C Überblick über die 2. Übung

- Hinweise zur Aufgabe 1
- Threads
- Serialisierung

C.1 Hinweise zur 1. Aufgabe: StringTokenizer

- Schneidet Strings in Tokens
- Definiert in: **`java.util`**
- Beispiel:

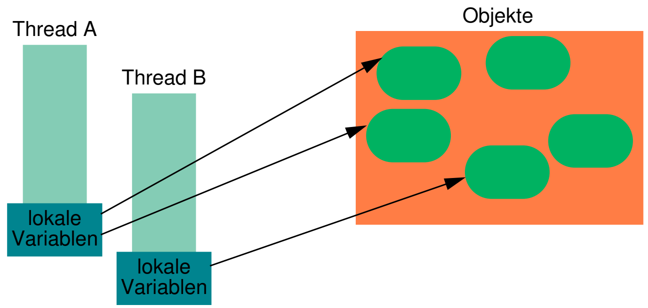
```
String str = "Hello this is a test"
StringTokenizer tokenizer = new StringTokenizer(str);

// Token einlesen bis zum Ende des Strings
while(tokenizer.hasMoreTokens()) {
        System.out.println(tokenizer.nextToken());
}
```


C.2 Threads

- Referenz:
  - D. Lea. *Concurrent Programming in Java - Design Principles and Patterns*. The Java Series. Addison-Wesley 1997.

1 Was ist ein Thread?

- Aktivitätsträger mit:
  - eigenem Instruktionszähler
  - eigenen Registern
  - eigenem Stack
- Alle Threads laufen im gleichen Adressbereich



2 Vorteile / Nachteile

- Vorteile:
  - ausführen paralleler Algorithmen auf einem Multiprozessorrechner
  - durch das Warten auf langsame Geräte (z.B. Netzwerk, Benutzer) wird nicht das gesamte Programm blockiert
- Nachteile:
  - komplexe Semantik
  - Fehlersuche sehr schwierig
  - John Ousterhout: *Why Threads Are A Bad Idea (for most purposes)*.

3 Thread Erzeugung (1)

1. Eine Unterklasse von **`java.lang.Thread`** erstellen.
2. Dabei die **`run()`**-Methode überschreiben.
3. Eine Instanz der Klasse erzeugen.
4. An dieser Instanz die Methode **`start()`** aufrufen.

- Beispiel:

```
class Test extends Thread {
    public void run() {
        System.out.println("Test");
    }
}

Test test = new Test();
test.start();
```


3 Thread Erzeugung (2)

1. Das Interface **`java.lang.Runnable`** implementieren.
   Dabei muss eine **`run()`**-Methode implementiert werden.
2. Ein Objekt instantiieren, welches das Interface **`Runnable`** implementiert.
3. Eine neue Instanz von **`Thread`** erzeugen, dem Konstruktor dabei das **`Runnable`**-Objekt mitgeben.
4. Am neuen Thread-Objekt die **`start()`**-Methode aufrufen.

- Beispiel:

```
class Test implements Runnable {
    public void run() {
        System.out.println("Test");
    }
}

Test test = new Test();
Thread thread = new Thread(test);
thread.start();
```


4 Die Methode `sleep`

- Ein Thread hat die Methode **`sleep(long n)`** um für **`n`** Millisekunden zu "schlafen".
- Der Thread kann jedoch verdrängt worden sein nachdem er aus dem **`sleep()`** zurückkehrt.

5 Die Methode `join`

- Ein Thread kann auf die Beendigung eines anderen Threads warten:

```
workerThread = new Thread(worker);
...
workerThread.join();
worker.result();
```


6 Daemon-Threads

- Daemon-Threads werden für Hintergrundaktivitäten genutzt
- Sie sollen nicht für die Hauptaufgabe einer Programmes verwendet werden
- Sobald alle *nicht-daemon* Threads beendet sind, ist auch das Programm beendet.
- Woran erkennt man, ob ein Thread ein Daemon-Thread sein soll?
  - Wenn man keine Bedingung für die Beendigung des Threads angeben kann.
- Wichtige Methoden der Klasse **`Thread`**:
  - **`setDaemon(boolean switch)`**: ein- oder ausschalten der Daemon-Eigenschaft
  - **`boolean isDaemon()`**: Prüft ob ein Thread ein Daemon ist.

