

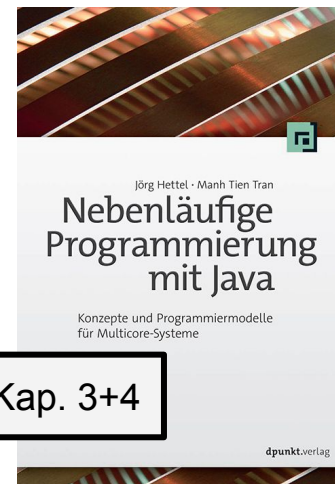
Parallele Programmierung



Konkurrierender Zugriff

Überblick

- Konkurrerender Zugriff auf Daten
 - Java Speichermodell
 - Instanzvariablen, kritische Abschnitte
 - sequentielle Konsistenz und Memory Barrieren (u.a. “volatile”)
 - **25 Minuten Laboraufgabe + 10 Minuten Lösungsskizze**
 - Thread-lokaler Speicher
 - **20 Minuten Laboraufgabe + 5 Minuten Lösungsskizze**
- Gegenseitiger Ausschluss
 - synchronized-Methoden und -Blöcke
 - Lock-Objekte
 - **15 Minuten Laboraufgabe + 10 Minuten Lösungsskizze**
 - Thread-Sicherheit von Vector
 - Deadlock durch zyklischen gegenseitigen Ausschluss



Konkurrierender Zugriff auf Daten

Java Speichermodell 1/2

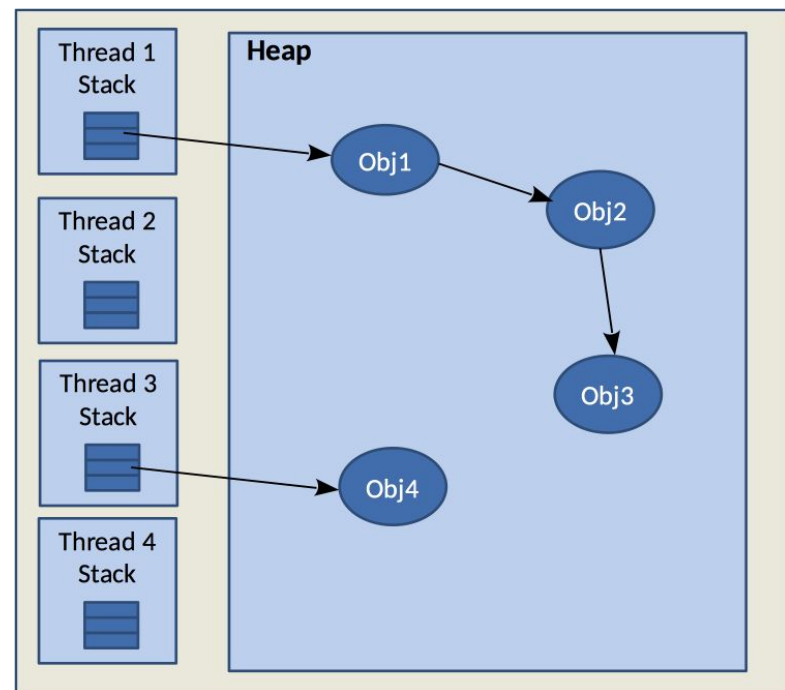
Folie mit
Anmerkungen

Stack (pro Thread):

- Parameter, die einer Methode übergeben werden
- lokale Variablen
 - Referenzen auf Objekte (im Heap)
 - Variablen von primitiven Typen
- etwaiger Rückgabewert

Heap (global: für alle Threads):

- Objekte
 - Instanzvariablen
 - Klassenvariablen (`static`)



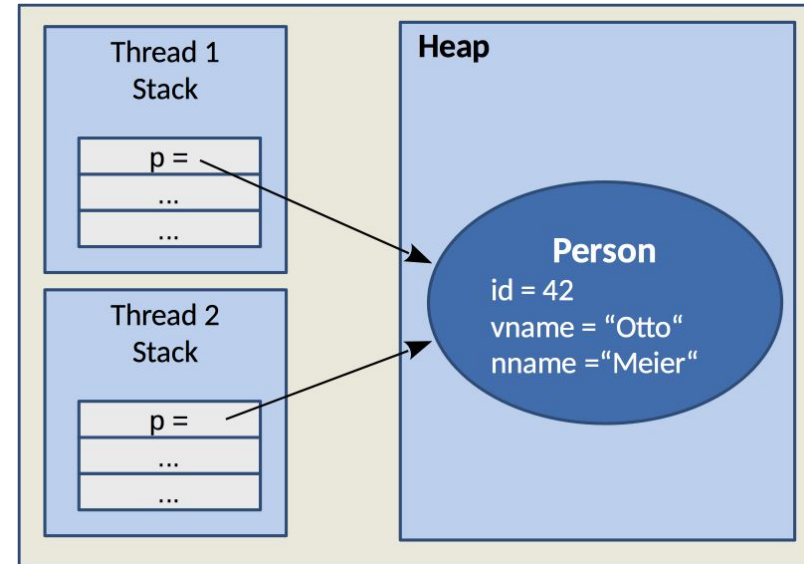
Java Speichermodell 1/2

Stack (pro Thread):

- Parameter, die einer Methode übergeben werden
- lokale Variablen
 - Referenzen auf Objekte (im Heap)
 - Variablen von primitiven Typen
- etwaiger Rückgabewert

Heap (global: für alle Threads):

- Objekte
 - Instanzvariablen
 - Klassenvariablen (`static`)



Java Speichermodell 2/2

```
public class RamTest extends Thread {
    private int i;
    public RamTest(final int i) {
        this.i = i;
    }
    public void print(final int i) {
        final var a = i * i;
        final var b = new Integer(a);
        System.out.println(b);
    }
    @Override
    public void run() {
        print(this.i);
    }
    public static void main(final String[] args) {
        new RamTest(2).start();
        new RamTest(3).start();
    }
}
```

Java Speichermodell 2/2

```
public class RamTest extends Thread {  
    private int i;  
    public RamTest(final int i) {  
        this.i = i;  
    }  
    public void print(final int i) {  
        final var a = i * i;  
        final var b = Integer.valueOf(a);  
        System.out.println(b);  
    }  
    @Override  
    public void run() {  
        print(this.i);  
    }  
    public static void main(final String[] args) {  
        new RamTest(2).start();  
        new RamTest(3).start();  
    }  
}
```

Java Speichermodell 2/2

```
public class RamTest extends Thread {  
    private int i;  
    public RamTest(final int i) {  
        this.i = i;  
    }  
    public void print(final int i) {  
        final var a = i * i;  
        final var b = Integer.valueOf(a);  
        System.out.println(b);  
    }  
    @Override  
    public void run() {  
        print(this.i);  
    }  
    public static void main(final String[] args) {  
        new RamTest(2).start();  
        new RamTest(3).start();  
    }  
}
```

