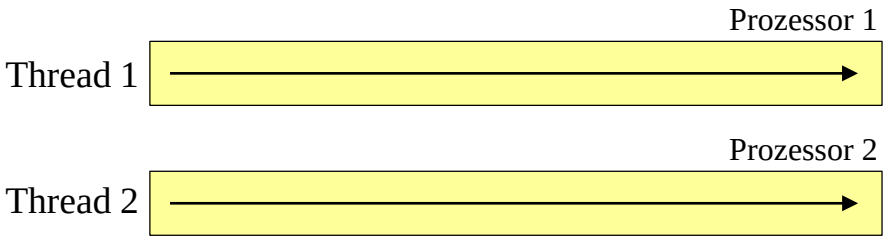
18. Threads

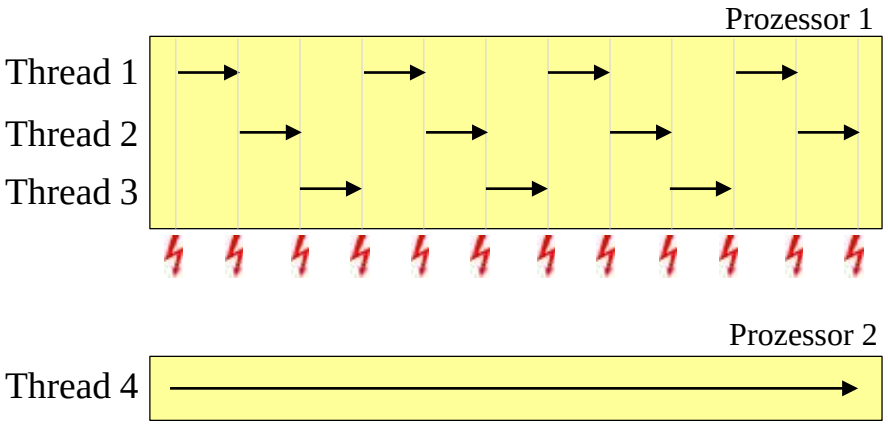
Parallelität und Quasiparallelität

Echte Parallelität



Thread ... ablaufendes Programm

Quasiparallelität



Betriebssystem (*Scheduler*) schaltet in kurzen Abständen zwischen den Threads um.

Alle Threads laufen "gleichzeitig"
-- nur langsamer und verzahnt

Wozu Parallelität?

- Verteilung komplexer Rechenaufgaben auf mehrere Threads (nur bei echter Parallelität sinnvoll)
- In GUIs: Ein Thread wartet auf Benutzereingaben, andere Threads führen sie aus
- ...

Deklaration von Threads

Bibliotheksklasse *Thread*

```
class Thread {
    void start() {...}
    void run() {...}
    static void sleep(long milliSec) {...}
    ...
}
```

startet den Thread
Anweisungen, die parallel zu anderen Threads laufen sollen
pausiert den Thread für *milliSec* Millisekunden

Eigene Threads sind Unterklassen davon (müssen *run()* überschreiben)

```
class CharPrinter extends Thread {
    char ch;
    int delay;

    CharPrinter (char ch, int delay) {
        this.ch = ch; this.delay = delay;
    }

    public void run() {
        for (int i = 0; i < 20; i++) {
            Out.print(ch);
            try { Thread.sleep(delay); }
            catch (InterruptedException e) { return; }
        }
    }
}
```

Anweisungen, die parallel zu anderen Threads ausgeführt werden sollen

Am Ende von *run()* "stirbt" der Thread

Erzeugen von Threads


```
class Sample {
   public static void main(String[] arg) {
      CharPrinter thread1 = new CharPrinter('.', 10);
      CharPrinter thread2 = new CharPrinter('*', 30);

      thread1.start();
      thread2.start();




      Out.print('+');
   }
}
```

erzeugt 2 *CharPrinter*-Threads,
startet sie aber noch nicht

startet die beiden Threads
=> bewirkt Aufruf ihrer *run()*-Methode

jetzt laufen 3 Threads (quasi)parallel
- *thread1*
- *thread2*
- *main*

main stirbt, aber *thread1* und *thread2* laufen noch

Programm wird beendet, wenn alle Threads gestorben sind

Ausgabe

```
+*...*...*...*...*...*...*..**********
```

Selbständig startende Threads


```
class CharPrinter extends Thread {
    ...
    CharPrinter(char ch, int delay) {
        ...
        start();
    }
    public void run() {
        ...
    }
}
```

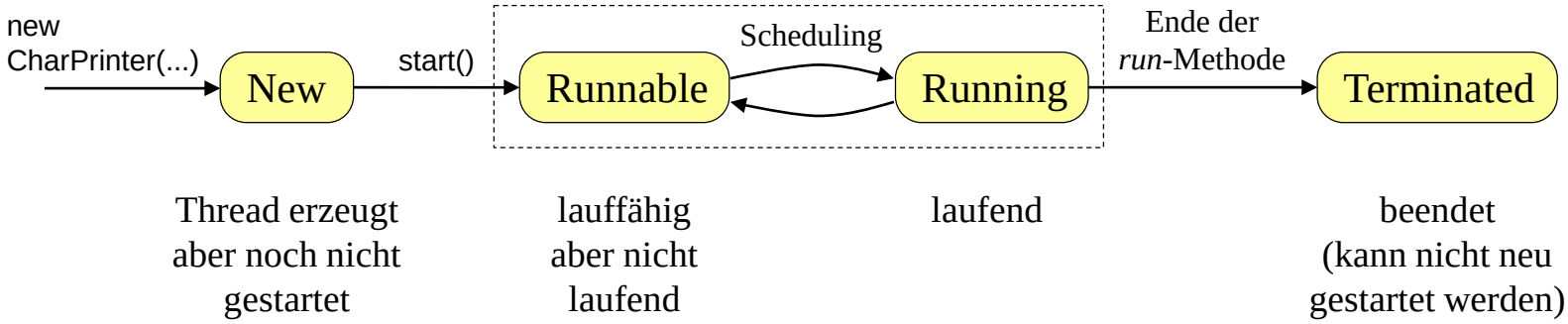
Benutzung

```
Thread thread = new CharPrinter('.', 10);
```

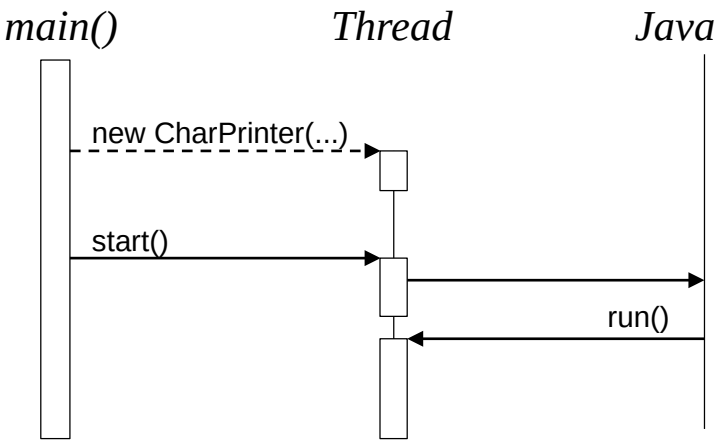
startet von selbst

Normalerweise starten Threads aber nicht selbständig

Thread-Zustände (vereinfacht)



Was geschieht beim Starten eines Threads?



run() wird von der Java VM aufgerufen

- Thread läuft noch nicht gleich nach *start()*
- Eigentlich erst nach Aufruf von *run()* in *Runnable*

18. Threads

Interface Runnable

Beschreibt etwas, das als Thread gestartet werden kann

```
interface Runnable {
    void run();
}
```

