title: "Parallele Verarbeitung" permalink: /12-parallel/ mathjax: true

# Parallele Verarbeitung

## Prozesse

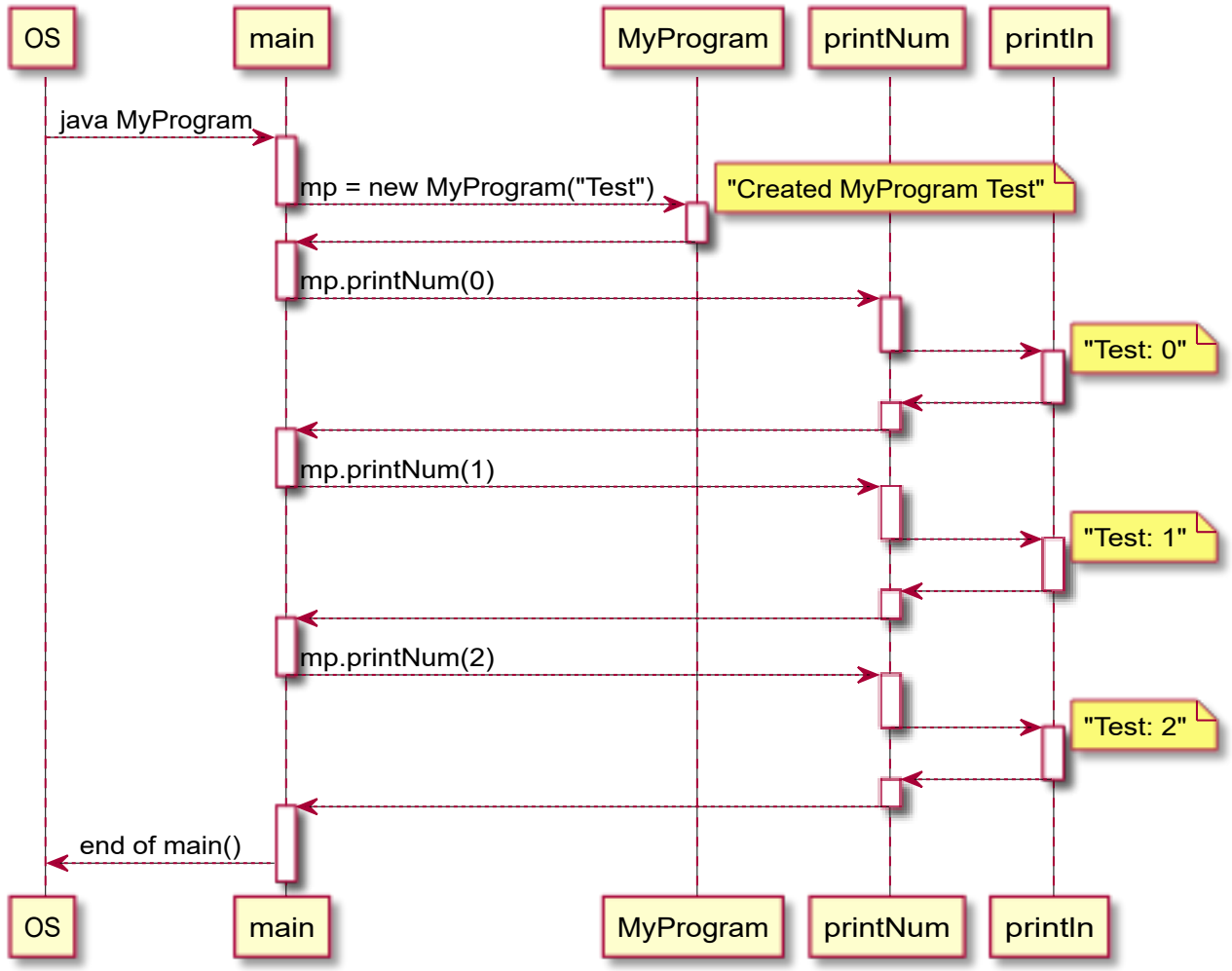
Bis jetzt waren unsere Programme alleinstehende *Prozesse*, in denen die Instruktionen einzeln, nacheinander und in der Reihenfolge ausgeführt wurden, in der sie geschrieben wurden. Das nachfolgende Programm

```
class MyProgram {
        String name;
        MyProgram(String name) {
                this.name = name;
                System.out.println("Created MyProgram: " + name);
        }
        void printNum(int n) {
                System.out.println(name + ": " + n);
        }
        public static void main(String[] args) {
                MyProgram mp = new MyProgram("Test");
                for (int i = 0; i < 3; i++)
                        mp.printNum(i);
        }
}
```

ergibt folgende Ausgabe:

```
Created MyProgram Test
Test: 0
Test: 1
Test: 2
```

Ein *Sequenzdiagramm* zeigt den Programmablauf; Methoden sind dabei als Spalten aufgetragen:




Dies entspricht auch der Reihenfolge im Debugger, wenn wir das Programm schrittweise ausführen (mit der *step-into* Anweisung).