Heap:

main-Thread Stack:

args = { }

Java Speichermodell 2/2

```
public class RamTest extends Thread {  
    private int i;  
    public RamTest(final int i) {  
        this.i = i;  
    }  
    public void print(final int i) {  
        final var a = i * i;  
        final var b = Integer.valueOf(a);  
        System.out.println(b);  
    }  
    @Override  
    public void run() {  
        print(this.i);  
    }  
    public static void main(final String[] args) {  
        new RamTest(2).start();  
        new RamTest(3).start();  
    }  
}
```

Heap:

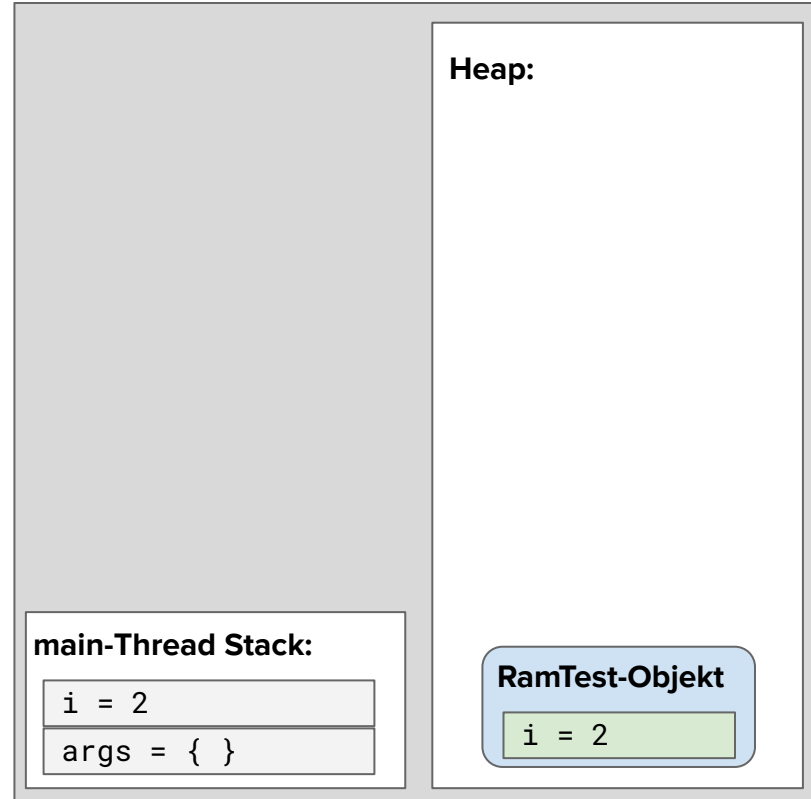
main-Thread Stack:

i = 2

args = { }

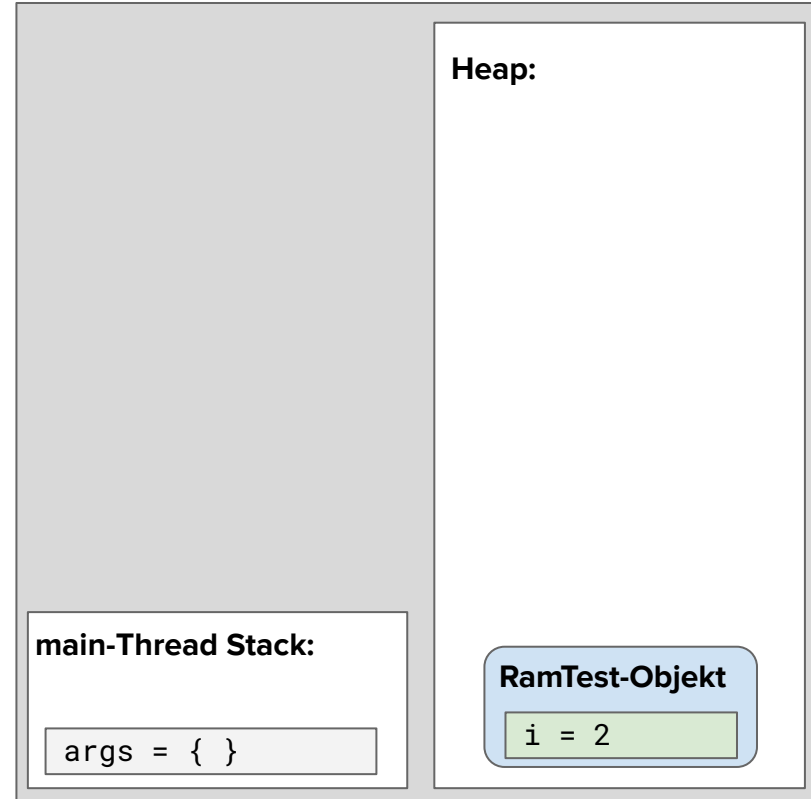
Java Speichermodell 2/2

```
public class RamTest extends Thread {  
    private int i;  
    public RamTest(final int i) {  
        this.i = i;  
    }  
    public void print(final int i) {  
        final var a = i * i;  
        final var b = Integer.valueOf(a);  
        System.out.println(b);  
    }  
    @Override  
    public void run() {  
        print(this.i);  
    }  
    public static void main(final String[] args) {  
        new RamTest(2).start();  
        new RamTest(3).start();  
    }  
}
```



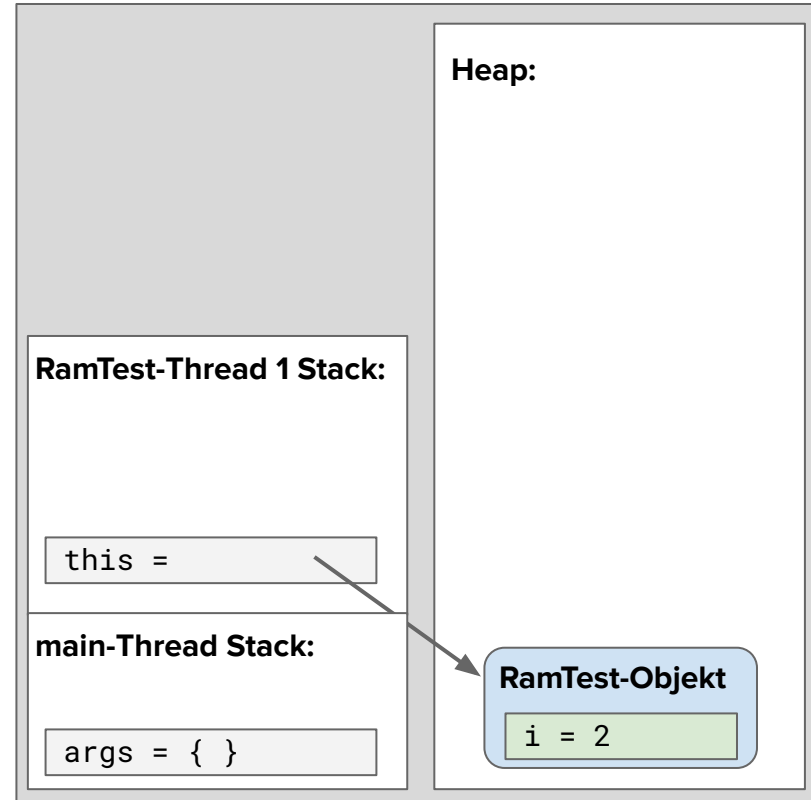
Java Speichermodell 2/2

```
public class RamTest extends Thread {  
    private int i;  
    public RamTest(final int i) {  
        this.i = i;  
    }  
    public void print(final int i) {  
        final var a = i * i;  
        final var b = Integer.valueOf(a);  
        System.out.println(b);  
    }  
    @Override  
    public void run() {  
        print(this.i);  
    }  
    public static void main(final String[] args) {  
        new RamTest(2).start();  
        new RamTest(3).start();  
    }  
}
```