Thread-Konstruktoren

```
class Thread {
    Thread() {...}
    Thread(Runnable r) {...}
    ...
}
```

Ermöglicht Threads, die nicht von der Klasse *Thread* abgeleitet sind

```
class CharPrinter2  extends ...  implements Runnable {
    ...
    public void run () {
        ...
    }
}
```

kann von etwas anderem als *Thread* abgeleitet sein

+ Vererbungsslot kann für andere Zwecke verwendet werden
- CharPrinter2 kann Thread-Methoden nicht aufrufen

Erzeugung

```
...
Thread thread = new Thread(new CharPrinter2('.', 10));
thread.start();
```

übergibt ein *Runnable*-Objekt

Threads als anonyme Klassen


```
Thread thread = new Thread() {
    public void run() {
        ...
    }
};
thread.start();
```

```
Thread thread = new Thread(new Runnable() {
    public void run() {
        ...
    }
});
thread.start();
```

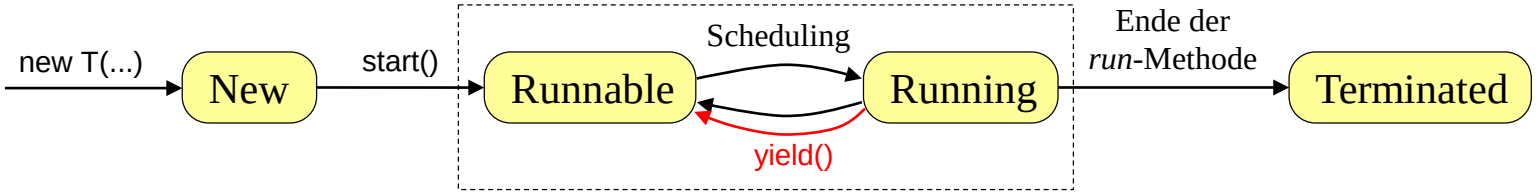
\+ Man muss keine benannte Klasse deklarieren

\- Man kann den Thread nicht über einen Konstruktor initialisieren

18. Threads

Thread.yield()

Freiwillige Abgabe des Prozessors



Normalerweise unnötig, weil Scheduler automatisch zwischen Threads umschaltet

Mögliche Anwendung: **Faires Busy Waiting**

Busy Waiting

```
while (In.available() == 0) ;
```

Thread wartet, ohne die Kontrolle abzugeben (Scheduler schaltet alle *x* ms um)

Faires Busy Waiting

```
while (In.available() == 0) Thread.yield();
```

Thread gibt Kontrolle sofort ab, ohne den Prozessor zu blockieren

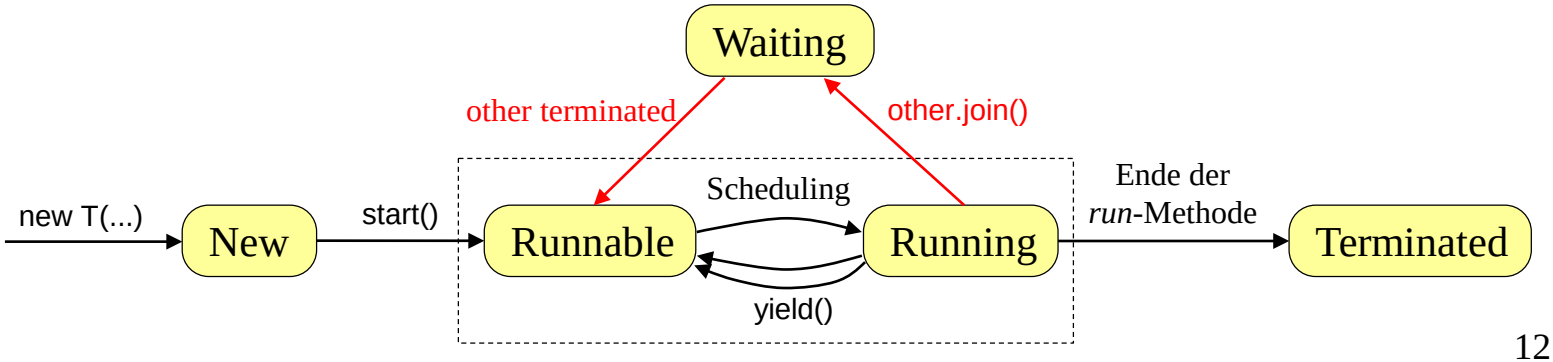
Thread-Ende

Man kann/darf Threads nicht explizit abbrechen (könnte zu inkonsistentem Zustand führen)

t.join() wartet auf das natürliche Ende des Threads *t*

```
CharPrinter thread1 = new CharPrinter('.', 10);
CharPrinter thread2 = new CharPrinter('*', 30);
thread1.start(); thread2.start();
try {
   thread1.join();  // wartet, bis thread1 fertis ist
   thread2.join();  // wartet, bis thread2 fertig ist
} catch (InterruptedException e) {
}
Out.println();
Out.println("thread1 and thread2 finished");
```

```
.*...*..*...*...*...*...*..*************
thread1 and thread2 finished
```



Abfragen des Thread-Endes

boolean alive = t.isAlive() *true*, wenn Thread *t* nicht im Zustand *Terminated* ist

```
CharPrinter thread1 = new CharPrinter('.', 10);
CharPrinter thread2 = new CharPrinter('*', 30);
thread1.start();
thread2.start();

long beg1 = System.currentTimeMillis();
long beg2 = System.currentTimeMillis();
long end1 = 0;
long end2 = 0;
while (end1 == 0 || end2 == 0) {
   if (end1 == 0 && !thread1.isAlive()) end1 = System.currentTimeMillis();
   if (end2 == 0 && !thread2.isAlive()) end2 = System.currentTimeMillis();
}
Out.println();
Out.println("thread1 took " + (end1 - beg1) + " ms");
Out.println("thread2 took " + (end2 - beg2) + " ms");
```

```
.*..*...*...*...*...*...*..************
thread1 took 212 ms
thread2 took 608 ms
```

Daemon-Threads

Werden abgebrochen, wenn alle "normalen" Threads zu Ende sind

t.setDaemon(true); Macht Thread *t* zu einem Daemon-Thread

if (t.isDaemon()) ... Liefert *true*, wenn Thread *t* ein Daemon-Thread ist

```
CharPrinter thread1 = new CharPrinter('.', 10);
CharPrinter thread2 = new CharPrinter('*', 30);
thread1.setDaemon(true);
thread2.setDaemon(true);
if (thread1.isDaemon()) Out.println("thread1 is a daemon thread");
if (thread2.isDaemon()) Out.println("thread2 is a daemon thread");
if (Thread.currentThread().isDaemon()) Out.println("main is a daemon thread");
thread1.start();
thread2.start();
try {
   Thread.sleep(100);
} catch (InterruptedException e) {
}
Out.println();
Out.println("end of main");
```

```
thread1 is a daemon thread
thread2 is a daemon thread
.*..*...*...*.
end of main
```

Thread-Interrupts

Interrupt ist hier eigentlich ein falsches Wort

Eher: **Signal von einem Thread an einen anderen**
Setzt dort ein Interrupt-Flag (wird meist dazu verwendet, den Thread abzubrechen)

t.interrupt();	Setzt im Thread *t* das Interrupt-Flag
if (t.isInterrupted()) ...	Liefert *true*, wenn im Thread *t* das Interrupt-Flag gesetzt ist

```
class EndlessPrinter extends Thread {
   public void run() {
      while (!isInterrupted()) {
         Out.print('.');
      }
      Out.println("EndlessPrinter interrupted");
   }
}
```

```
class InterruptSample {
   public static void main(String[] arg) {
      Thread thread = new EndlessPrinter();
      thread.start();
      String line = In.readLine();
      thread.interrupt();
      for (int i = 1; i < 10000000; i++);
      Out.println("main terminated");
   }
}
```

```
.....................................................
.................. (usw.) ..........................
..EndlessPrinter interrupted
main terminated
```