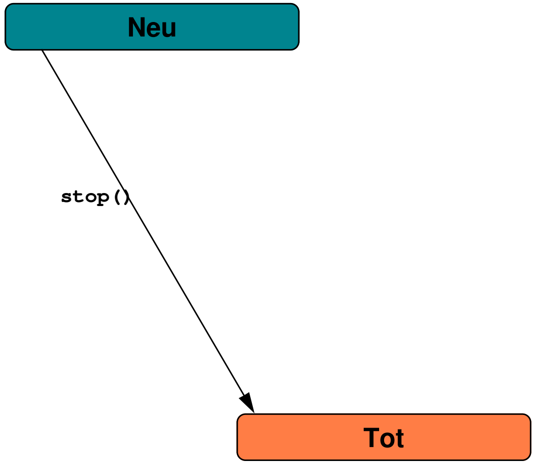
7 Die Klasse `ThreadGroup`

- Gruppe von verwandten Threads (**`ThreadGroup`**):
  - Eine Threadgruppe kann Threads enthalten und andere Threadgruppen.
  - Ein Thread kann nur Threads in der eigenen Gruppe beeinflussen.
- Methoden, die nur mit Threads der gleichen Gruppe angewendet werden können:
  - **`list()`**
  - **`stop()`**
  - **`suspend()`**
  - **`resume()`**

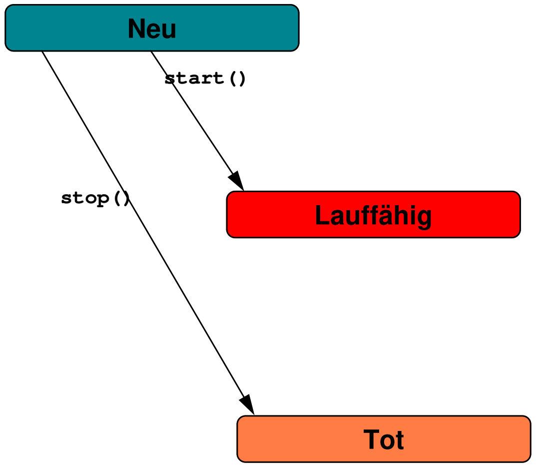
8 Zustände von Threads

Neu

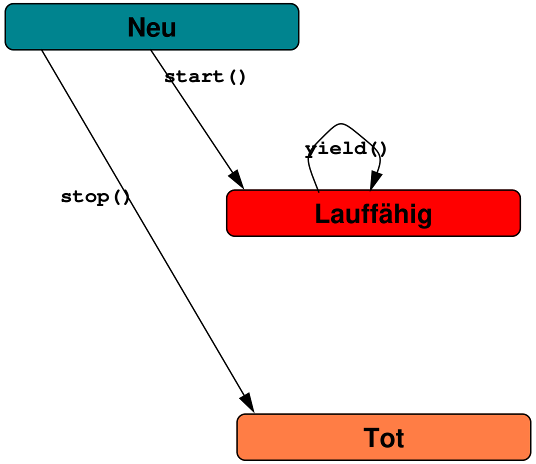
8 Zustände von Threads (2)