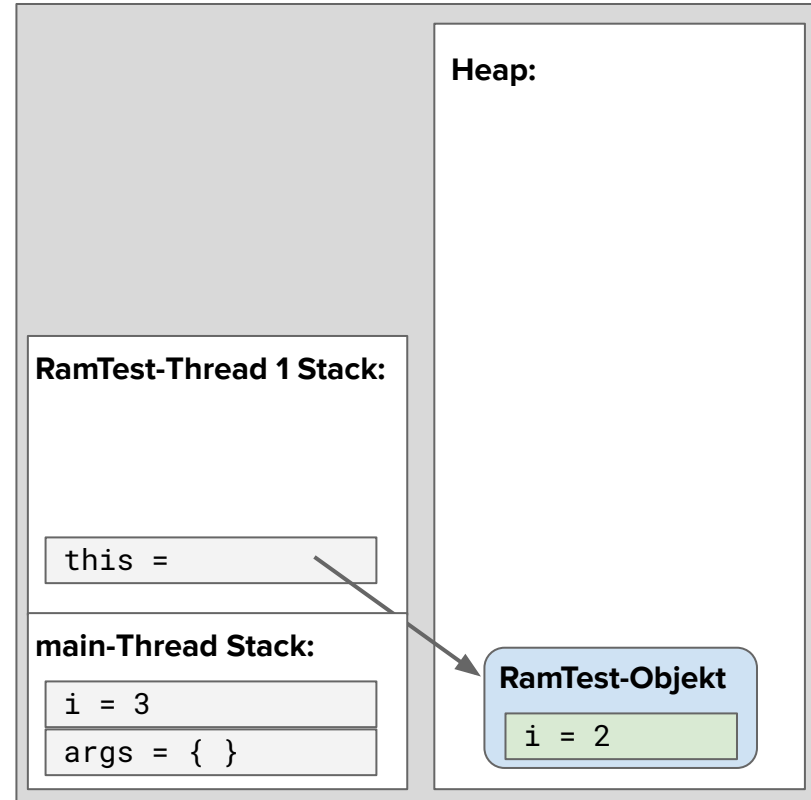
Java Speichermodell 2/2

```
public class RamTest extends Thread {  
    private int i;  
    public RamTest(final int i) {  
        this.i = i;  
    }  
    public void print(final int i) {  
        final var a = i * i;  
        final var b = Integer.valueOf(a);  
        System.out.println(b);  
    }  
    @Override  
    public void run() {  
        print(this.i);  
    }  
    public static void main(final String[] args) {  
        new RamTest(2).start();  
        new RamTest(3).start();  
    }  
}
```



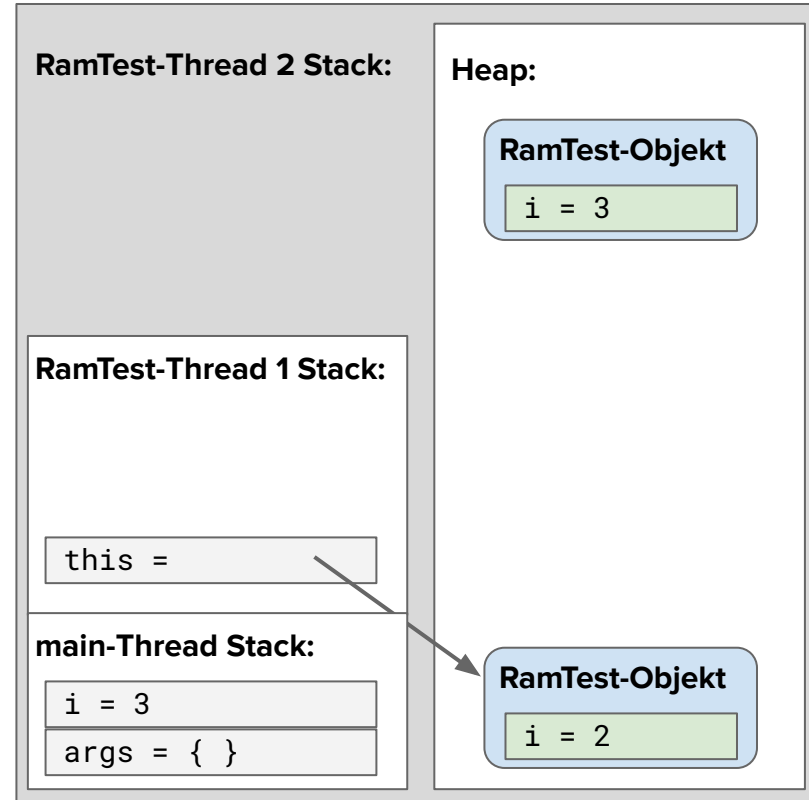
Java Speichermodell 2/2

```
public class RamTest extends Thread {  
    private int i;  
    public RamTest(final int i) {  
        this.i = i;  
    }  
    public void print(final int i) {  
        final var a = i * i;  
        final var b = Integer.valueOf(a);  
        System.out.println(b);  
    }  
    @Override  
    public void run() {  
        print(this.i);  
    }  
    public static void main(final String[] args) {  
        new RamTest(2).start();  
        new RamTest(3).start();  
    }  
}
```