InterruptedException

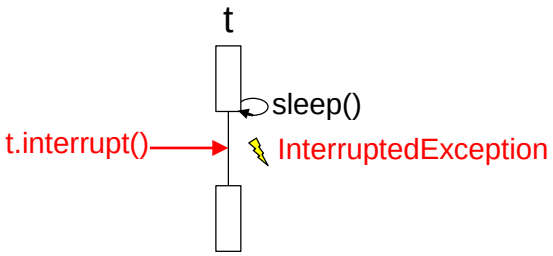
t.interrupt() während Thread *t* in *sleep()* oder *join()* wartet
löst automatisch *InterruptedException* aus (bricht das Warten ab)

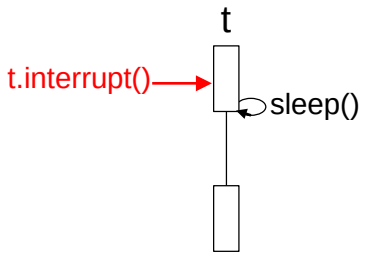
```
class CharPrinter extends Thread {

   ...
   public void run() {
      for (int i = 0; i < 20 && !isInterrupted(); i++) {
         Out.print(ch);
         try {
            Thread.sleep(delay);
         } catch (InterruptedException e) {
            return;
         }
      }
   }
}
```

warum ist diese Abfrage nötig?

t.interrupt() löst *InterruptedException* aus

Warum ist Abfrage auf *interrupted()* nötig?





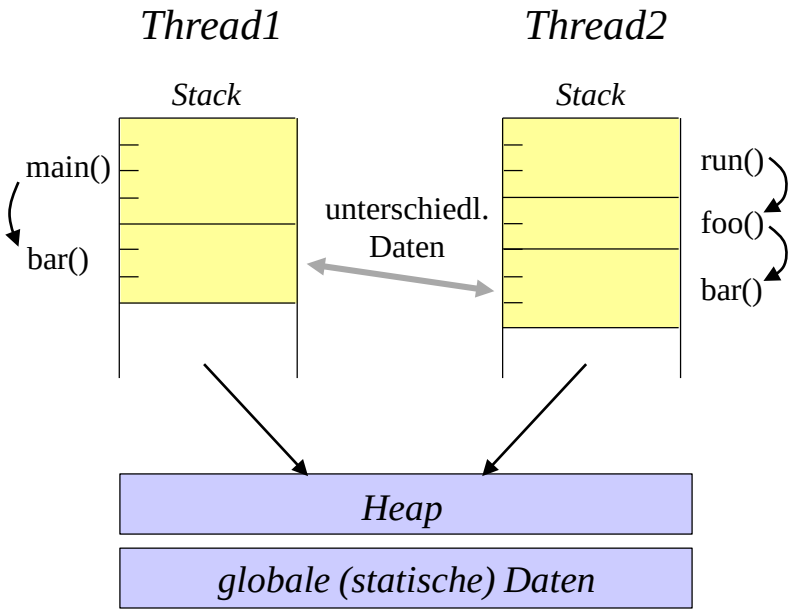
würde keine Exception auslösen

Klasse Thread (Zusammenfassung)


```
class Thread {
   Thread() {...}                                                    Konstruktoren
   Thread(Runnable r) {...}

   static Thread currentThread() {...}                               liefert den gerade laufenden Thread
   static void   sleep(long ms) throws InterruptedException {...}    pausiert den Thread für ms Millisek.
   static void   yield() {...}                                       gibt Kontrolle an anderen Thread ab

   long          getID() {...}                                       liefert von VM vergebenen ID
   Thread.State  getState() {...}                                    liefert aktuellen Thread-Zustand

   void          setName(String name) {...}                          man kann Threads einen Namen geben
   String        getName() {...}

   void          setPriority(int prio) {...}                         man kann Threads eine Priorität
   int           getPriority() {...}                                 geben (1 = niedrigste, 10 = höchste)

   void          setDaemon(boolean on) {...}                         Daemon-Threads werden am
   boolean       isDaemon() {...}                                    Programmende abgebrochen

   boolean       isAlive() {...}                                     true, wenn nicht im Zustand Terminated

   void          interrupt() {...}                                   setzt Interrupt-Flag
   boolean       isInterrupted() {...}                               prüft Interrupt-Flag

   void          run() {...}                                         eigentliche Anweisungen des Threads
   void          start() {...}                                       startet den Thread
   void          join() throws InterruptedException {...}            Rufer wartet, bis Thread beendet
   ...
}
```

18. Threads

Datenaustausch zwischen Threads



Jeder Thread hat seine eigenen lokalen Variablen

Alle Threads greifen auf dieselben globalen Daten und Heap-Objekte zu

⇨ Datenaustausch zwischen Threads nur über globale Daten und den Heap

⇨ Gemeinsamer Datenzugriff erfordert Synchronisation!

Beispiel: gemeinsam benutztes Objekt

Klasse *Account*

```
public class Account {
   private int balance = 0;
   public void deposit(int x) {
      balance = balance + x;
   }
   public void withdraw(int x) {
      balance = balance - x;
   }
   ...
}
```

Thread, der mit *Account* arbeitet

```
class Worker extends Thread {
   Account account;
   Worker(Account account) {
      this.account = account;
   }
   public void run() {
      ... account.deposit(...); ...
      ... account.withdraw(...); ...
   }
}
```

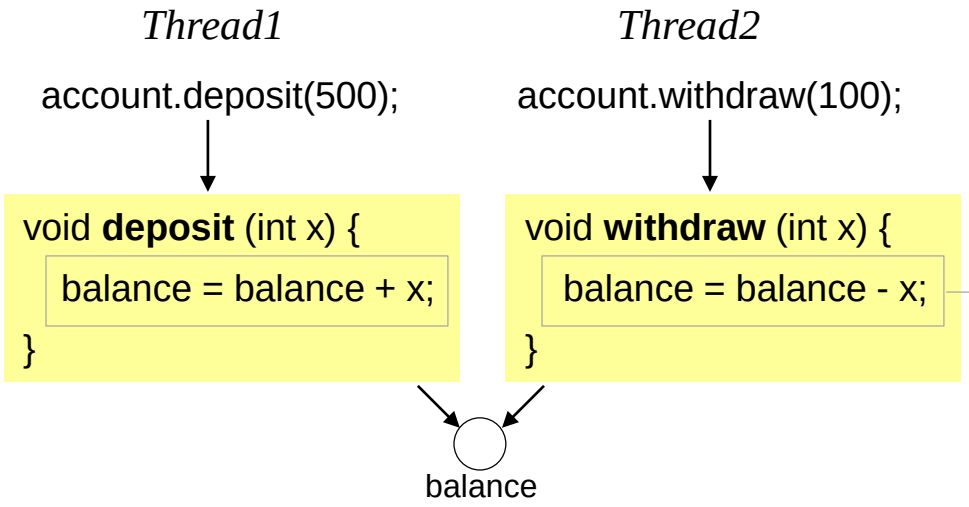
Benutzung

```
Account account = new Account();
...
Thread thread1 = new Worker(account);
Thread thread2 = new Worker(account);
thread1.start();
thread2.start();
...
```