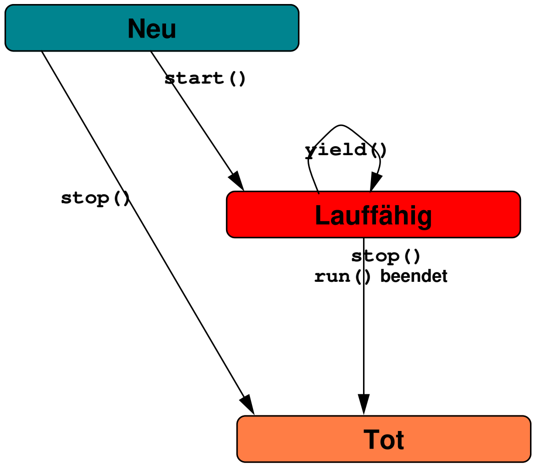
8 Zustände von Threads (3)



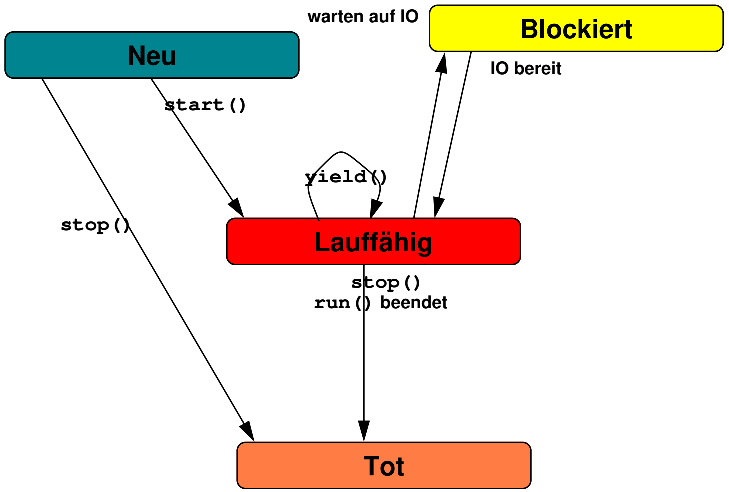
8 Zustände von Threads (4)



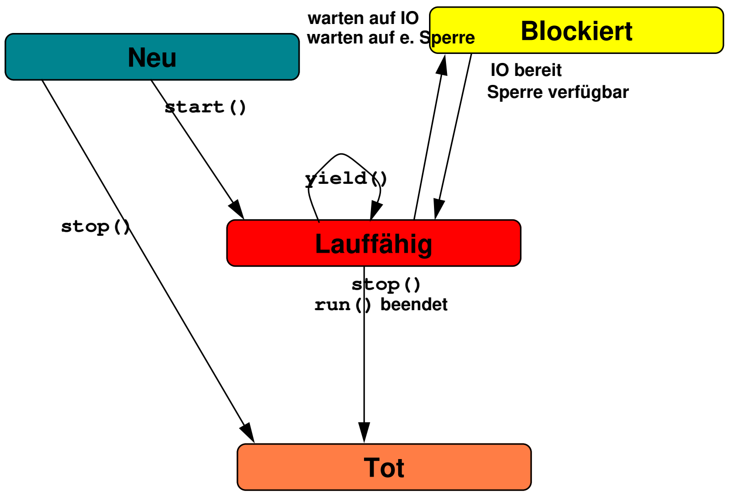
8 Zustände von Threads (5)



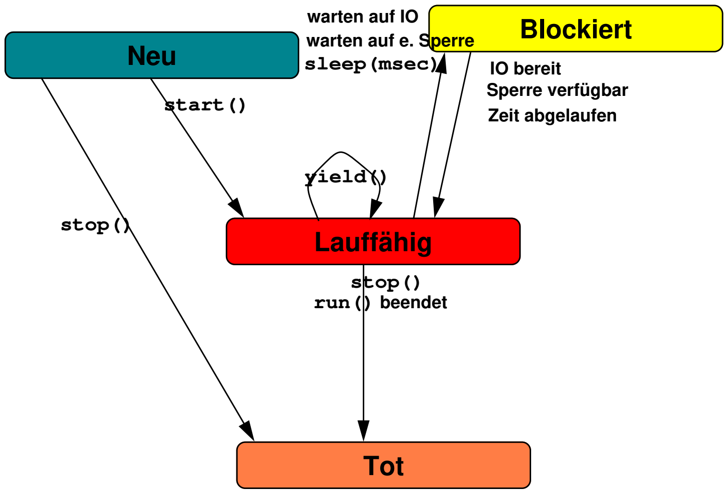
8 Zustände von Threads (6)