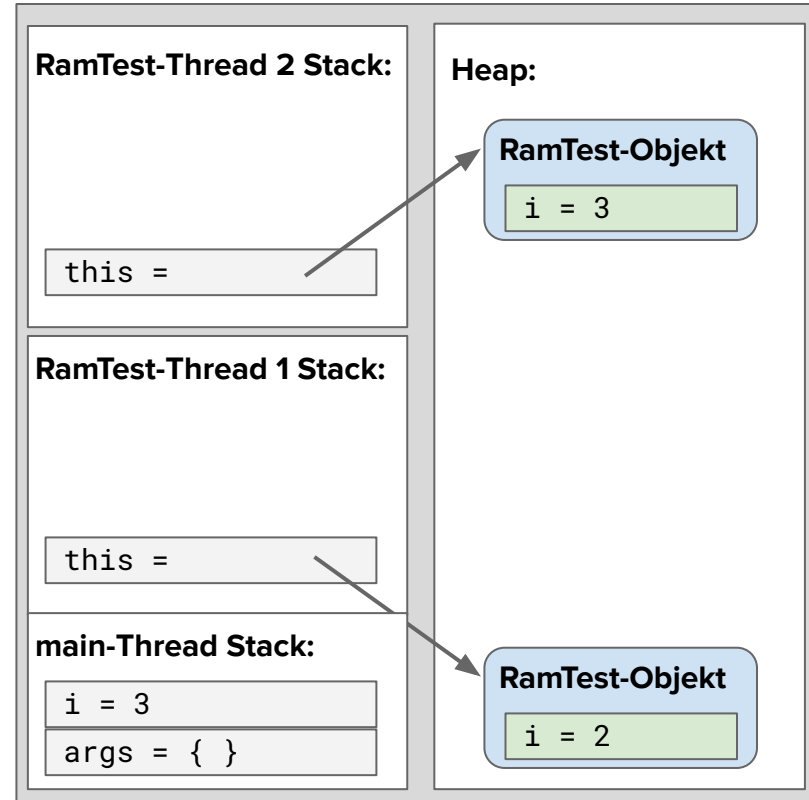
Java Speichermodell 2/2

```
public class RamTest extends Thread {  
    private int i;  
    public RamTest(final int i) {  
        this.i = i;  
    }  
    public void print(final int i) {  
        final var a = i * i;  
        final var b = Integer.valueOf(a);  
        System.out.println(b);  
    }  
    @Override  
    public void run() {  
        print(this.i);  
    }  
    public static void main(final String[] args) {  
        new RamTest(2).start();  
        new RamTest(3).start();  
    }  
}
```



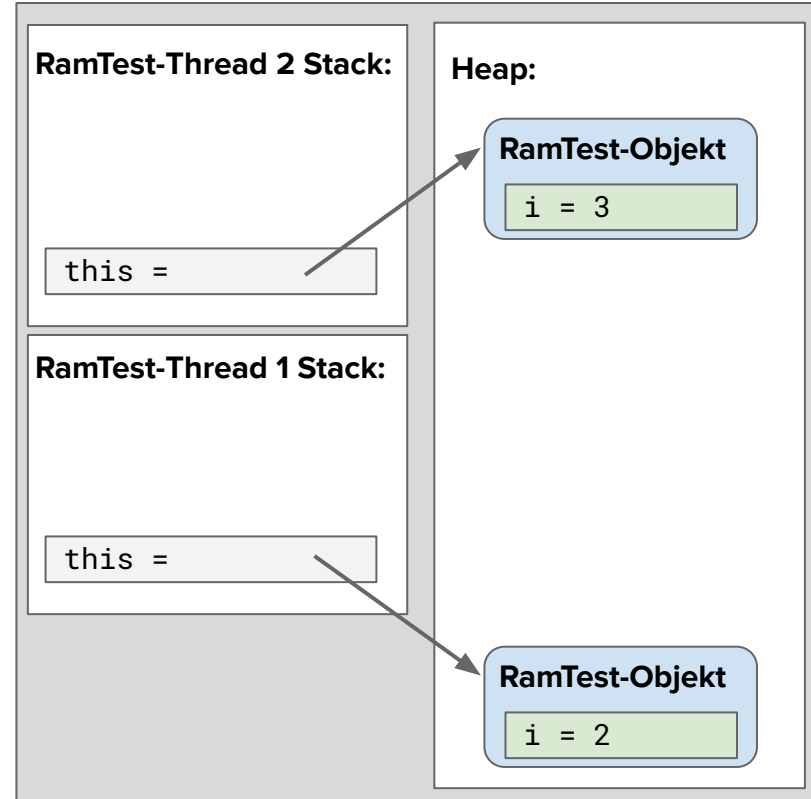
Java Speichermodell 2/2

```
public class RamTest extends Thread {  
    private int i;  
    public RamTest(final int i) {  
        this.i = i;  
    }  
    public void print(final int i) {  
        final var a = i * i;  
        final var b = Integer.valueOf(a);  
        System.out.println(b);  
    }  
    @Override  
    public void run() {  
        print(this.i);  
    }  
    public static void main(final String[] args) {  
        new RamTest(2).start();  
        new RamTest(3).start();  
    }  
}
```



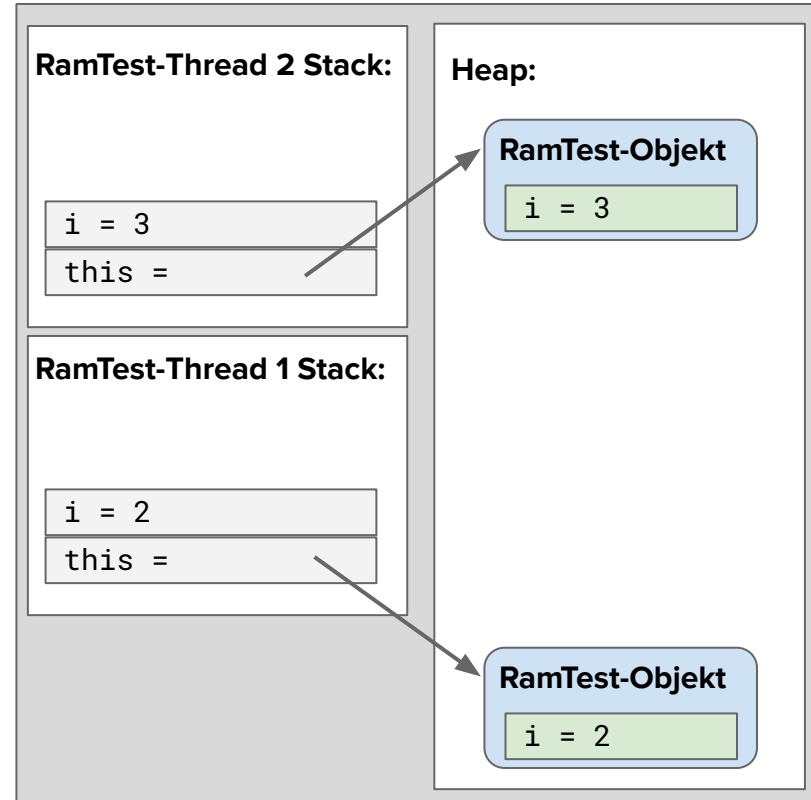
Java Speichermodell 2/2

```
public class RamTest extends Thread {  
    private int i;  
    public RamTest(final int i) {  
        this.i = i;  
    }  
    public void print(final int i) {  
        final var a = i * i;  
        final var b = Integer.valueOf(a);  
        System.out.println(b);  
    }  
    @Override  
    public void run() {  
        print(this.i);  
    }  
    public static void main(final String[] args) {  
        new RamTest(2).start();  
        new RamTest(3).start();  
    }  
}
```