Ein *Prozess* hat eine isolierte und eigenständige Umgebung. Er hat eine eigene, vollständige Menge an Laufzeitressourcen; das gilt insbesondere für den Speicher: Jeder Prozess hat seinenen eigenen Adressraum für Variablen.

# Threads

Die folgende Klasse `BeanCounter` modelliert einen Erbsenzähler. Im Konstruktor erhält eine Instanz einen Namen und allokiert ein großes Array von Zahlen. Der Aufruf von `run()` (des `Runnable`-Interfaces) sortiert dieses Array.

```
class BeanCounter implements Runnable {
        private final String name;
        private final double[] data;
        BeanCounter(String name, int n) {
                this.name = name;
                this.data = new double [n];
        }

        @Override
        public void run() {
```

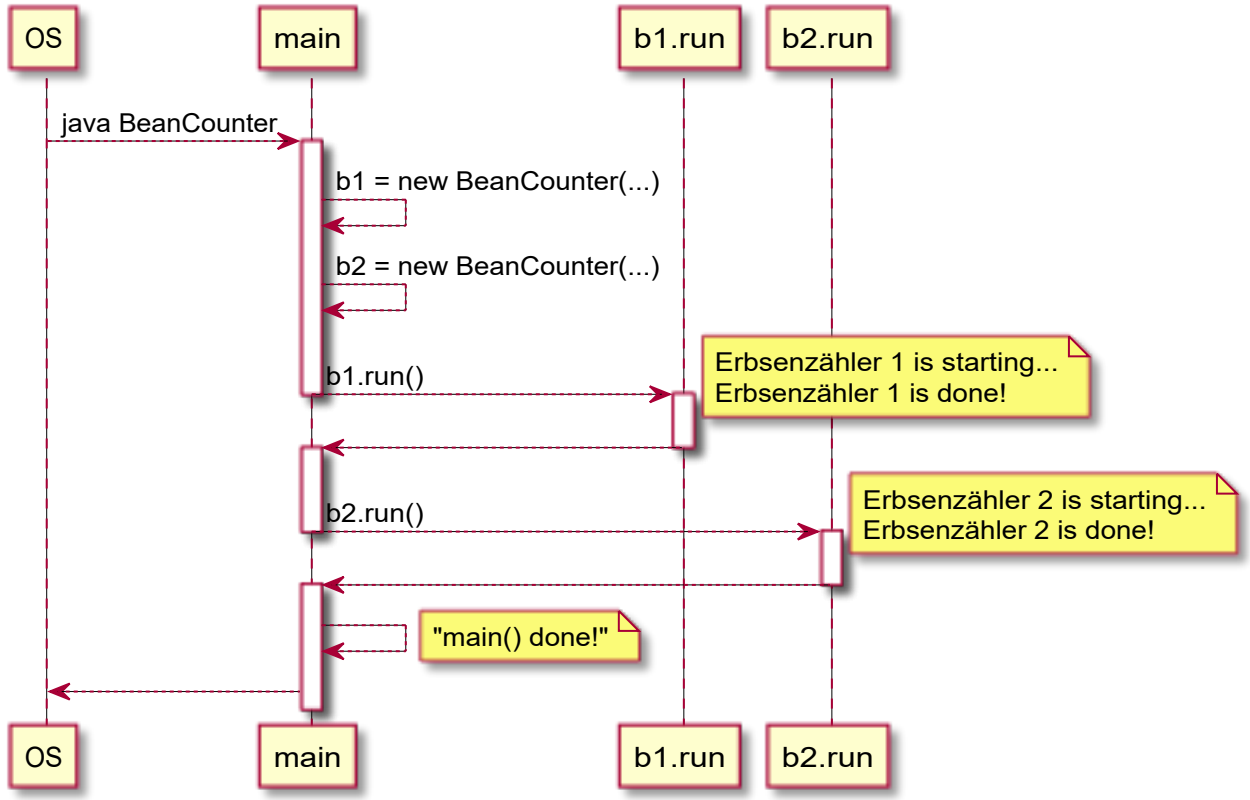
```
                System.out.println(name + " is starting...");
                Arrays.sort(data);
                System.out.println(name + " is done!");
        }
}
```