Beide Threads arbeiten mit demselben *Account*-Objekt

Race Conditions

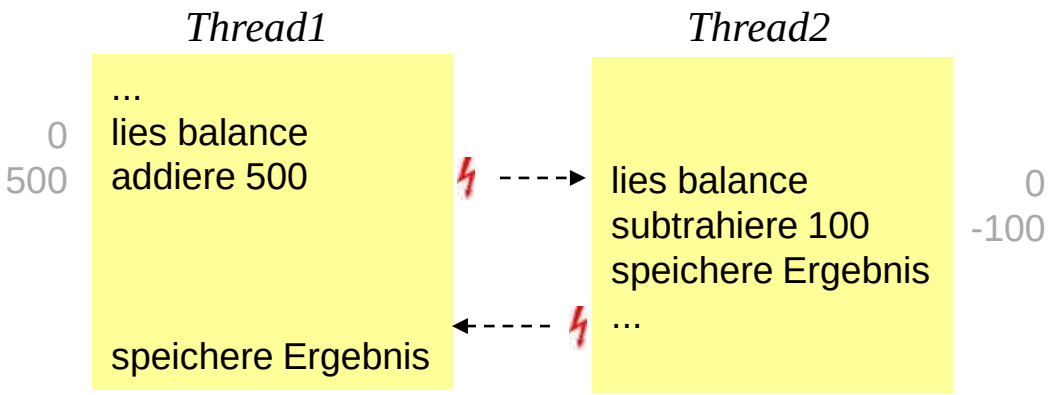
Gefahr von **Inkonsistenz** beim **Zugriff mehrerer Threads auf gemeinsame Daten**



Critical Regions, die nicht überlappend ausgeführt werden dürfen

Prinzip der Mutual Exclusion
Es darf immer nur 1 Thread gleichzeitig in einer der Critical Regions sein.

Was kann passieren? Annahme: *balance* == 0



Race Condition
Schwer zu findende Fehler
Kaum reproduzierbar!

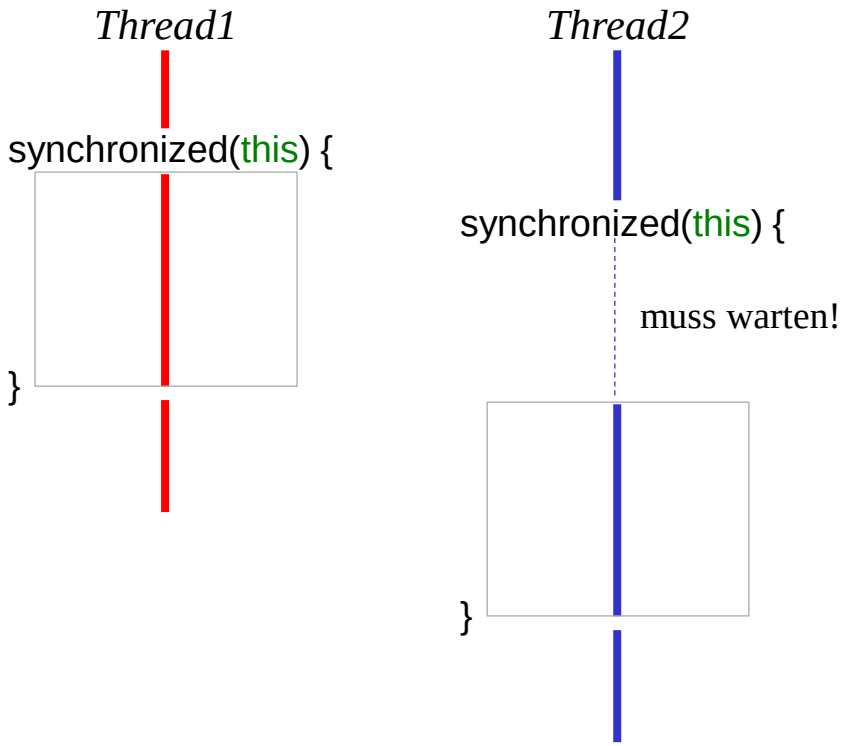
balance == 500 => Subtraktion ging verloren

Mutual Exclusion durch synchronized

```
class Account {
   int balance = 0;
   void deposit (int x) {
      synchronized(this) {
         balance = balance + x;
      }
   }
   void withdraw (int x) {
      synchronized(this) {
         balance = balance - x;
      }
   }
}
```

critical regions

Ablauf



Allgemein

```
synchronized (object) { ... }
```

- Thread fordert Sperre (*lock*) auf *object* an
- Thread blockiert, bis er Sperre bekommt
- Thread gibt Sperre am Ende d. Anw. wieder frei

Garantiert, dass Critical Regions nie überlappen
=> garantiert Mutual Exclusion
=> keine Race Conditions

Jedes Objekt hat seine eigene Sperre

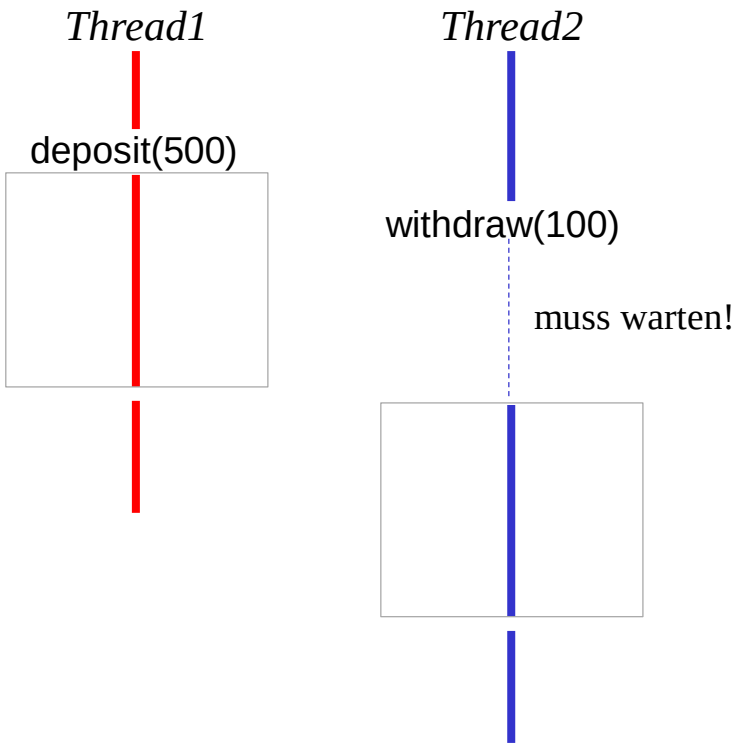
synchronized-Methoden

Man kann auch ganze Methoden als *synchronized* deklarieren

```
class Account {
   int balance = 0;
   synchronized void deposit (int x) {
      balance = balance + x;
      ...
   }
   synchronized void withdraw (int x) {
      balance = balance - x;
      ...
   }
}
```