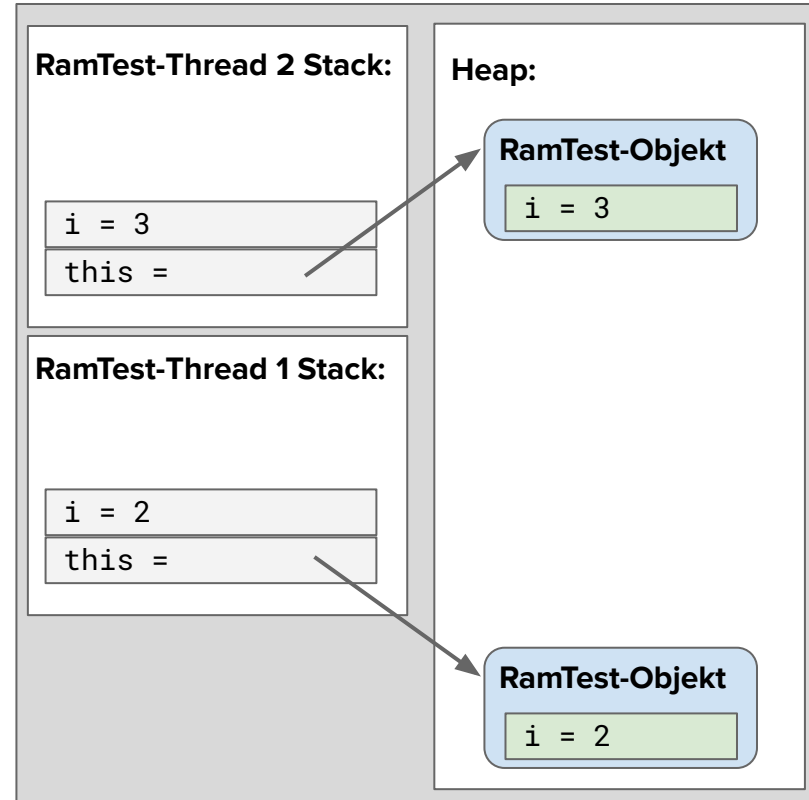
Java Speichermodell 2/2

```
public class RamTest extends Thread {  
    private int i;  
    public RamTest(final int i) {  
        this.i = i;  
    }  
    public void print(final int i) {  
        final var a = i * i;  
        final var b = Integer.valueOf(a);  
        System.out.println(b);  
    }  
    @Override  
    public void run() {  
        print(this.i);  
    }  
    public static void main(final String[] args) {  
        new RamTest(2).start();  
        new RamTest(3).start();  
    }  
}
```



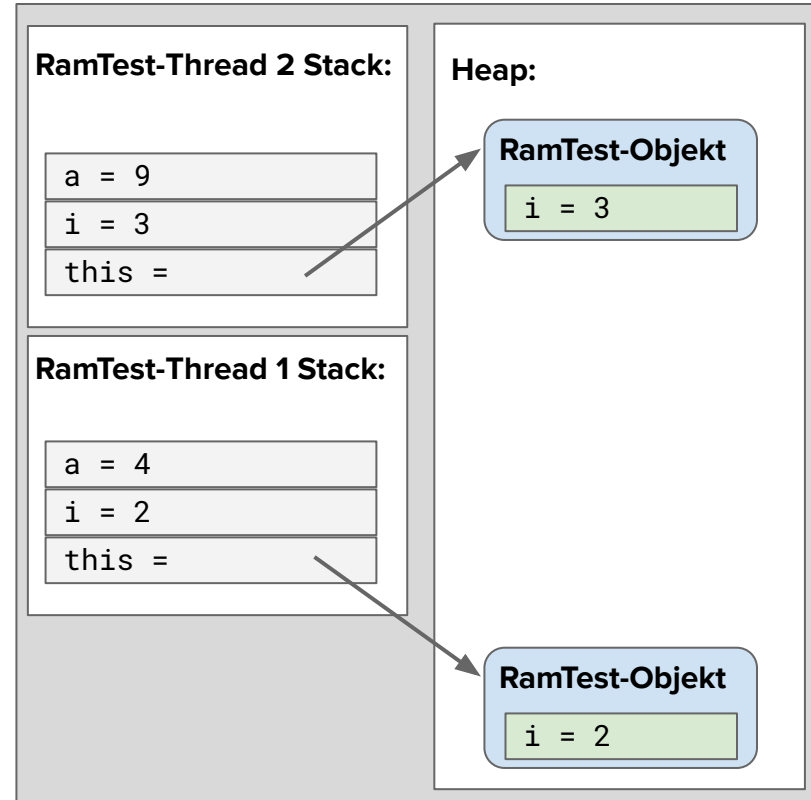
Java Speichermodell 2/2

```
public class RamTest extends Thread {  
    private int i;  
    public RamTest(final int i) {  
        this.i = i;  
    }  
    public void print(final int i) {  
        final var a = i * i;  
        final var b = Integer.valueOf(a);  
        System.out.println(b);  
    }  
    @Override  
    public void run() {  
        print(this.i);  
    }  
    public static void main(final String[] args) {  
        new RamTest(2).start();  
        new RamTest(3).start();  
    }  
}
```