Und hier ist ein Beispiel, in dem zwei Erbsenzähler erst erstellt und dann an die Arbeit geschickt werden; Abschließend wird eine End-Nachricht ausgegeben.

```
public static void main(String... args) {
        BeanCounter b1 = new BeanCounter("Erbsenzähler 1", 10000);
        BeanCounter b2 = new BeanCounter("Erbsenzähler 2", 1000);

        b1.run();
        b2.run();

        System.out.println("main() done!");
}
```

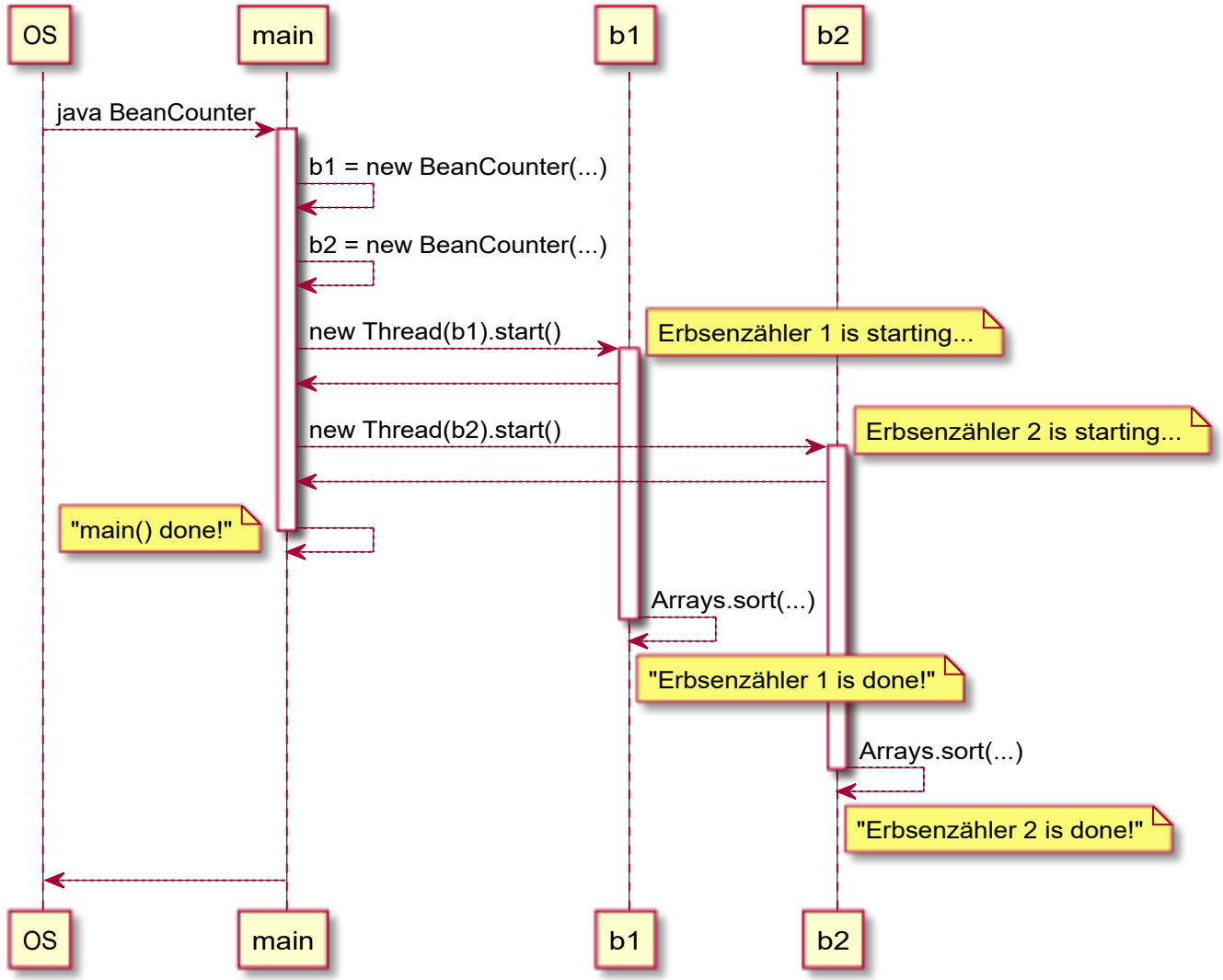
Die Ausgabe ist wie erwartet:

```
Erbsenzähler 1 is starting...
Erbsenzähler 1 is done!
Erbsenzähler 2 is starting...
Erbsenzähler 2 is done!
main() done!
```