critical regions

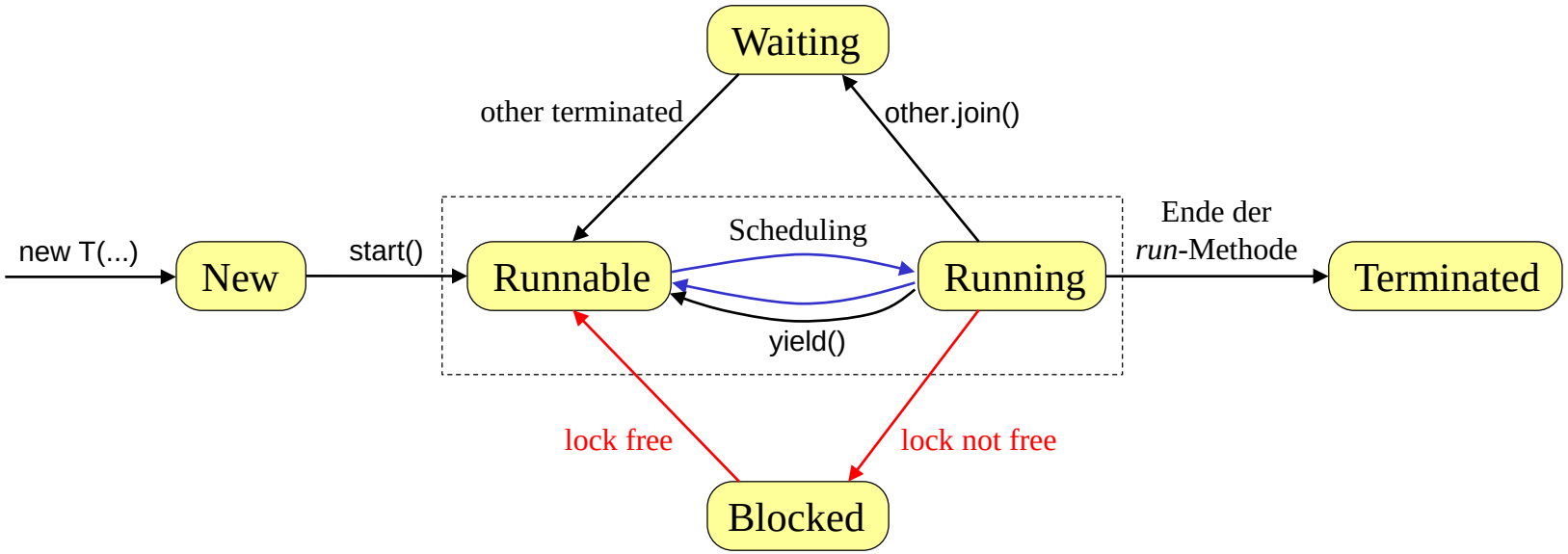
Ablauf



Die Sperre wird hier auf *this* gesetzt

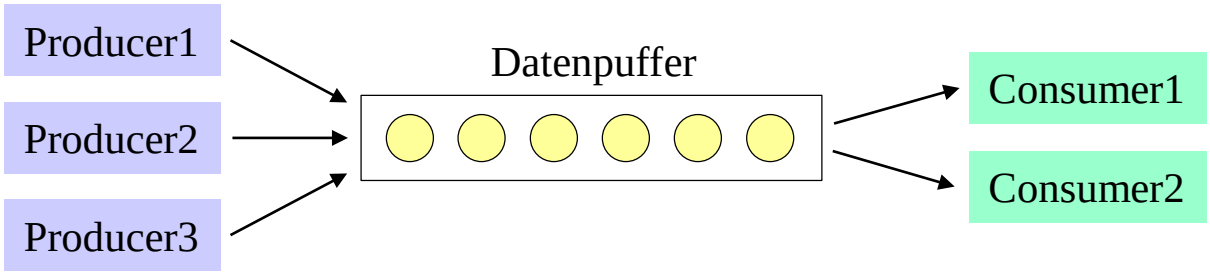
Eine Klasse, die eine Datenstruktur mittels *synchronized*-Methoden verwaltet, nennt man einen **Monitor**

Thread-Zustände

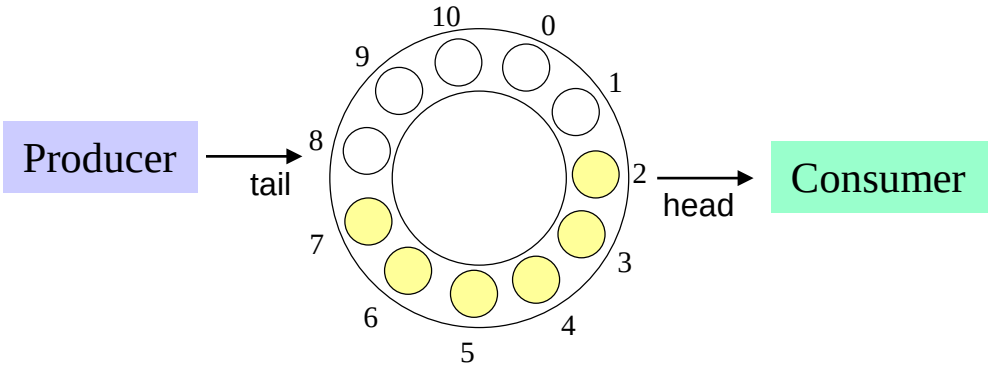


Producer-Consumer-Schema

1..n Threads produzieren Daten und legen sie in einem Puffer ab
1..m Threads konsumieren Daten aus dem Puffer
=> zeitliche Entkopplung der Producers und Consumers



Puffer wird oft als Ringpuffer verwaltet



Was ist, wenn Puffer voll ist: Producer muss warten
Was ist, wenn Puffer leer ist: Consumer muss warten

```
class Buffer {
    Data[] buffer = new Data[N];
    int head = tail = 0;

    void add(Data data) {
        buffer[tail] = data;
        tail = (tail+1) % N;
    }

    Data remove() {
        Data data = buffer[head];
        head = (head+1) % N;
        return data;
    }
}
```

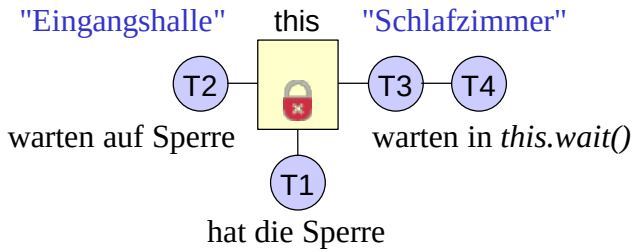
wait(), notify(), notifyAll()

Threads müssen manchmal mitten in der Critical Region auf eine Bedingung warten

```
synchronized(this) {
    ...
    if (!someCondition) {
        gib Sperre frei;
        gehe schlafen;
    }
    ...
}
```

while

this.wait();

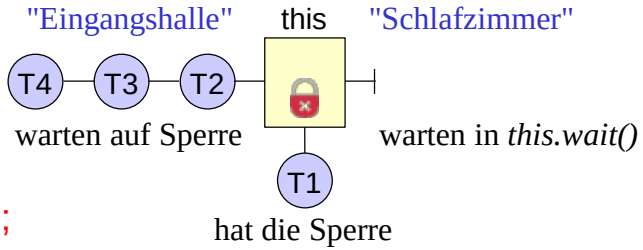


- wenn geweckt, muss sich Thread wieder um Sperre bewerben
- wenn noch immer !*someCondition* => nachmals *this.wait();*

Andere Threads können die Bedingung herstellen

```
synchronized(this) {
    ... // change
    ... // someCondition
    signalisiere, dass
    someCondition ge-
    ändert wurde
}
```

this.notifyAll();



- weckt alle schlafenden Threads
- verschiebt sie in die "Eingangshalle"

this.notify();

- weckt nur einen der schlafenden Threads

wait(), notify(), notifyAll()

Methoden der Klasse *Object* => werden an alle Klassen vererbt

Objekt, auf das der laufende Thread eine Sperre hält (meist *this*)

obj.wait();
- rufender Thread "geht schlafen" (reiht sich in Warteschlange von *obj* ein)
- ... gibt Sperre frei
- ... gibt Kontrolle an anderen Thread ab

