Elementen
  - DelayQueue
    Elemente müssen eine bestimmte Zeit in der Schlange verbringen und können erst danach entnommen werden.
  - PriorityBlockingQueue
    Implementierung, die nach einer Rangordnung Elemente ausliefert (siehe PriorityQueue)
  - SynchronousQueue
    Implementierung mit Puffergröße=0
    - put() wartet auf take()
    - take() wartet auf put()

3.1.4 Leser/Schreiber-Ausschluss

- (reader/writer exclusion)
- Beispiel: (vgl. Map, HashMap, 3.1.2🡐)

```
class LinearMap<K,D> implements Map<K,D> {
public  LinearMap() {.....}
private int size;
private int length = 0;
private K[] key = new K[size];
private D[] data= new D[size];
```



Leser/Schreiber-Ausschluss


```
public synchronized void enter(K k, D d) {
      for (int i=0; i<length; i++)
          if (k==key[i])
            {data[i] = d; return;}
      if (length<size) {
         key[length] = k; data[length]= d;
         length++; }
      return;
}

public synchronized void remove(K k) {
      .....
}
```

Leser/Schreiber-Ausschluss


```
public synchronized D lookup(K k) {
      for(int i=0; i<length; i++)
         if(k==key[i]) return data[i];
      return null;
}

public synchronized int count() {
      return length;
}
```

- Ausschluss ist unnötig strikt: lookup, count lesen nur !
- → count braucht nicht synchronized zu sein
- → lookups könnten einander überlappen

Leser/Schreiber-Ausschluss

- Wünschenswertes Sprachkonstrukt anstelle von synchronized :
  - Statement = ... | **READING**(Expression) Block
    | **WRITING**(Expression) Block

- *Semantik:*
  Wie bei synchronized
  - wechselseitiger Ausschluss von Abschnitten, deren Expression sich auf das gleiche Objekt bezieht –
  - außer wenn beide READING sind
- Statische oder dynamische Schachtelung möglich, z.B. auch *upgrading:*
  READING(x){... **WRITING**(x){...} ... }

Leser/Schreiber-Ausschluss

- →

```
public WRITING void enter(K k, D d) {
    .....
}
public WRITING void remove(K k) {
    .....
}
public READING D lookup(K k) {
    .....
}
public int count() {
    .....
}
```

Leser/Schreiber-Ausschluss

Freie Universität Berlin

- Achtung:

- Die wünschenswerte Fairness (3.1.1🡐)
  ist u.U. *nicht garantiert:*
  - Wenn eine nicht abbrechende Reihe von Lesern sich „die Klinke in die Hand gibt“, kommen Schreiber nicht zum Zuge.

3.1.5 Sperroperationen

- Alternative zum Sperren mittels syntaktischer Klammerung:

Sperroperation;
  kritischer Abschnitt;
Entsperroperation;

spezielle *Anweisungen* zum Sperren/Entsperren

- Motivation?

Motivationen

1. *Sequentielle Sprache:* kennt keine Synchronisationskonstrukte ➔ Einsatz von *Bibliotheksroutinen*
2. *Übersetzerbau:* Letztlich müssen auch klammernde Konstrukte in *Anweisungen* umgesetzt werden.
3. *Restriktionen: reine Schachtelung* von kritischen Abschnitten kann u.U. zu restriktiv sein.

Sperroperationen

- (Nicht Java!)
- void lock()
  - *„setzt Sperre“*
  - sperrt einen objektbezogenen kritischen Abschnitt, gegebenenfalls nach Warten auf Freigabe durch einen anderen Prozess
- void unlock()
  - *„löst Sperre“*
  - gibt einen zuvor gesperrten kritischen Abschnitt frei
- synchronized(x){...}
  - ist fast äquivalent zu x.lock();...; x.unlock();