Möchte man diese Erbsenzähler nun gleichzeitig (nebenläufig, parallel) arbeiten lassen, so kann man die `Thread` -Klasse verwenden. Diese nimmt eine Instanz von `Runnable` entgegen, deren `run()` -Methode in einem separaten Ausführungsstrang (engl. *thread*) abläuft, sobald die Methode `start()` des Threads aufgerufen wird.

```
public static void main(String[] args) {
        BeanCounter b1 = new BeanCounter("Erbsenzähler 1", 10000);
        BeanCounter b2 = new BeanCounter("Erbsenzähler 2", 1000);

        new Thread(b1).start();
        new Thread(b2).start();

        System.out.println("main() done!");
}
```




Die Ausgabe könnte nun so aussehen:

```
Erbsenzähler 1 is starting...
main() done!
Erbsenzähler 2 is starting...
```

```
Erbsenzähler 1 is done!
Erbsenzähler 2 is done!
```

Die drei Methoden `main`, `b1.run` und `b2.run` wurde *parallel* abgearbeitet. Je nach Prozessoranzahl und -auslastung kann auch eine andere Reihenfolge der Zeilen ausgegeben werden. Das liegt daran, dass zu jeder Zeit nur genau ein Thread auf `System.out` schreiben darf.

Anstatt dem `Thread` ein `Runnable` zu geben, kann man auch von `Thread` erben und die `run` Methode überschreiben.

Threads werden manchmal *leichtgewichtige Prozesse* genannt; solche existieren *in Prozessen* und haben geteilte Resourcen. Das ermöglicht einerseits die Kommunikation der Threads, bringt andererseits aber Risiken mit sich.

### Beispiele

Nebenläufige Programmierung mit Threads ist in allen modernen Applikationen zu finden:

- Browser: gleichzeitiges Laden und Rendern von Ressourcen auf einer Webseite
- Gleichzeitiges Rendering mehrerer Animationen
- Behandlung von Benutzerinteraktionen wie Klicks oder Wischen
- Sortieren mit Teile-und-Herrsche-Verfahren
- Gleichzeitige Datenbank-, Netzwerk- und Dateioperationen
- Steuerbarkeit von lang laufenden Prozessen

# Synchronisation

## Joining (Zusammentreffen)

Das obige Codebeispiel hat einen Nachteil: Die `main` Methone ist zu Ende bevor die eigentliche Arbeit (das Sortieren) abgeschlossen ist. Das erkennt man daran, dass die entsprechende Ausgabe *vor* den "is done" Ausgaben erscheint. Übertragen auf das echte Leben heisst das: man deligiert Arbeit an das Team, meldet aber nach oben sofort, dass alles erledigt ist (obwohl das Team noch arbeitet).

Eine (sehr schlechte) Lösung dafür ist es, *aktiv zu warten* (engl. *active wait*) bis ein Thread abgeschlossen hat, indem man wiederholt die `isAlive` Methode aufruft.

In folgendem Beispiel soll ein `Runnable` 15 Sekunden schlafen (also nichts tun):

```
public class Joining implements Runnable {
        @Override
        public void run() {
                System.out.println("Sleeping for 15 seconds");
                try {
                        Thread.sleep(15000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
```

```
        }

        public static void main(String[] args) {
                Thread t = new Thread(new Joining());
                t.start();

                while (t.isAlive())
                        ;  // do nothing, but really fast...

                System.out.println("Done!");
        }
}
```

Dieses aktive Warten funktioniert zwar, ist aber eine schreckliche Idee: Die Statusabfrage mit `isAlive` ist sehr schnell, wodurch der `main` Thread "heiss läuft" ohne wirklich etwas (sinnvolles) zu leisten.

Eine viel elegantere Lösung ist es, mit der `join` Methode von `Thread` auf diesen zu warten:

```
public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(new Joining());
        t.start();

        t.join();  // block/sleep until t is done
        System.out.println("Done!");
}
```

Die Methoden `join` und `sleep` können hier Ausnahmen vom Typ `InterruptedException` werfen, welche entsprechend behandelt werden müssen.

> `join` kann immer dann verwendet werden, wenn Zugriff auf die Thread-Referenz besteht. So kann man z.B. eine Threadreferenz an einen anderen Thread geben, um z.B. diesen Thread erst nach Ende des anderen Threads starten zu lassen.