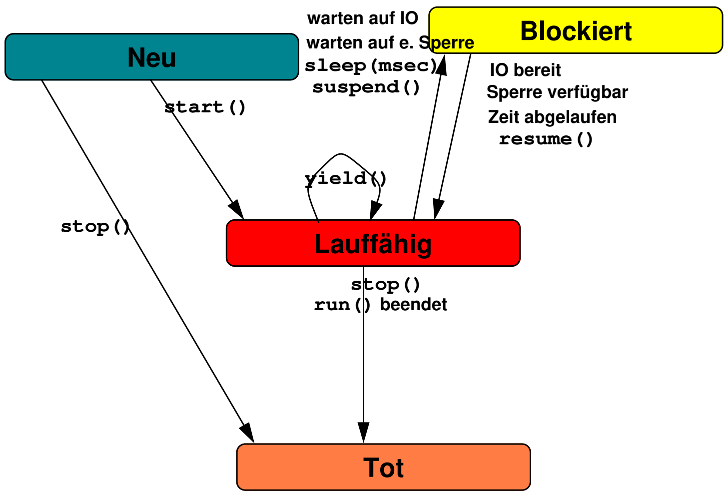
8 Zustände von Threads (7)



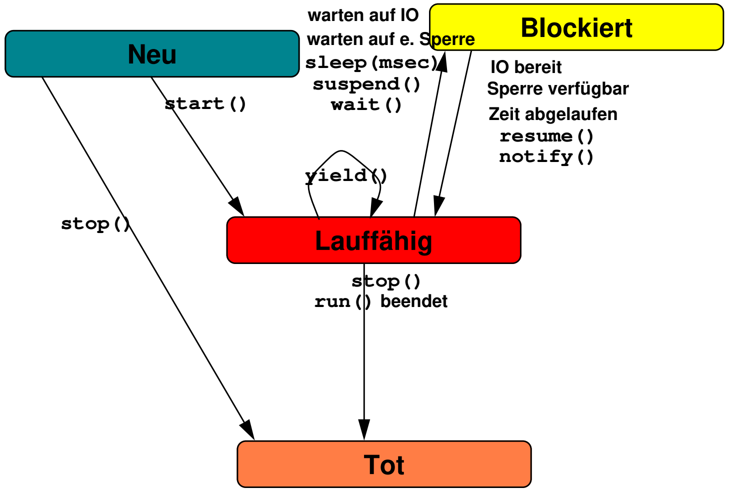
8 Zustände von Threads (8)



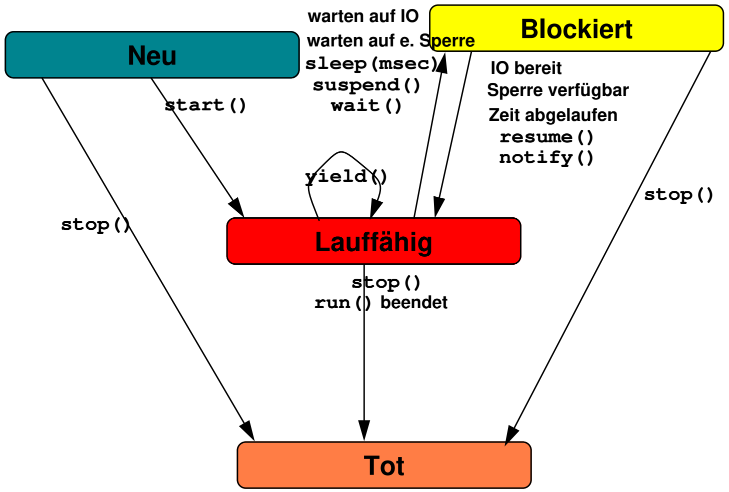
8 Zustände von Threads (9)