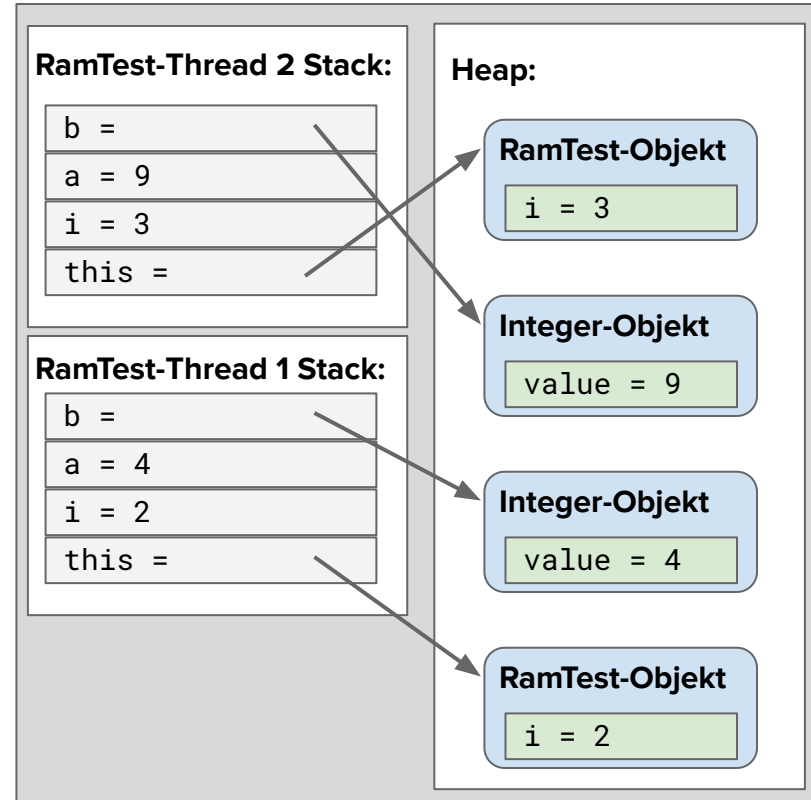
Java Speichermodell 2/2

```
public class RamTest extends Thread {  
    private int i;  
    public RamTest(final int i) {  
        this.i = i;  
    }  
    public void print(final int i) {  
        final var a = i * i;  
        final var b = Integer.valueOf(a);  
        System.out.println(b);  
    }  
    @Override  
    public void run() {  
        print(this.i);  
    }  
    public static void main(final String[] args) {  
        new RamTest(2).start();  
        new RamTest(3).start();  
    }  
}
```



Java Speichermodell 2/2

```
public class RamTest extends Thread {  
    private int i;  
    public RamTest(final int i) {  
        this.i = i;  
    }  
    public void print(final int i) {  
        final var a = i * i;  
        final var b = Integer.valueOf(a);  
        System.out.println(b);  
    }  
    @Override  
    public void run() {  
        print(this.i);  
    }  
    public static void main(final String[] args) {  
        new RamTest(2).start();  
        new RamTest(3).start();  
    }  
}
```



Zugriff auf Daten von Threads aus 1/2

Folie mit
Anmerkungen

```
public class Counter {  
    public int counter = 0;  
    public static void main(String[] args) {  
        Counter c = new Counter();  
        new Thread(() -> {  
            var a = c.counter;  
            a++;  
            c.counter = a;  
        }).start();  
        new Thread(() -> {  
            var b = c.counter;  
            b += 2;  
            c.counter = b;  
        }).start();  
        System.out.println("counter: " + c.counter);  
    }  
}
```

counter = 0

Thread A

```
a = c.counter;  
a++;  
c.counter = a;
```

Thread B

```
b = c.counter;  
b += 2;  
c.counter = b;
```

Zugriff auf Daten von Threads aus 2/2

Folie mit
Anmerkungen

gegenseitiger Ausschluss

kritischer Abschnitt
("region")

Thread A
`a = c.counter;`
`a++;`
`c.counter = a;`

Thread B
`b = c.counter;`
`b += 2;`
`c.counter = b;`

kritischer Abschnitt
("region")

counter = 0

Möglichkeit 1

`a = c.counter;`
`b = c.counter;`
`a++;`
`c.counter = a;`
`b += 2;`
`c.counter = b;`

counter = 2

Möglichkeit 2

`a = c.counter;`
`a++;`
`c.counter = a;`
`b = c.counter;`
`b += 2;`
`c.counter = b;`

counter = 3

Möglichkeit 3

`a = c.counter;`
`b = c.counter;`
`a++;`
`b += 2;`
`c.counter = b;`
`c.counter = a;`

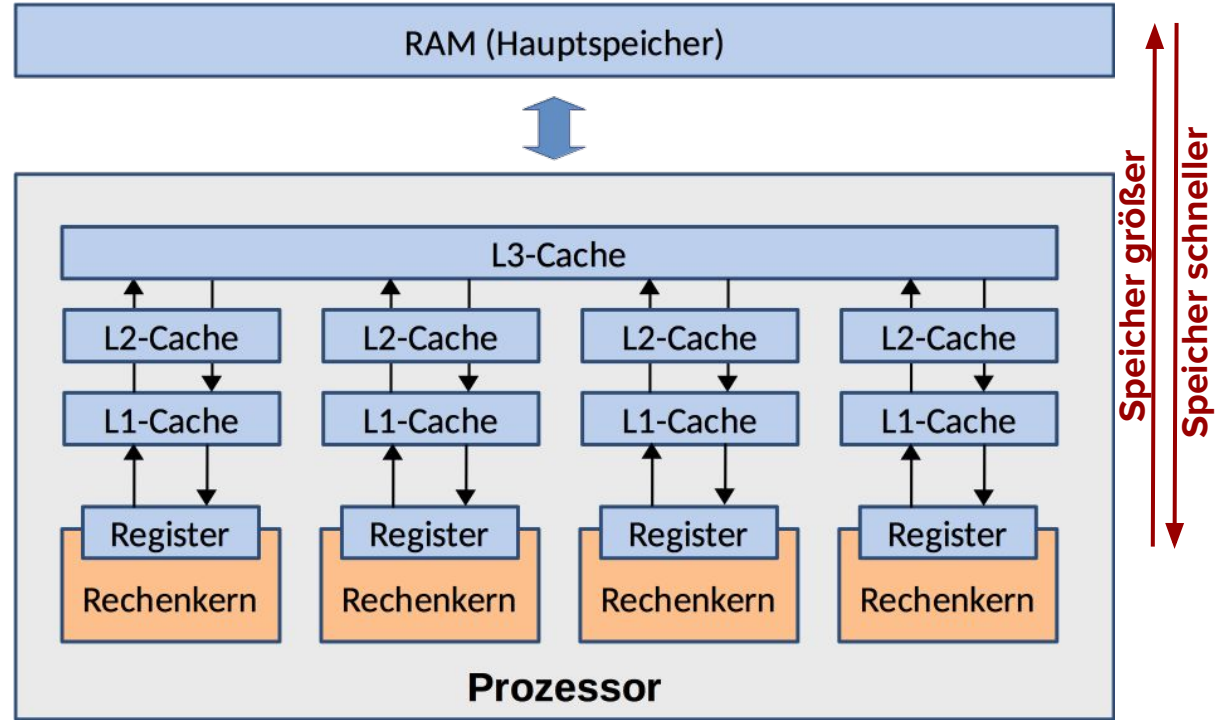
counter = 1

Hardware-Speichermodell und “sequentielle Konsistenz”

JVM arbeitet intern auf einem **hardwarenäheren Speichermodell**.

Zur **Optimierung** der Verschiebeoperationen von/nach Caches darf die **Reihenfolge** von Operationen geändert werden, wenn das keine inhaltliche Änderung bedeutet (außer eine Variable ist mit **volatile** gekennzeichnet).

Threads werden dabei aber nicht berücksichtigt.