Sperroperationen

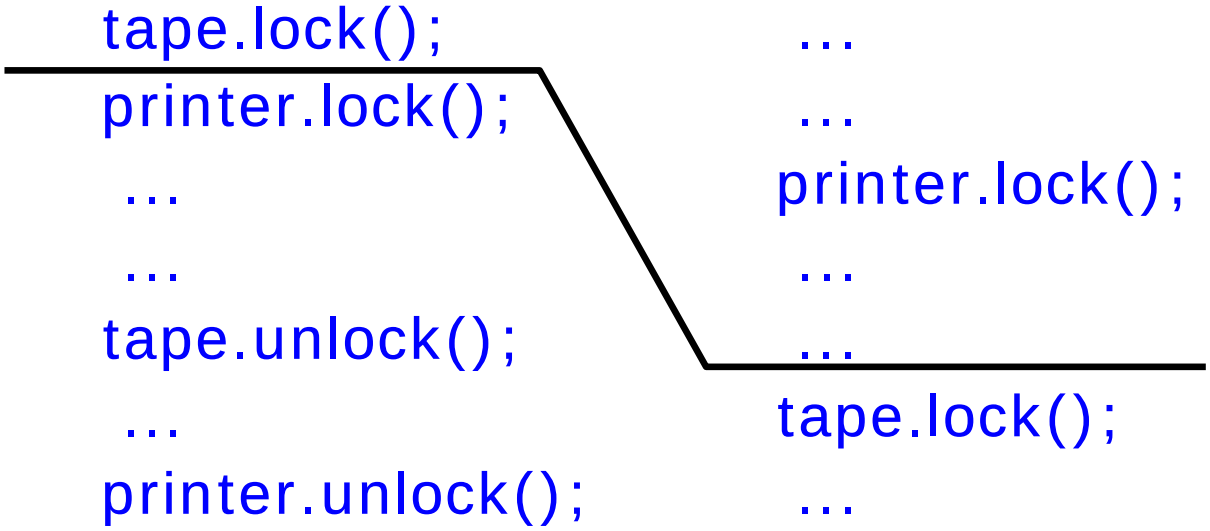
- Gefahr des Vergessens von unlock bedeutet *unsicheres Programmieren !*

- *Beispiele:*
  - lock();... **return**;... unlock();...
  - lock(); **if** (C) {... unlock();...} a = b; ...
  - **try** {lock(); .... unlock();...}
    catch(Exception e) {...}

Vorteile

- Sperren und Entsperren in *verschiedenen* Routinen möglich
- Zwang zur Schachtelung entfällt (Verklemmungsgefahr bleibt!)

- Beispiel:



Lese/Schreibsperren

- Sperroperationen für den Leser/Schreiber-Ausschluss:
  - Rlock() „setzt Lesesperre“
  - Wlock() „setzt Schreibsperre“
  - unlock() „löst (gibt frei) Sperre“
- Erlaubt ist auch *upgrading* (3.1.3←) einer Sperre :
  - Rlock();... Wlock();... unlock();

java.util.concurrent: Erweiterte Sperren

- Sperren mit synchronized sind
  - an Blockstruktur des Programms gebunden
  - in ihrem Verhalten (z.B. Fairness) nicht veränderbar
  - nicht sehr ausdrucksmächtig
- java.util.concurrent.locks führt ein
  - Sperroperationen
  - Verschiedene Implementierungen

Lock Schnittstelle

- Vorgesehene Methoden:
  - void lock()
    Sperre anfordern (blockiert)
  - void lockInterruptibly()
    Sperre anfordern, solange Thread nicht unterbrochen wird
  - boolean tryLock()
    Sperre anfordern, wenn Sperre frei
  - boolean tryLock(long time, TimeUnit unit)
    Sperre für einen bestimmten Zeitraum anfordern
  - void unlock()
    Sperre aufgeben

- Idiom:

```
Lock l = ...;
l.lock();
try {
  // access the resource protected by this lock
} finally {
  l.unlock();
}
```

ReentrantLock

- Entspricht den implizit bei jedem Objekt vorhandenen Sperren, aber explizit als Objekt
- Konstruktoren:
  - ReentrantLock()
  - ReentrantLock(boolean fair)
    Neue Sperre mit Fairness-Einstellung
- Zusätzliche Methoden:
  - isLocked()
    Hat ein Thread die Sperre?
  - protected Thread getOwner()
    Welcher Thread hat diese Sperre
  - boolean isHeldByCurrentThread()
    Hat dieser Thread die Sperre
  - int getHoldCount()
    wie oft hat der Thread die Sperre bekommen?
  - protected Collection<Thread> getQueuedThreads()
    Welche Threads wollen diese Sperre?
  - boolean hasQueuedThread(Thread thread)
    Will jener Thread diese Sperre

ReentrantReadWriteLock

- Lese-/Schreibsperren
- Konstruktoren
  - ReentrantReadWriteLock()
  - ReentrantReadWriteLock(boolean fair)
    Neue Sperre mit Fairness-Einstellung
- Zusätzliche Methoden:
  - ReentrantReadWriteLock.ReadLock readLock()
    Lesesperre erhalten
  - ReentrantReadWriteLock.WriteLock writeLock()
    Schreibsperre erhalten
- Verhalten
  - writeLock().lock() blockiert wenn andere Lese- oder Schreibsperren gesetzt sind
  - readLock().lock() blockiert wenn andere Schreibsperren gesetzt sind

Zusammenfassung

Zusammenfassung

- Interaktion über Objekte
  - Sperrsynchronisation
  - Kritische Abschnitte
  - Monitore
  - Nachrüsten von Ausschluss
  - Leser/Schreiber-Ausschlus
  - Sperroperationen