8 Zustände von Threads (10)



8 Zustände von Threads (11)



9 Veraltete Methoden der Klasse Thread

- `stop()`, `suspend()`, `resume()` sind seit Java 1.2 unerwünscht.
- `stop()` gibt alle Sperren des Thread frei - kann zu Inkonsistenzen führen
- `suspend()` und `resume()` können zu einem Deadlock führen:
  - suspend gibt keine Sperren frei
  - angehaltener Thread kann Sperren halten
  - Thread, der `resume` aufrufen will blockiert an einer Sperre

10 Multithreading Probleme

```
public class Test implements Runnable {
  public int a=0;
  public void run() {
    for(int i=0; i<100000; i++) {
      a = a + 1;
    }
  }                             Was ist das Ergebnis dieses Programmes?

  public static void main(String[] args) {
    Test value = new Test();
    Thread t1 = new Thread(value); zwei Threads erzeugen,
    Thread t2 = new Thread(value); mit demselben Runnable Objekt
    t1.start();   beide Threads starten
    t2.start();
    try {
      t1.join();   auf Beendigung der beiden Threads warten
      t2.join();
    } catch(Exception e) {
      System.out.println("Exception");
    }
    System.out.println("Expected a=200000 but a="+value.a);
  }
}
```

10 Multithreading Probleme (2)

- Ergebnis einiger Durchläufe: 173274, 137807, 150683
- Was passiert, wenn **a = a + 1** ausgeführt wird?

```
LOAD a into Register
ADD 1 to Register
STORE Register into a
```

- mögliche Verzahnung wenn zwei Threads beteiligt sind (initial a=0):
  - T1-load:a=0,Reg1=0
  - T2-load:a=0,Reg2=0
  - T1-add:a=0,Reg1=1
  - T1-store:a=1,Reg1=1
  - T2-add:a=1,Reg2=1
  - T2-store:a=1,Reg2=1
- Die drei Operationen müssen *atomar* ausgeführt werden!

11 Das Schlüsselwort synchronized

- Jedes Objekt kann als Sperre verwendet werden.
- Um eine Sperren anzufordern und freizugeben wird ein **synchronized** Konstrukt verwendet.
- Methoden oder Blöcke können als **synchronized** deklariert werden:

```
class Test {
    public synchronized void m() { ... }
    public void n() { ...
            synchronized(this) {
                ...
            }
    }
}
```

- ein Thread kann eine Sperre mehrfach halten (rekursive Sperre)
- verbessertes Beispiel: **synchronized(this) { a = a + 1; }**

12 Wann soll synchronized verwendet werden?

- **synchronized** ist nicht notwendig:
  - wenn Code immer nur von einem Thread ausgeführt wird (single-threaded context)
  - für einfache get-Methoden (siehe Ausnahmen unten)
- **synchronized** sollte verwendet werden:
  - wenn Daten geschrieben wird
  - wenn mit dem Objekt Berechnungen durchgeführt werden (auch wenn der Zustand *nur gelesen* wird)
  - für get-Methoden, die **long** oder **double** Typen zurückliefern
  - für einfache get-Methoden, die blockieren sollen wenn eine Zustandsveränderung durchgeführt wird

13 Synchronisationsvariablen (Condition Variables)

- Thread muss warten bis eine Bedingung wahr wird.
- zwei Möglichkeiten:
  - aktiv (polling)
  - passiv (condition variables)
- Jedes Objekt kann als Synchronisationsvariable verwendet werden.
- Die Klasse **Object** enthält Methoden um ein Objekt als Synchronisationsvariable zu verwenden.
  - **wait**: auf ein Ereignis warten

```
while(! condition) { wait(); }
```

