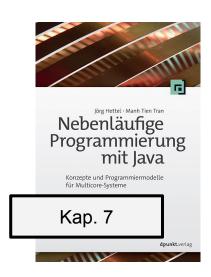
## Parallele Programmierung

Atomic Variablen

### Überblick

- Test, Musterlösungen
- java.util.concurrent.atomic
- "compare-and-set"
  - Atomare Skalare (AtomicInt, AtomicLong, AtomicBoolean)
  - Laborübung: Counter mit synchronized und mit Compare-And-Set
- Atomare double und float
- beliebige atomare Funktionen
- Atomare Referenzen



Folie mit Anmerkungen

### **Atomarer Zugriff**

lesen/schreiben von Variablen

- bei primitiven Datentypen außer long und double atomar
- bei volatile immer atomar
- Variablen mit Verweisen auf Objekte immer atomar

Bei atomarem Zugriff wird der Wert nie inkonsistent.

Lesen während anderer Thread ändert:

Ergebnis ist Wert vor Änderung oder nach Änderung.

### **Increment/Decrement**

Increment (++)/Decrement (--) sind nicht atomar, Synchronisation ist erforderlich.

```
class SyncCounter {
    private int counter = 0;
    public synchronized void increment() {
        counter++;
                                                  public synchronized void increment();
                                                   Code:
                                                      0: aload 0
    public synchronized void decrement()
                                                      1: dup
        counter--;
                                                      2: getfield #12 // Field counter:I
                                                      5: iconst_1
                                                      6: iadd
    public synchronized int get() {
                                                      7: putfield #12 // Field counter:I
        return counter;
                                                     10: return
```

sehr ineffizient (hoher Anteil an gegenseitigem Ausschluss)

### **Compare-And-Set**

Wenn der Inhalt der Speicherzelle mit dem erwarteten, alten Wert übereinstimmt, wird der neue an die Speicherstelle geschrieben (atomare Operation).

Stimmt der erwartete, alte Wert nicht mit dem aktuellen Wert überein, da ihn z.B. ein anderer Thread zwischenzeitlich geändert hat, findet keine Modifikation statt, dann wäre der Rückgabewert false, sonst true.

Inkrement mit compareAndSet, kommt ohne Locking aus:

```
int tmp = counter;
while (! compareAndSet( counter, tmp, tmp+1)) { // 2
    tmp = counter;
}
[Pseudocode, nicht Java!]
// 3
```

Man versucht es solange, bis zwischen 1 und 2 bzw. 3 und 2 keine andere Änderung von counter erfolgt ist.

### AtomicBoolean/Integer/Long

AtomicBoolean, AtomicInteger und AtomicLong sammeln atomare Operationen für die Java-Benutzung

#### Konstruktoren:

- AtomicBoolean/Integer/Long()
- AtomicBoolean/Integer/Long(boolean/int/long initialValue)

#### Lesen/Schreiben (alle atomar):

- public final boolean/int/long get()
- public final void set(boolean/int/long newValue)
- public final boolean/int/long getAndSet(boolean/int/long newValue)
   liefert den alten Wert zurück

#### Compare-And-Set (atomar):

 public final boolean compareAndSet(boolean/int/long expect, boolean/int/long update)

### **Atomare double und float**

gibt es nicht: "You can also hold floats using Float.floatToIntBits and Float.intBitstoFloat conversions, and doubles using Double.doubleToLongBits and Double.longBitsToDouble conversions."

```
public AtomicFloat(final float initialValue) {
    this.bits = new AtomicInteger(Float.floatToIntBits(initialValue));
public final boolean compareAndSet(final float expect, final float update) {
    return this.bits.compareAndSet(Float.floatToIntBits(expect),
                                   Float.floatToIntBits(update));
public final void set(final float newValue) {
    this.bits.set(Float.floatToIntBits(newValue));
public final float get() {
    return Float.intBitsToFloat(this.bits.get());
```

#### Folie mit Anmerkungen

# Beliebige atomare Funktion seit Java 8 in Atomic\*

```
public final long updateAndGet(LongUnaryOperator updateFunction)
public final int updateAndGet(IntUnaryOperator updateFunction)
@FunctionalInterface
public interface LongUnaryOperator {
    long applyAsLong(long operand);
@FunctionalInterface
public interface IntUnaryOperator {
    long applyAsInt(int operand);
final var i = new AtomicInteger(4711);
i.updateAndGet((in) -> (in * x));
final var 1 = new AtomicLong(4711);
1.updateAndGet((in) -> (in * x));
```