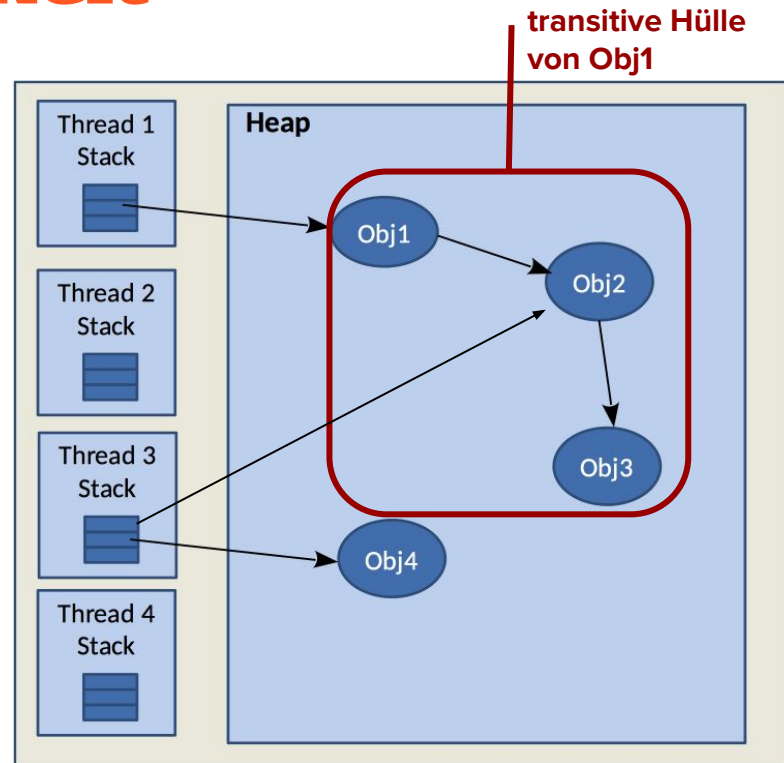
Besonderheit des Java Speichermodells bei Nebenläufigkeit

Folie mit
Anmerkungen

Threads greifen **immer** auf eine lokale Kopie des Heaps zu!

- enthält die *transitive Hülle* aller vom dem Stack referenzierten Objekte im Heap
- **lokale Heap-Kopien können untereinander abweichen**
- **Kommunikation zwischen Threads über Variablen so nicht möglich**
- **Ausnahme:** Schlüsselwort “**volatile**” für Variablen
 - solche Variablen verhalten sich so, als wären sie im “echten Heap” (und nicht in der gecachten Kopie)
- **Ausnahme **synchronized****
 - lokal gecachte Kopie des Heaps wird aktualisiert:
 - “pull” beim Eintritt in **synchronized**-Block
 - “push” beim Ende eines **synchronized**-Blocks

“memory barriers”



Sequentielle Konsistenz durch *memory barrier* 1/2

Folie mit
Anmerkungen

Folgende Fälle bilden *memory barriers*, an denen die Cache-lokalen Änderungen zwischen Threads ausgetauscht werden:

- **Zugriff auf `volatile`-Variablen**

Das Schreiben einer `volatile`-Variablen bewirkt die Synchronisierung mit allen Threads, die zu einem späteren Zeitpunkt die Variable lesen.

- **Lock-Objekte bzw. `synchronized`**

Eine Lock-Freigabe synchronisiert alle durchgeführten Änderungen mit allen Threads, die danach den Lock erwerben.

- **Starten eines Threads**

Alle Aktionen vor dem Starten finden vor der ersten Operation des neu gestarteten Threads statt.

Sequentielle Konsistenz durch *memory barrier* 2/2

Folgende Fälle bilden **memory barriers**, an denen die Cache-lokalen Änderungen zwischen Threads ausgetauscht werden:

- Die **Initialisierung** mit Defaultwerten (0, false oder null) aller Variablen sorgt für die Synchronisierung mit dem ersten Zugriff.
- Das **Ende eines Threads** bewirkt die Synchronisierung mit jeder Aktion eines auf dessen Ende wartenden Threads. Wenn z.B. ein `join()`-Aufruf zurückkehrt, sieht der Aufrufer alle von dem Thread gemachten Änderungen.
- Wenn Thread T1 Thread T2 unterbricht, wird garantiert, dass alle Threads die **Unterbrechung** sehen. Ein Aufruf von `isInterrupted()` liefert immer den aktuellen Unterbrechungsstatus.

Laboraufgabe (15 Minuten)

Memory Barriers

pp.02.01-MemoryBarrier

Thread-lokale Daten

`java.lang.ThreadLocal<T>`

Besonderheit: Thread-lokaler Speicher (TLS)

- TLS mit Heap für Objekte, die nur ein bestimmter Thread “sehen” kann.
- nur für Objekte vom Typ T, wenn über `ThreadLocal<T>`-Objekt erzeugt
=> hier ist T Integer
- sehr viel schneller als Heap, braucht nicht synchronisiert werden

```
public class ThreadLocalDemo {  
    public static class Runner implements Runnable {  
        public static ThreadLocal<Integer> mem =  
            new ThreadLocal<>() {  
                @Override  
                protected Integer initialValue() {  
                    return Integer.valueOf(1);  
                }  
            };  
        @Override  
        public void run() {  
            while (true) {  
                mem.set(mem.get() + 1);  
            }  
        }  
    }  
    public static void main(final String[] args) {  
        final var runnable = new Runner();  
        new Thread(runnable, "Runner-1").start();  
        new Thread(runnable, "Runner-2").start();  
    }  
}
```

Thread-lokaler Zufallszahlengenerator

Man kann einen Thread-lokalen Zufallszahlengenerator abrufen. Dieser ist nicht Thread-sicher implementiert und damit effizienter als der “normale” Zufallszahlengenerator `java.util.Random`, der Thread-sicher implementiert ist. `java.util.concurrent.ThreadLocalRandom` hat aber dieselbe Signatur wie `java.util.Random`:

```
import java.util.Random;  
import java.util.concurrent.ThreadLocalRandom;
```

```
Random r = ThreadLocalRandom.current();
```

Laboraufgabe (15 Minuten)

Thread-Local Storage

pp.02.02-ThreadLocal

Gegenseitiger Ausschluss ("Mutex")

synchronized für kritische Regionen

```
public class SynchAccess {  
    private int counter = 0;  
    public synchronized void doubler() {  
        this.counter = this.counter * 2; <= äquivalent =>  
    }  
}
```

```
public class SynchAccess {  
    private int counter = 0;  
    public void doubler() {  
        synchronized (this) {  
            this.counter = this.counter * 2;  
        }  
    }  
}
```

```
public static void main(final String[] args) {  
    var counter = new SynchAccess();  
    (new Thread(() -> {  
        while (true) {  
            counter.doubler();  
        }  
    }, "Doubler")).start();  
    // ...  
}
```

```
public static void main(final String[] args) {  
    var counter = new SynchAccess();  
    (new Thread(() -> {  
        while (true) {  
            counter.doubler();  
        }  
    }, "Doubler")).start();  
    // ...  
}
```