  - **notify**: Zustand wurde verändert, die Bedingung könnte wahr sein, einen anderen Thread benachrichtigen
  - **notifyAll**: alle wartenden Threads aufwecken (teuer)

14 Warten und Sperren

- `wait` kann nur ausgeführt werden, wenn der aufrufende Thread eine Sperre an dem Objekt hält.
- `wait` gibt die Sperre frei bevor der Thread blockiert wird (atomar)
- beim Deblockieren wird die Sperre wieder atomar angefordert

15 Condition Variables - Beispiel

- PV-System: Bedingung: *count > 0*

```
class Semaphore {
    private int count;
    public Semaphore(int count) { this.count = count; }
    public synchronized void P() throws InterruptedException{
        while (count <= 0) {
            wait();
        }
        count--;
    }
    public synchronized void V() {
        count++;
        notify();
    }
}
```


15 Condition Variables - Beispiel (2)

- Bestellsystem: ein Thread akzeptiert Kundenabfragen (`SecretaryThread`) ein anderer Thread bearbeitet sie (`WorkerThread`)
- Secretary:

```
class SecretaryThread implements Runnable {
   public void run() {
      for(;;) {
         Customer customer = customerLine.nextCustomer();
         WorkerThread worker = classify(customer);
         worker.insertCustomer(customer);
      }
   }
}

interface WorkerThread {
   public void insertCustomer(Customer c);
}
```


15 Condition Variables - Beispiel (3)

- Worker:

```
class SpecificWorker implements Runnable, WorkerThread {
   public void run() {
      for(;;) {
         while(queue.empty())  // race condition!!
            synchronized (this) { wait(); }
         Customer customer = queue.next();
         // do something nice with customer
         // ...
      }
   }
   public void insertCustomer(Customer c) {
      queue.insert(c);
      this.notify();
   }
}
```

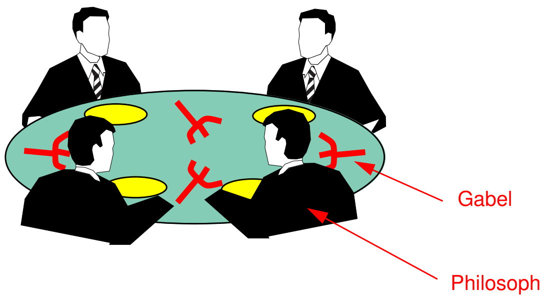
C.3 Korrektheit nebenläufiger Programme

- Safety: "Es passiert niemals etwas Schlechtes"
- Liveness: "Es passiert überhaupt etwas"

1 Safety

- gegenseitiger Ausschluss durch **`synchronized`**
- optimistische Nebenläufigkeitskontrolle

2 Deadlock: Das Philosophenproblem



denken → essen → denken → essen → ...

- ein Philosoph braucht beide Gabeln zum Essen
- alle Philosophen nehmen zuerst die rechte Gabel dann die linke → Verklemmung

3 Liveness

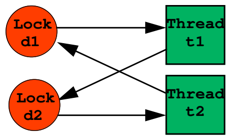
- keine Sprachunterstützung zur Deadlock-Verhinderung/Erkennung
- Deadlock-Beispiel:

```
class Deadlock implements Runnable {
  Deadlock other;
  void setOther(Deadlock other) {this.other = other; }
  synchronized void m() {
    try { Thread.sleep(1000); }
        catch(InterruptedException e) {}
    other.m();
  }
  public void run() { m(); }
}
```

3 Liveness (2)

- Verwendung, die zum Deadlock führt:

```
Deadlock d1 = new Deadlock();
Deadlock d2 = new Deadlock();
d1.setOther(d2); d2.setOther(d1);
Thread t1 = new Thread(d1);
t1.start();
Thread t2 = new Thread(d2);
t2.start();
try { t1.join(); t2.join(); } catch(InterruptedException e) {}
```