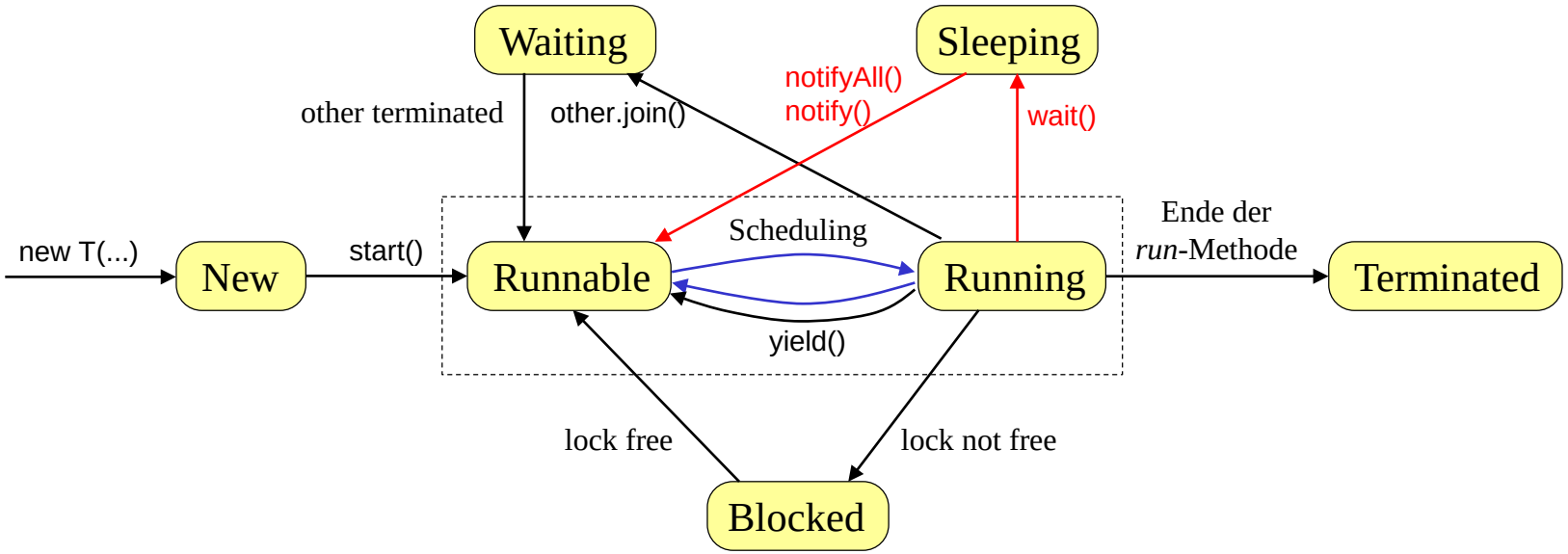
obj.notify();
- weckt einen der Threads aus der Warteschlange von *obj* auf
- dieser muss aber erst auf Freiwerden der Sperre warten

obj.notifyAll();
- weckt alle Threads aus der Warteschlange von *obj* auf
- diese müssen sich aber wieder um die Sperre bewerben

Dürfen nur in *synchronized*-Anweisung/Methode aufgerufen werden.

wait() kann eine *InterruptedException* werfen, die eigentlich abgefangen werden muss (wird in folgenden Beispielen der Einfachheit weggelassen)

Thread-Zustände



Beispiel eines Ablaufs (1)

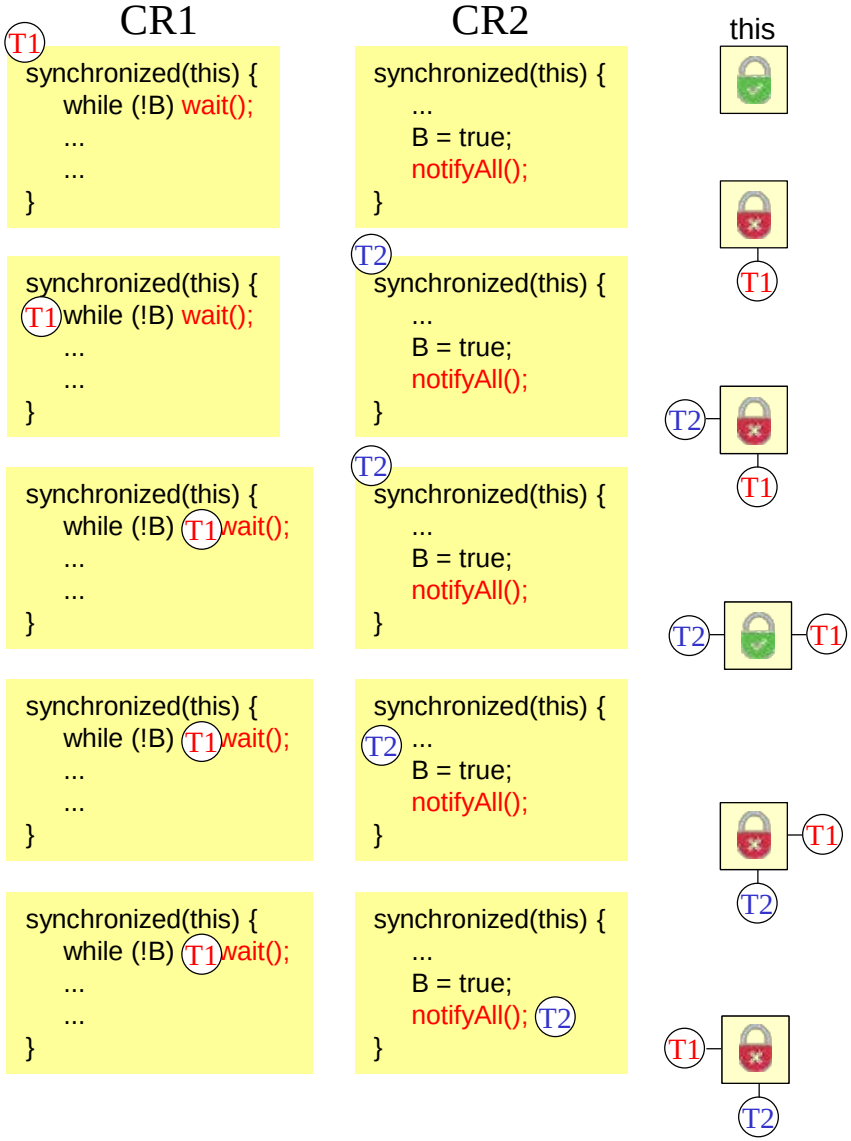
T1 kommt zu CR1
Sperre frei => betritt CR1

T2 kommt zu CR2
Sperre blockiert => warten

T1 findet *!B* und macht *wait()*
=> gibt Sperre frei

T2 findet Sperre frei
=> betritt CR2

T2 stellt *B* her und macht *notifyAll()*
=> weckt T1



Beispiel eines Ablaufs (2)

T2 verlässt CR2
=> Sperre frei

T1 findet Sperre frei
=> setzt nach *wait()* fort

T1 prüft nochmals *B* (== *true*)
macht weiter, verlässt CR1
=> Sperre frei

CR1	CR2

synchronized(this) {
while (!B) T1 wait();
...
...
}
	synchronized(this) {
	...
	B = true;
	notifyAll();
	} T2

synchronized(this) {
while (!B) wait(); T1
...
...
}
	synchronized(this) {
	...
	B = true;
	notifyAll();
	}

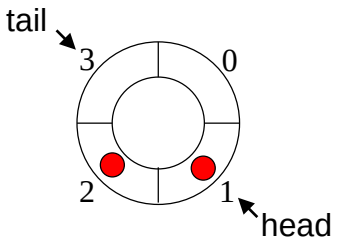
synchronized(this) {
while (!B) wait();
...
...
} T1
	synchronized(this) {
	...
	B = true;
	notifyAll();
	}

Beispiel: Synchronisierter Puffer

Producer-Consumer-Schema

```
class Buffer {
    static final int SIZE = 4;
    char[] buf = new char[SIZE];
    int head = 0, tail = 0, n = 0;

    public synchronized void put(char ch) {
        while (n == SIZE) wait();
        buf[tail] = ch;
        tail = (tail + 1) % SIZE;
        n++;
        notifyAll();
    }

    public synchronized char get() {
        while (n == 0) wait();
        char ch = buf[head];
        head = (head + 1) % SIZE;
        n--;
        notifyAll();
        return ch;
    }
}
```

Zyklischer Puffer der Länge 4



Wenn der *Producer* schneller ist

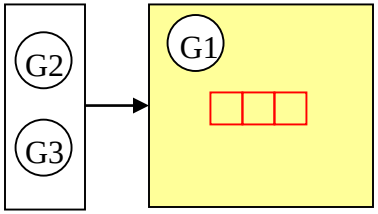
put
put
put
put
get
put
get
...