## Geteilte Ressourcen

Aber zurück zum Beispiel mit den Erbsenzählern. Angenommen man hat ein Team von Erbsenzählern, welche alle den gleichen Zähler verwenden. Das ist im Prinzip möglich, da Threads sich den Speicher teilen:

```
class Counter {
        private int c = 0;
        int getCount() {
                return c;
        }
        void increment() {
                c = c + 1;
        }
}
```

```
public class TeamBeanCounter implements Runnable {
        Counter c;
        TeamBeanCounter(Counter c) {
                this.c = c;
        }

        @Override
        public void run() {
                for (int i = 0; i < 100000; i++) {
                        c.increment();
                }
                System.out.println("Total beans: " + c.getCount());
        }
}
```

Es erhält also jeder `TeamBeanCounter` eine Referenz auf den gemeinsamen Zähler, welcher in der jeweiligen `run` Methode 100000 mal erhöht wird. Abschließend wird der Zählerstand ausgegeben.

Entsprechend sollte das folgende Programm...

```
public static void main(String[] args) {
        Counter c = new Counter();

        new Thread(new TeamBeanCounter(c)).start();
        new Thread(new TeamBeanCounter(c)).start();
        new Thread(new TeamBeanCounter(c)).start();
        new Thread(new TeamBeanCounter(c)).start();
}
```

...etwas in der Art ausgeben:

```
...
Total beans: 400000
```

Wir erhalten aber stattdessen eine Ausgabe, welche z.B. so aussehen könnte:

```
Total beans: 362537
```

Was ist passiert? Alle Erbsenzähler teilen sich *die selbe* `Counter` Instanz, verwenden sie aber jeweils in ihren eigenen Ausführungssträngen. Sehen wir uns dazu die `increment` Methode genauer an:

```
void increment() {
        c = c + 1;
}
```

Hier ist es nun wichtig zu verstehen, wie solche Befehle tatsächlich auf der CPU bzw. in der JVM ausgeführt werden: Es muss erst der Wert von `c` geladen werden, dann `1` addiert, und das Ergebnis schließlich `c` zugewiesen werden. Zerlegt in einzelne Instruktionen sieht der Anweisungsblock nun so aus:

```
void increment() {
        int tmp = c;
        ++tmp;
        c = tmp;
}
```

Da nun aber jeder Thread seinen eigenen Ausführungsstrang hat, kann es sein, dass zwei Threads den Instruktionszeiger (also quasi die Zeile im Programm) an verschiedener Position haben, sich aber wie oben besprochen den Speicher teilen. Es können so also Wechselwirkungen zwischen zwei oder mehreren Threads entstehen:

| | **Thread 1** | **Thread 2** | ***result*** |
|---|---|---|---|
| 1 | tmp1 = c | | tmp1 = 0 |
| 2 | | tmp2 = c | tmp2 = 0 |
| 3 | ++tmp1 | | tmp1 = 1 |
| 4 | | ++tmp2 | tmp2 = 1 |
| 5 | c = tmp1 | | c = 1 |
| 6 | | c = tmp2 | **c = 1 !** |

Da Threads den Speicher teilen, lesen beide zunächst den selben Wert von `c` in eine eigene Hilfsvariable (`tmp1` und `tmp2`, zu Beginn `0`). Dann erhöhen beide separat ihre Hilfsvariablen, bevor sie die interne Zählvariable `c` mit ihrem (nun falschen) Wert überschreiben.

Um das zu verhindern, müssen wir als Programmierer der JVM mitteilen, welche Programmabschnitte *nur von einem Thread gleichzeitig* betreten werden dürfen. Dieses sogenannte Sperren (engl. *locking*) wird in Java mit dem Schlüsselwort `synchronized` erreicht, entweder als Modifikator für Methoden, oder als Blockanweisung mit einem Schlüsselobjekt.

```
synchronized void increment() {
        c = c + 1;
}
```

Oder äquivalent:

```
void increment() {
        synchronized (this) {
```

```
                c = c + 1;
            }
        }
```

Hierbei ist zu beachten, dass lokale Variablen (im Beispiel oben: `tmp1` und `tmp2`) auf dem Stack angelegt werden, und damit für jeden Thread separat erstellt werden. Im Gegensatz dazu werden Variablen des Heaps (mit `new` allokierte Objekte, im obigen Beispiel der `Counter`) mit den Threads geteilt (siehe auch Java VM Spezifikation).

Jeder Thread, welcher einen sog. *kritischen Abschnitt* (engl. *critical section*) betreten möchte, muss zunächst den Schlüssel (hier: `this`) "an sich nehmen", oder warten, bis dieser wieder verfügbar ist. Nach Durchlaufen des kritischen Abschnittes wird der Schlüssel wieder freigegeben.