4 Deadlocks

- Counter

```
class Counter {
    private int count = 0;

    public synchronized void inc() { count++; }

    public int getCount() {return count; }

    public void setCount(int count) { this.count = count; }

    public synchronized void swap(Counter counter) {
        synchronized (counter) { // Deadlock Gefahr
            int tmp = counter.getCount();
            counter.setCount(count);
            count = tmp;
        }
    }
}
```


5 Deadlock - Vermeidung (1)

- Verhinderung zyklischer Ressourcenanforderung, Ordnung auf Locks

```
class Counter {
  ...
  public void swap(Counter counter) {
    Counter first = this;
    Counter second = counter;
    if (System.identityHashCode(this)
        < System.identityHashCode(counter)) {
      first = counter;
      second = this;
    }
    synchronized (first) {
      synchronized (second) {
        int tmp = counter.getCount();
        counter.setCount(count);
        count = tmp;
      }
    }
  }
```


5 Deadlock Vermeidung (2)

- Ressourcen (Locks) werden atomar angefordert:

```
class Counter {
  ...
  static Object lock = new Object();
  public void swap(Counter counter) {
    synchronized (lock) {
      synchronized(this) {
        synchronized (counter) {
          int tmp = counter.getCount();
          counter.setCount(count);
          count = tmp;
        }
      }
    }
  }
}
```

6 Nachteile der Java-Locks

- Methoden (lock, unlock) von Java-Locks sind unsichtbar, nur mit synchronized beeinflussbar
- kein Timeout beim Warten auf ein Lock möglich (Deadlock-Erkennung)
- Es können keine Unterklassen von Locks mit speziellem Verhalten erzeugt werden (Authentifizierung, Queueing-Strategien, ...)
- Locks können nicht referenziert werden (keine Deadlock-Erkennung oder Recovery möglich)

7 Optimistische Nebenläufigkeitskontrolle

- Vorteile:
  - keine Deadlocks
  - höhere Parallelität möglich
- Nachteile:
  - Designs werden komplexer
  - ungeeignet bei hoher Last
- Rollback/Recovery
  - Aktionen müssen umkehrbar sein, keine Seiteneffekte
  - zu jeder Methode muss es eine “Antimethode” geben
- Versioning
  - Methoden arbeiten auf shadow-Kopien des Objektzustandes
  - atomares commit überprüft, ob sich Ausgangszustand geändert hat (Konflikt) und setzt shadow-Zustand als neuen Objektzustand

7 Optimistisch - Beispiel (1)

- Counterzustand (Instanzvariablen) ist in separates Objekt ausgelagert (Memento Design-Pattern)

```
class CounterState {
  int count;
  CounterState(CounterState state) { ... }
  void inc() { ... }
  void dec() { ... }
  void swap(CounterState counter) { ... }
}
```


7 Optimistisch - Beispiel (2)

- Counter führt alle Operationen auf Kopie des Zustands aus
- am Ende wird die Kopie in einer atomaren Operation als Ist-Zustand gesetzt

```
class Counter {
  CounterState state;
  synchronized boolean commit(CounterState assumed,
                              CounterState next) {
    if (state != assumed) return false;
    state = next;
    return true;
  }
  void inc {
    do {
      assumed = state;
      next = new CounterState(assumed);
      next.inc();
    } while ( ! commit(assumed, next));
  }
}
```

C.4 Objekt Serialisierung

- Motivation:
  - Objekte sollen von der Laufzeit einer Anwendung unabhängig sein
  - Objekte sollen zwischen Anwendungen ausgetauscht werden können

1 Objektströme

- Mit Objektströmen können Objekte in Byteströme geschrieben werden und von dort gelesen werden.
- Klasse `java.io.ObjectOutputStream`
  - `void writeObject(Object o)`: Serialisierung eines Objekts (transitiv) (java.io.NotSerializableException)
- Klasse `java.io.ObjectInputStream`
  - `Object readObject()`: Lesen eines serialisierten Objekts (ClassNotFoundException)