Wenn der *Consumer* schneller ist

put
get
put
get
...

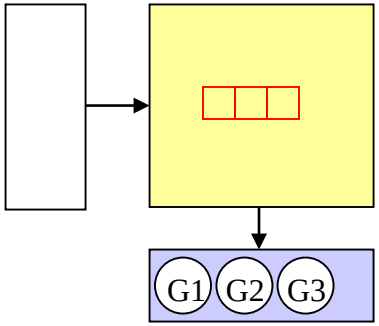
Visualisierung d. Puffersynchronisation

3 *get*-Threads kommen;
Puffer leer



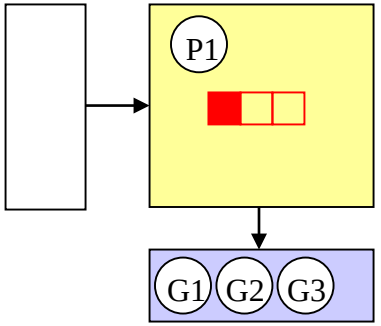
Eingangs-halle *Critical Region*

sie betreten nacheinander
die CR und gehen "schlafen"
weil Puffer leer

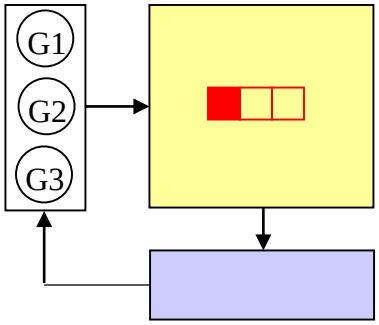


Schlafzimmer

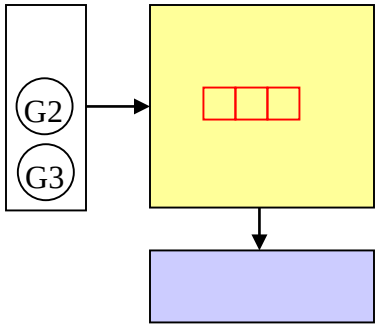
ein *put*-Thread kommt,
betritt CR, legt seine Daten ab
und ruft *notifyAll()*



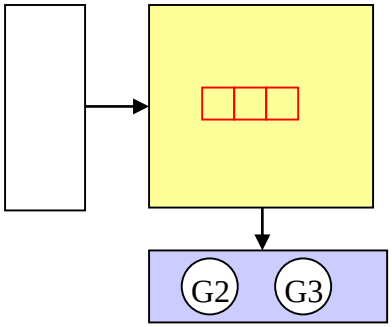
alle *get*-Threads wachen auf;
der erste betritt CR, holt Daten
aus dem Puffer und geht;



die anderen betreten ebenfalls CR,
finden aber den Puffer leer ...



... so dass sie wieder
schlafen gehen



18. Threads

Deadlock (Systemverklemmung)

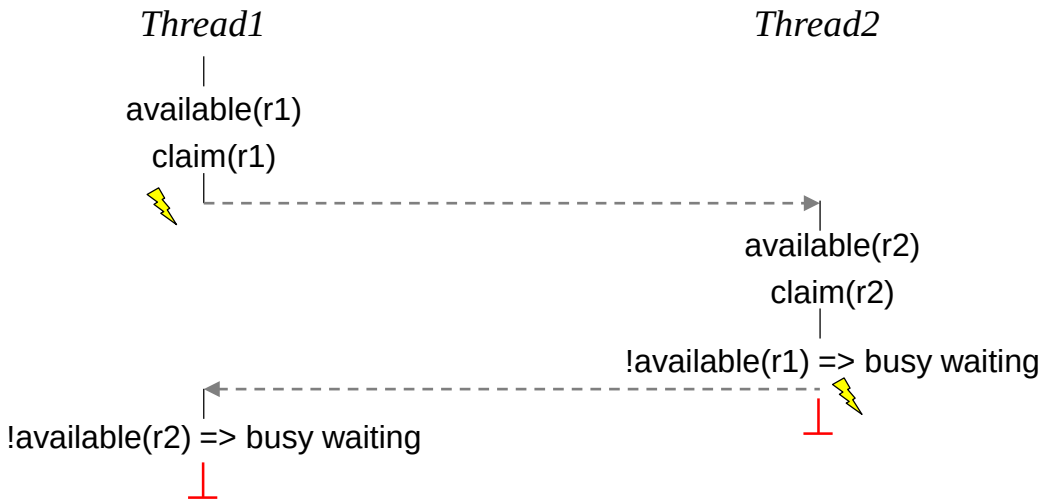
2 oder mehr Threads warten auf Bedingungen,
die nur andere wartende Threads herstellen können => alles steht

Beispiel mit Busy Waiting

```
while (!available(r1)) ;
claim(r1);
while (!available(r2)) ;
claim(r2);
... use r1 and r2 ...
free(r1);
free(r2);
```

```
while (!available(r2)) ;
claim(r2);
while (!available(r1)) ;
claim(r1);
... use r1 and r2 ...
free(r1);
free(r2);
```

r1, r2 ... Ressourcen



Deadlock!

Synchronisation hilft meist

CR1

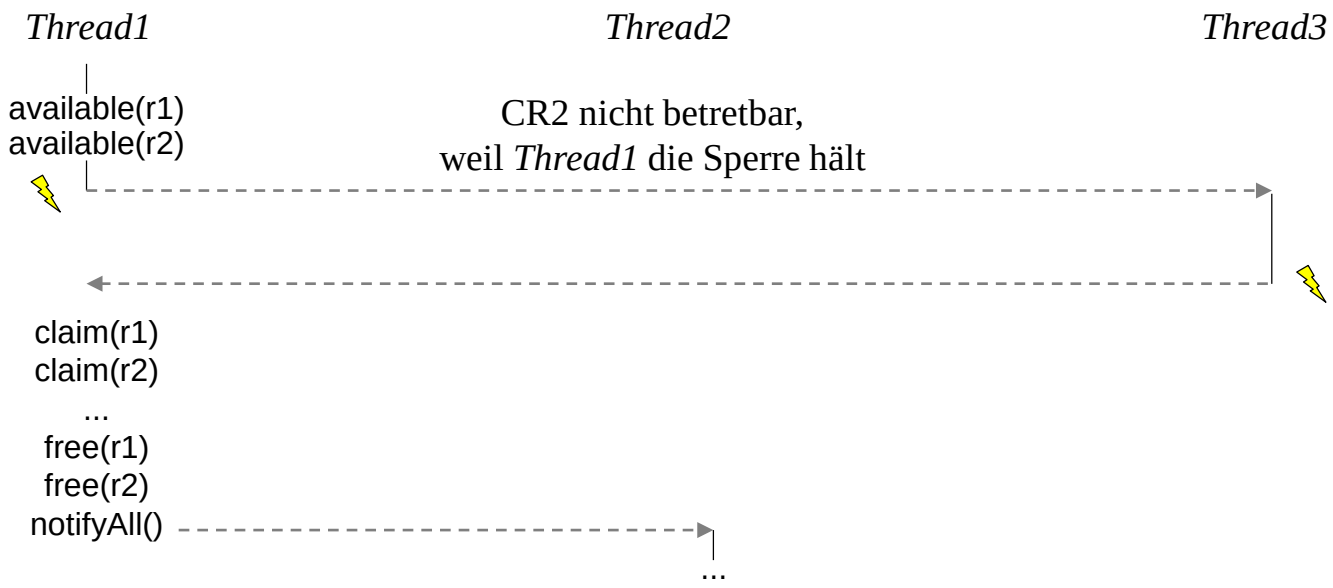
```
synchronized (this) {
   while (!available(r1) || !available(r2))
      wait();
   claim(r1);
   claim(r2);
   ... use r1 and r2 ...
   free(r1);
   free(r2);
   notifyAll();
}
```