Für die Blocksyntax wird das Schlüsselobjekt als Argument zum Schlüsselwort `synchronized` spezifiziert. Dieses kann im Prinzip jedes Objekt sein, und ist implizit `this`, wenn `synchronized` als Methodenmodifikator verwendet wird. Verwendet man nun eine der beiden Methoden, so ist die Ausgabe des `TeamBeanCounter` wie ursprünglich erwartet:

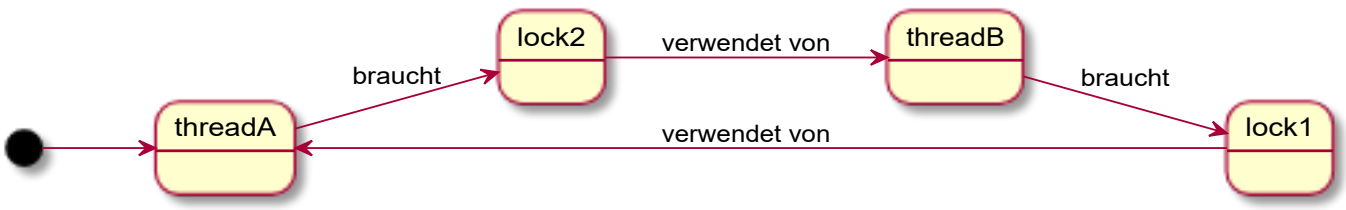
```
Total beans: 400000
```

## Synchronisierte Methoden vs. Schlüsselobjekte

- Der Vorteil von `synchronized` Methoden ist, dass es oft eine sehr einfache Lösung eines Synchronisierungsproblems ist. Der Nachteil ist, dass der gesamte Methodenblock gesperrt ist.
- Der Vorteil der Blockschreibweise (`synchronized (lock) { ... }`) ist dagegen, dass das Sperren nur dann angewendet wird, wenn es wirklich notwendig ist.
- Es kann zwar jedes Objekt als Schlüssel verwendet werden, jedoch muss die selbe Instanz in allen relevanten Stellen verwendet werden.

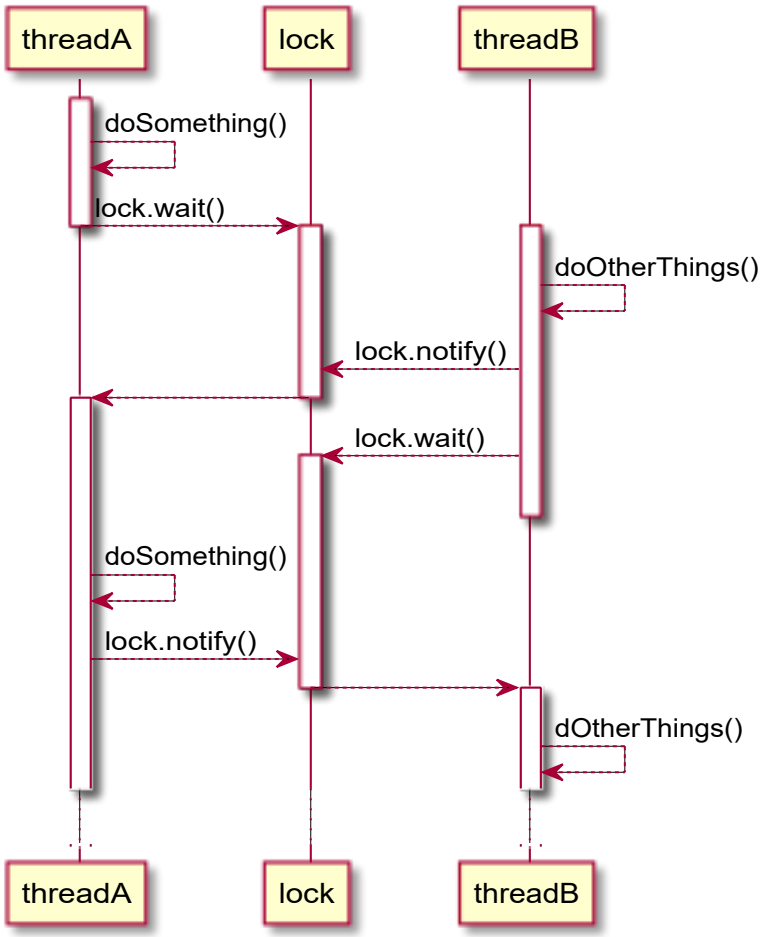
# Kommunikation

Das `synchronized`-Schlüsselwort erlaubt es uns, sicher Werte zu verändern, welche von mehreren Threads verwendet werden. Manchmal ist aber ein einfaches Sperren nicht genug, wie das folgende Diagramm veranschaulicht:




Hierbei handelt es sich um einen klassischen *Deadlock*, wie man es manchmal auch von Behörden kennt: `threadA` braucht `lock2`, welches aber von `threadB` verwendet wird; `threadB` braucht `lock1`, welches aber von `threadA` verwendet wird. Das Ergebnis ist, dass nichts vorwärts geht, die Situation ist festgefahren.