} gegenseitiger Ausschluss durch Verwendung einer Lock-Variablen (hier: this bzw. SynchAccess.class falls doubler() static wäre),

Eintritt/Austritt in/aus synchronized-Block fungiert gleichzeitig als Memory Barrier (wichtig für Getter)

synchronized-Block mit anderem Objekt

```
public class SynchAccess {  
    private int counter = 0;  
    private Object lock = new Object();  
    public void doubler() {  
        synchronized (lock) {  
            this.counter = this.counter * 2;  
        }  
    }  
    public static void main(final String[] args) {  
        final var counter = new SynchAccess3();  
        (new Thread(() -> {  
            while (true) {  
                counter.doubler();  
            }  
        }, "Doubler")).start();  
    }  
}
```

Schlossvariable kann ein Objekt einer beliebigen Klasse sein, z.B. `Object`. Muss von allen Threads “gesehen” werden können.

Laboraufgabe (15 Minuten)

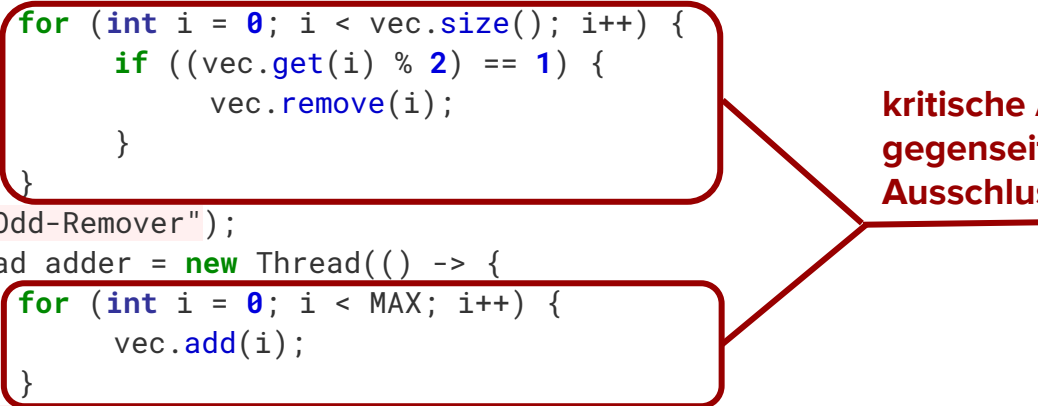
gegenseitiger Ausschluss

pp.02.03-Lock

Thread-Sicherheit von Vector

Alle Methoden in Vector sind synchronized. Allerdings können sich Iterationen überlappen:

```
public class SynchVector {  
    private static Vector<Integer> vec = new Vector<>();  
    public static void main(final String[] args) throws InterruptedException {  
        //...  
        Thread remover = new Thread(() -> {  
            for (int i = 0; i < vec.size(); i++) {  
                if ((vec.get(i) % 2) == 1) {  
                    vec.remove(i);  
                }  
            }  
        }, "Odd-Remover");  
        Thread adder = new Thread(() -> {  
            for (int i = 0; i < MAX; i++) {  
                vec.add(i);  
            }  
        }, "Odd-Adder");  
        remover.start(); adder.start();  
        remover.join(); adder.join();  
    }  
}
```

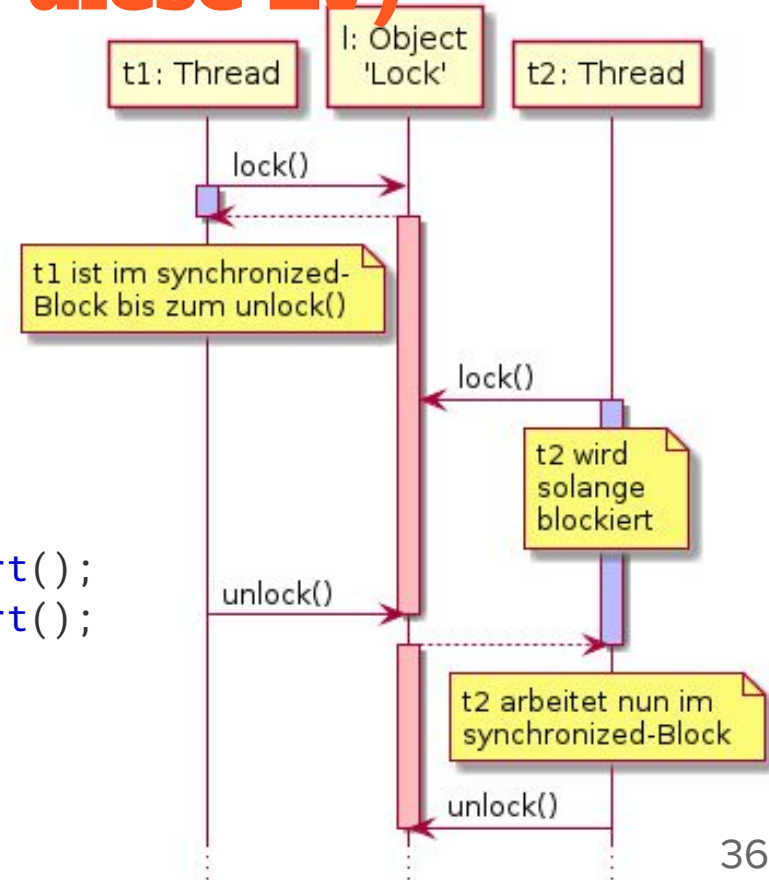


kritische Abschnitte mit gegenseitigem Ausschluss

synchronized im Sequenzdiagramm (Notation für diese LV)

Folie mit
Anmerkungen

```
public class Sync {  
    private static Object l = new Object();  
  
    public static void m1() {  
        synchronized (l) {  
            /* ... */  
        }  
    }  
  
    public static void main(String...args) {  
        var t1 = new Thread(Sync:m1); t1.start();  
        var t2 = new Thread(Sync:m1); t2.start();  
    }  
}
```



Deadlock durch gegenseitiges Warten

Folie mit
Anmerkungen

```
public class Deadlock {  
    private static Object l1 = new Object();  
    private static Object l2 = new Object();  
  
    public static void m1() {  
        synchronized (l1) {  
            synchronized (l2) {  
                /* ... */  
            }  
        }  
    }  
  
    public static void m2() {  
        synchronized (l2) {  
            synchronized (l1) {  
                /* ... */  
            }  
        }  
    }  
  
    public static void main(final String[] args) {  
        var t1 = new Thread(Deadlock::m1); t1.start();  
        var t2 = new Thread(Deadlock::m2); t2.start();  
    }  
}
```