CR2

... detto ...

Voraussetzung

Anforderung aller gemeinsam benötigten Ressourcen muss in *synchronized*-Block stattfinden

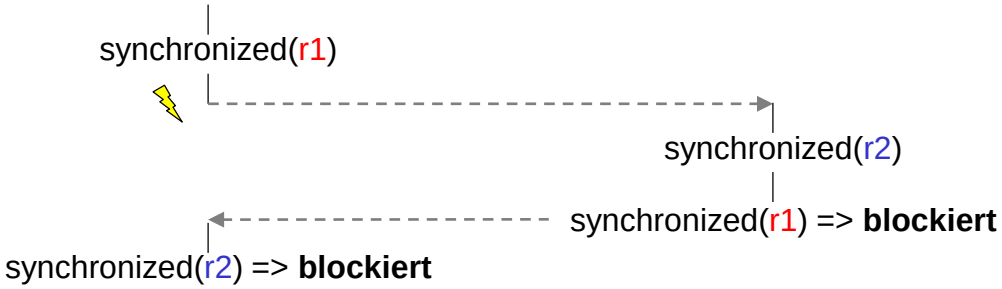


Deadlock-Gefahr trotz synchronized

Bei geschachtelter Anforderung von Sperren

```
synchronized (r1) {
    ...
    synchronized (r2) {
        ... use r1 and r2 ...
    }
}
```

```
synchronized (r2) {
    ...
    synchronized (r1) {
        ... use r1 and r2 ...
    }
}
```



Vermeidung von Deadlocks

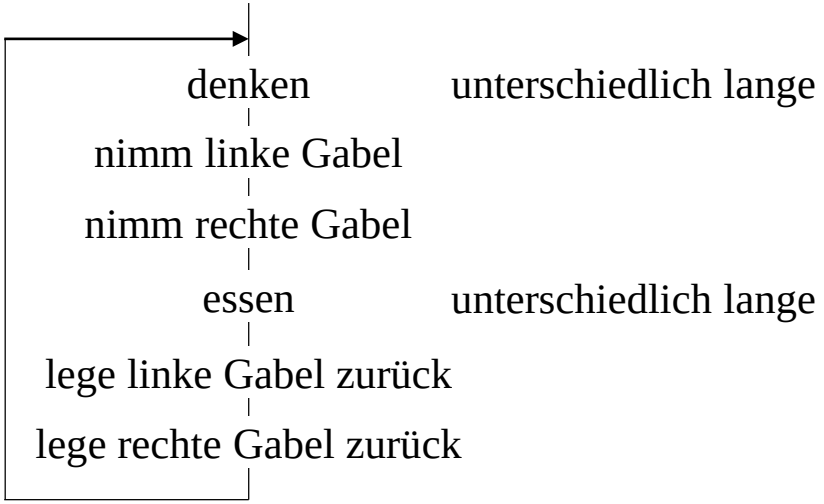
1. Kein *synchronized*-Block darf ewig brauchen
2. Geschachtelte Sperren müssen immer in derselben Reihenfolge angefordert werden

```
synchronized (r1) {
    ...
    synchronized (r2) {
        ... use r1 and r2 ...
    }
}
```

Beispiel: Dining Philosophers

5 Philosophen sitzen an einem runden Tisch, zwischen ihnen liegt je eine Gabel

Jeder Philosoph macht folgendes



Deadlock-Gefahr

Wenn zufällig alle Philosophen gleichzeitig die linke Gabel nehmen,
warten sie ewig auf die rechte Gabel

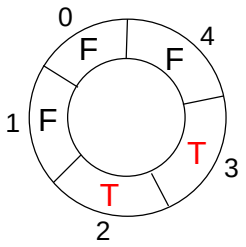
2 Ressourcen: linke Gabel, rechte Gabel
Müssen mit Synchronisation angefordert werden
(gleichzeitig oder in derselben Reihenfolge)

Philosoph als Thread

```
class Philo extends Thread {
   int n;                          // Nummer des Philosophen
   int thinkDelay, eatDelay;       // Denkdauer, Essensdauer
   Forks forks;                    // Gabeln
   int left, right;                // Index der linken/rechten Gabel

   Philo (int n, int thinkDelay, int eatDelay, Forks forks) {
      this.n = n;
      this.thinkDelay = thinkDelay; this.eatDelay = eatDelay;
      this.forks = forks;
      left = n;
      right = (n+1) % 5;
   }

   public void run() {
      while (!isInterrupted()) {
         try {
            sleep(thinkDelay);
            forks.get(left, right);
            Out.println("Philo " + n + " is eating...");
            sleep(eatDelay);
            forks.put(left, right);
         } catch (InterruptedException e) { System.exit(0); }
      }
   }
}
```

Gabeln

```
boolean[] used;
```



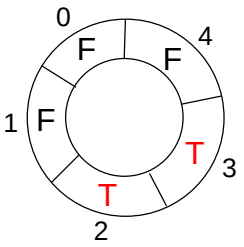
in Klasse *Forks* verwaltet

Synchronisierte Gabeln

```
class Forks {
   boolean[] used = new boolean[5]; // initially false, i.e. not used

   // Try to pick up the forks with the designated numbers
   synchronized void get(int left, int right) {
      try {
         while (used[left] || used[right]) wait();
      } catch (InterruptedException e) {
         System.exit(0);
      }
      used[left] = true;
      used[right] = true;
   }

   // Lay down the forks with the designated numbers
   synchronized void put(int left, int right) {
      used[left] = false;
      used[right] = false;
      notifyAll();
   }
}
```

Gabeln

boolean[] used;



Gabeln werden nur genommen, wenn beide frei sind

Hauptprogramm


```
public class DiningPhilosophers {
    public static void main (String[] arg) {
        Forks forks = new Forks();
        new Philo(0, 100, 500, forks).start();
        new Philo(1, 200, 400, forks).start();
        new Philo(2, 300, 300, forks).start();
        new Philo(3, 400, 200, forks).start();
        new Philo(4, 500, 100, forks).start();
    }
}
```

Ausgabe

```
Philo 0 is eating...
Philo 2 is eating...
Philo 4 is eating...
Philo 1 is eating...
Philo 3 is eating...
Philo 0 is eating...
Philo 2 is eating...
Philo 3 is eating...
Philo 1 is eating...
Philo 4 is eating...
Philo 0 is eating...
Philo 2 is eating...
Philo 3 is eating...
...
```