Um solchen Situationen nun vorzubeugen, kann man innerhalb von `synchronized` Abschnitten die Methoden `wait()`, `notify()` und `notifyAll()` des Schlüsselobjekts verwenden, um an bestimmten Stellen zu *warten*, bis man von einem anderen Thread *benachrichtigt* wird. Dieser Mechanismus kann verwendet werden, um Threads nur dann arbeiten zu lassen, wenn es auch etwas zu tun gibt.

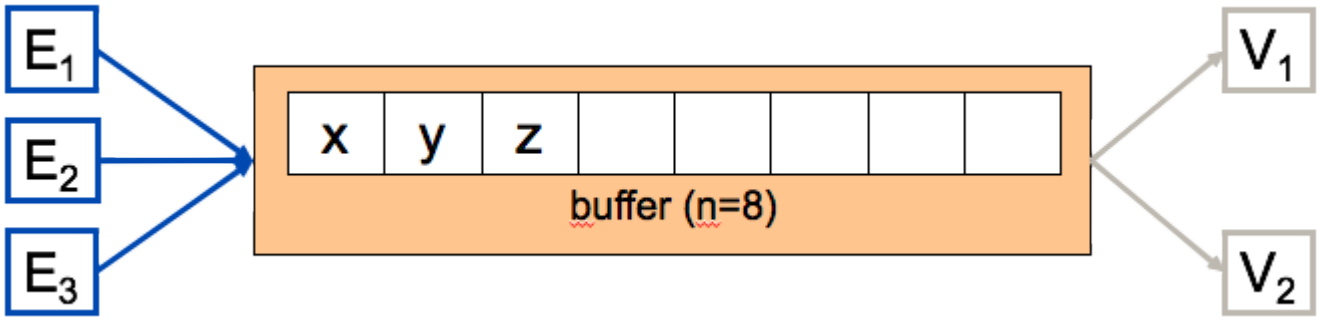



> Hinweis: Diese Methoden sind teil der Java Thread API und bereits als `final` Methoden in `Object` definiert.

> Hinweis: `notify()` weckt *einen* anderen Thread auf, `notifyAll()` entsprechend alle.

## Das Erzeuger-Verbraucher-Problem

Das klassische Beispiel zur Demonstation von `wait()` und `notify()` (bzw. `notifyAll()`) ist das *Erzeuger-Verbraucher-Problem* (engl. *consumer-producer problem*). Ein oder mehrere Erzeuger speichern Daten in eine Warteschlange, ein oder mehrere Verbraucher verarbeiten Daten, in der Reihenfolge in der sie bereit gestellt wurden. Eine typische Anwendung dieses Musters ist ein Videostreamplayer: der Erzeuger ist der Decoder, welcher den Datenstrom in Bildsequenzen umrechnet; der Verbraucher ist der Grafiktreiber, welcher die Bilder dann tatsächlich darstellt.




Ein einfaches Beispiel ist eine Erweiterung des bereits betrachteten `Counter`, hier als Klasse `ErzeugerVerbraucher` modelliert. Erzeuger rufen nun `erzeugen()` auf, um den internen Zähler zu erhöhen; Verbraucher rufen `verbrauchen()` auf, um ihn zu verringern:

```
class ErzeugerVerbraucher {
        private int c = 0;      // darf nie unter 0!
        private final int max;  // es muss immer c <= max gelten

        ErzeugerVerbraucher(int max) {
                this.max = max;
        }

        int getVerfuegbar() {
                return c;
        }

        void erzeugen() {
                c = c + 1;
        }

        void verbrauchen() {
                c = c - 1;
        }
}
```

Was passiert nun aber, wenn `c == 0` und `verbrauchen()` aufgerufen wird, oder wenn `c == max` und `erzeugen()`? Dann würde `c` in einen nicht zugelassenen Wertebereich gehen!