2 Beispiel

- Speichern eines Strings und eines `Date`-Objekts:

```
FileOutputStream f = new FileOutputStream("/tmp/objects");
ObjectOutput s = new  ObjectOutputStream(f);
s.writeInt(42);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
f.close();
```

- Lesen der Objekte:

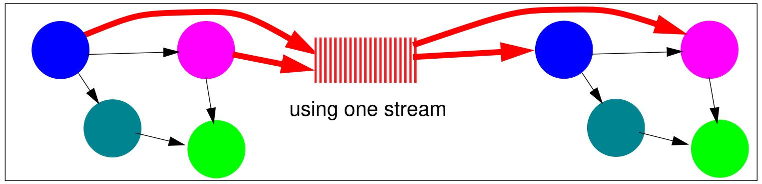
```
FileInputStream in = new FileInputStream("/tmp/objects");
ObjectInputStream s = new ObjectInputStream(in);
int i = s.readInt();
String today = (String)s.readObject();
Date date = (Date)s.readObject(); //! ClassNotFoundException
in.close();
```

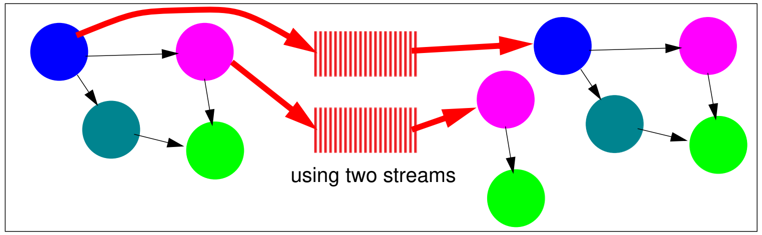

3 Schnittstellen

- “Marker Interface” `java.io.Serializable`:
  - Instanzvariablen werden automatisch gesichert
  - Variablen, die mit `transient` deklariert wurden, werden nicht gesichert
- Interface `java.io.Externalizable`:
  - Ein Objekt kann seine Serialisierung selbst vornehmen
  - folgende Methoden müssen implementiert werden:
    - `writeExternal(ObjectOutput out)`
    - `readExternal(ObjectInput in)`

4 Probleme

- Alle Objekte eines Objekt-Graphen sollten in den gleichen Strom geschrieben werden, ansonsten werden die Objekte beim lesen dupliziert





4 Probleme (2)

- der Objekt-Graph muss atomar geschrieben werden
- Klassen werden nicht gespeichert: sie müssen verfügbar sein, wenn ein Objekt später wieder eingelesen wird
- statische Elemente werden nicht gesichert
  - ◆ Lösung: Serialisierung beeinflussen:
    - private void writeObject(java.io.ObjectOutputStream out)
    - private void readObject(java.io.ObjectInputStream in)

5 Versionskontrolle der Klassen

- serialisierte Objekte müssen mit der “richtigen” Klasse gelesen werden
- serialisierte Objekte enthalten dazu eine Klassenreferenz, welche den Namen und eine Versionsnummer der Klasse enthält
- Die Versionsnummer wird durch einen Hash-Wert repräsentiert, der über den Klassennamen, die Schnittstellen und die Namen der Instanzvariablen und Methoden gebildet wird.
- Problem: kleine Änderungen an der Klasse führen dazu, dass alte serialisierte Objekte unlesbar sind.
- Lösung:
  - ◆ eine Klasse kann ihre Versionsnummer festlegen:
    `static final long serialVersionUID = 1164397251093340429L;`
  - ◆ initiale Versionsnummer kann mittels `serialver` berechnet werden.
  - ◆ ⇒ Versionsnummer nur nach inkompatiblen Änderungen verändern