Eine Möglichkeit wäre es, in den entsprechenden Fällen eine `Exception` zu werfen:

```
void erzeugen() {
        if (c == max)
                throw new RuntimeException("Kein Platz mehr!!");
        c = c + 1;
}
```

Das hat aber den Nachteil, dass der aufrufende Thread dann zum einen die Ausnahme behandeln muss, und zum anderen nur *aktiv warten* kann, bis wieder Daten vorhanden sind -- was um jeden Preis vermieden

werden soll.

Der Schlüssel liegt in der Abstimmung der Threads untereinander:

- Wenn einer `erzeugen()` möchte und kein Platz ist, so muss zunächst *gewartet* werden bis etwas abgeholt wurde.
- Wenn einer `verbrauchen()` möchte und nichts da ist, so muss zunächst *gewartet* werden, bis etwas bereitgestellt wurde.

Diese Synchronisierung wird durch `wait` und `notify` (bzw. `notifyAll`) am Schlüsselobjekt und innerhalb der `synchronized` Methode bzw. Blocks realisiert:

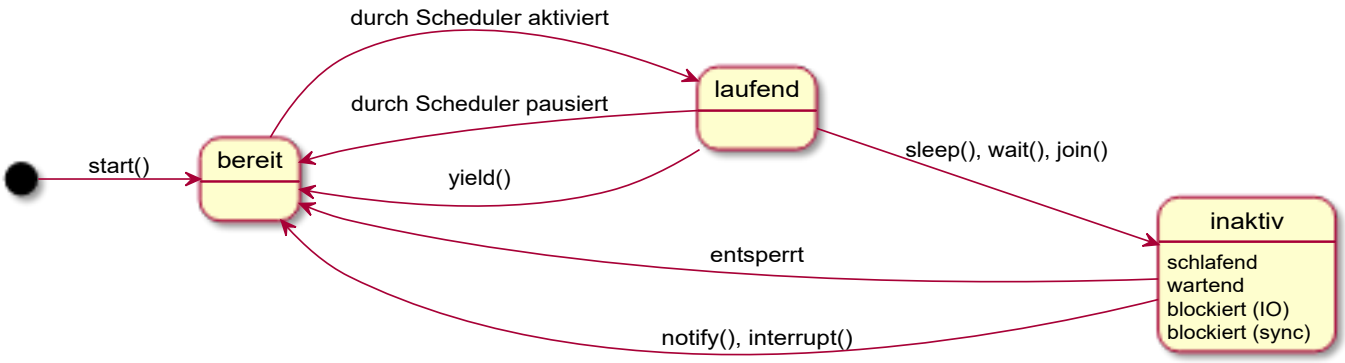
```
class ErzeugerVerbraucher {
        // ...

        synchronized void erzuegen() throws InterruptedException {
                // solange warten, bis wieder was rein passt
                while (c >= max)
                        wait();

                // jetzt passt wieder was rein!
                c = c + 1;

                // andere Threads benachrichtigen, die moeglicherweise warten
                notifyAll();
        }

        synchronized void verbrauchen() throws InterruptedException {
                // solange nicht mind. 1 Element da ist, warten!
                while (c < 0)
                        wait();

                // es gibt jetzt mind. 1
                c = c - 1;

                // andere Threads benachrichtigen, die moeglicherweise warten
                notifyAll();
        }
}
```

So können nun mehrere Verbraucher- und Erzeugerthreads dieselbe `ErzeugerVerbraucher` Instanz verwenden, und dabei einen inkonsistenten Zustand ausschließen. Wenn ein Thread `verbrauchen()` aufruft, aber nichts verfügbar ist, so wird er so lange warten (`wait()`) bis etwas bereit gestellt wurde. Wenn ein Thread `erzeugen()` aufruft, aber kein Platz ist, so wird er warten, bis wieder Platz ist.

All das funktioniert, da immer genau ein Thread gleichzeitig in den kritischen Abschnitten aktiv ist. Das ist auch der Grund, warum `wait` und `notify` **nur in kritischen Abschnitten**, also innerhalb einer `synchronized` Methode oder eines `synchronized (...)` Blocks, verwendet werden können.

# Lebenszyklus von Threads

Zum Abschluss noch der vollständige Lebenszyklus eines Threads, welcher die Wirkung von `start`, `wait`, `sleep` und `join` veranschaulicht.




# Weitere Literatur

Die heute besprochenen Konzepte sind lediglich die Grundbausteine effizienter paralleler Verarbeitung. Weitere Details können z.B. im Kapitel Concurrency des Oracle Java SE Tutorials nachgelesen werden. Im Kap. 17 der Java Language Specification ist die vollständige Spezifikation der Java Threads und Locks zu finden.

■