

Definition / Eigenschaften

Unter einem Thread versteht man einen Aktivitätsträger oder Handlungsstrang innerhalb des Adressraums eines Prozesses. Er stellt eine **Abstraktion des physischen Prozessors** dar.

Ein Thread arbeitet **sequentiell** eine **Handlungsvorschrift (Algorithmus)** quasi **parallel** zu anderen Threads ab. Es können **mehrere Threads** innerhalb eines Prozesses ausgeführt werden, die sich **gemeinsame Ressourcen (Objekte, Variablen)** innerhalb des Prozessadressraums teilen können.

Für die **korrekte Kooperation** von mehreren Threads (Verriegelungen gemeinsamer Ressourcen, notwendige Reihenfolgekoordination bei der Abarbeitung, ...) ist der **Programmierer verantwortlich!**

Threads sind insbesondere notwendig ...

- ... bei mehreren **Ein-/Ausgabekanälen** (Benutzereingaben, Kommunikationsverbindungen zu anderen Anwendungen, ...) innerhalb einer Anwendung.
- ... bei der **Auslagerung zeitintensiver Aufgaben**, bei interaktiven Anwendungen, die die **Reaktionsfähigkeit** für den Bediener immer aufrecht erhalten müssen.
- ... bei GUI -Anwendungen zur Ereignisbearbeitung und **asynchroner Signalbearbeitung**.
- ... zur **besseren Auslastung/Ausnutzung** bei Prozessoren mit mehreren Rechenkernen oder Multiprozessorsystemen.

Probleme / Tücken des Multi-Threading:

- Es sind **keinerlei Annahmen**, bei der **Abarbeitung** von Threads zulässig. Ein Thread kann „**jederzeit**“ inmitten der Abarbeitung seiner Handlungsanweisung von Betriebssystem/VM **unterbrochen** werden. Selbst eine einfache Inkrementoperation (i++) stellt keine „**unteilbare/atomare**“ Operation dar!
- **Race Conditions** (Ergebnis einer Operation ist vom zeitlichen Verhalten der Einzeloperationen abhängig) sind zu vermeiden.
- **Verriegelungen/Synchronisationen** sind nicht korrekt
- **Deadlock-Gefahr** bei 2 Threads die wechselseitig auf geblockte Ressourcen warten.
- Einsatz von „**Nicht-Thread-sicherem**“-Code/-Bibliotheken führt zu nicht offensichtlichen/sporadischen Fehlern.
- Fehler werden oft erst nach längerem Einsatz sichtbar (geändertes Timing, andere Hardware, Aus-/Belastung des Rechners, Änderungen mit Seiteneffekten) und sind schwer zu finden, und meist nicht richtig reproduzierbar.
- unsaubere Thread-Terminierung kann erhebliche Probleme verursachen (memory-leak, ...)



è Empfehlung: Vorgehen nach der „2+3 Regel“:

- erst 2-mal überlegen, ob Thread-Einsatz notwendig ist
- dann 3-mal hinschauen und Ergebnis überprüfen!

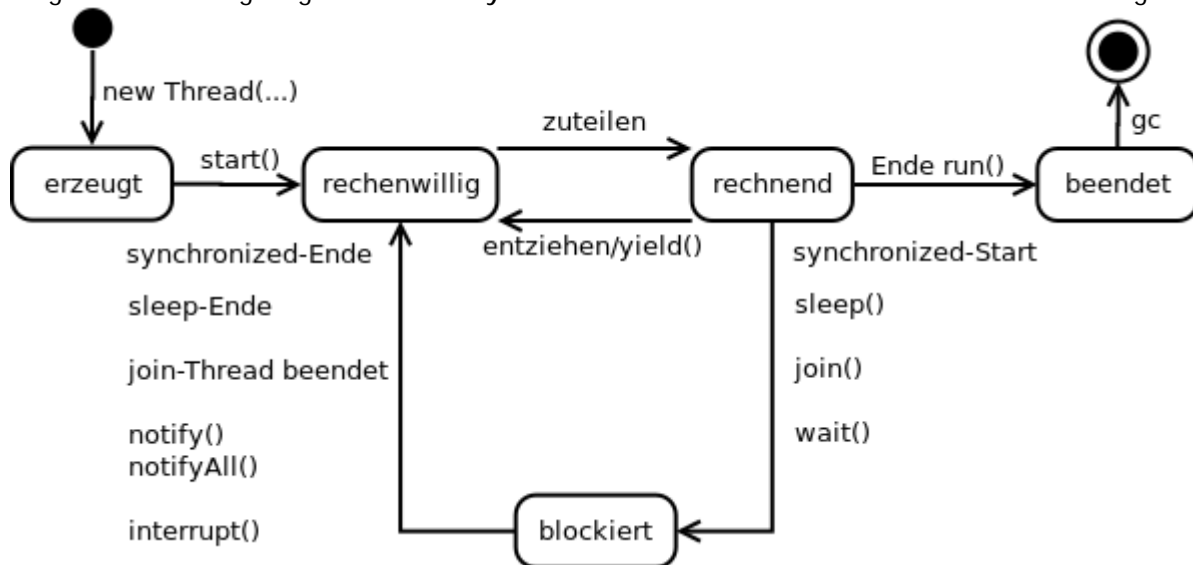
Java-Threads

Ein Thread befindet sich in genau einem Zustand des State-enums der Klasse Thread. Bedeutung:

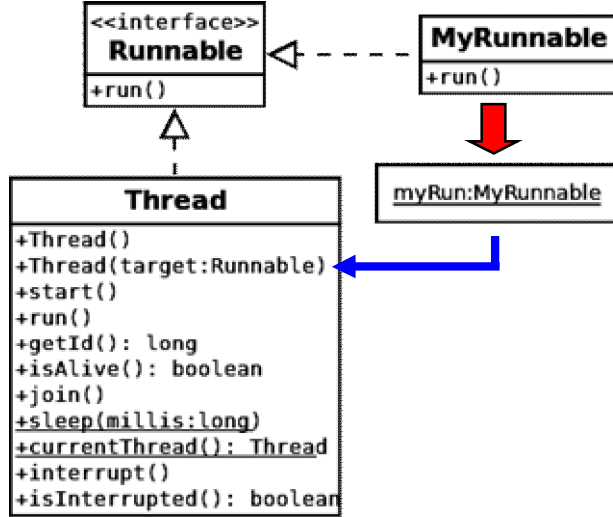
- **NEW**, wenn er noch nicht gestartet wurde,
- **TERMINATED**, wenn er bereits beendet wurde,
- **BLOCKED** wenn er auf die Zuteilung eines Locks wartet und dadurch blockiert ist,
- **WAITING**, wenn er die Methoden **Thread.join()** oder **Object.wait()** ohne Zeitangabe aufgerufen hat und daher blockiert ist,
- **TIMED_WAITING**, wenn er eine der Methoden **Thread.join()**, **Thread.sleep()**, oder **Object.wait()** mit Zeitangabe aufgerufen hat und daher blockiert ist,
- **RUNNABLE**, wenn er gerade rechnend oder rechenwillig ist oder auf eine Betriebssystemressource wartet.

```
public class Thread {  
    public enum State {  
        NEW ,  
        TERMINATED ,  
        BLOCKED ,  
        WAITING ,  
        TIMED_WAITING ,  
        RUNNABLE  
    }  
}
```

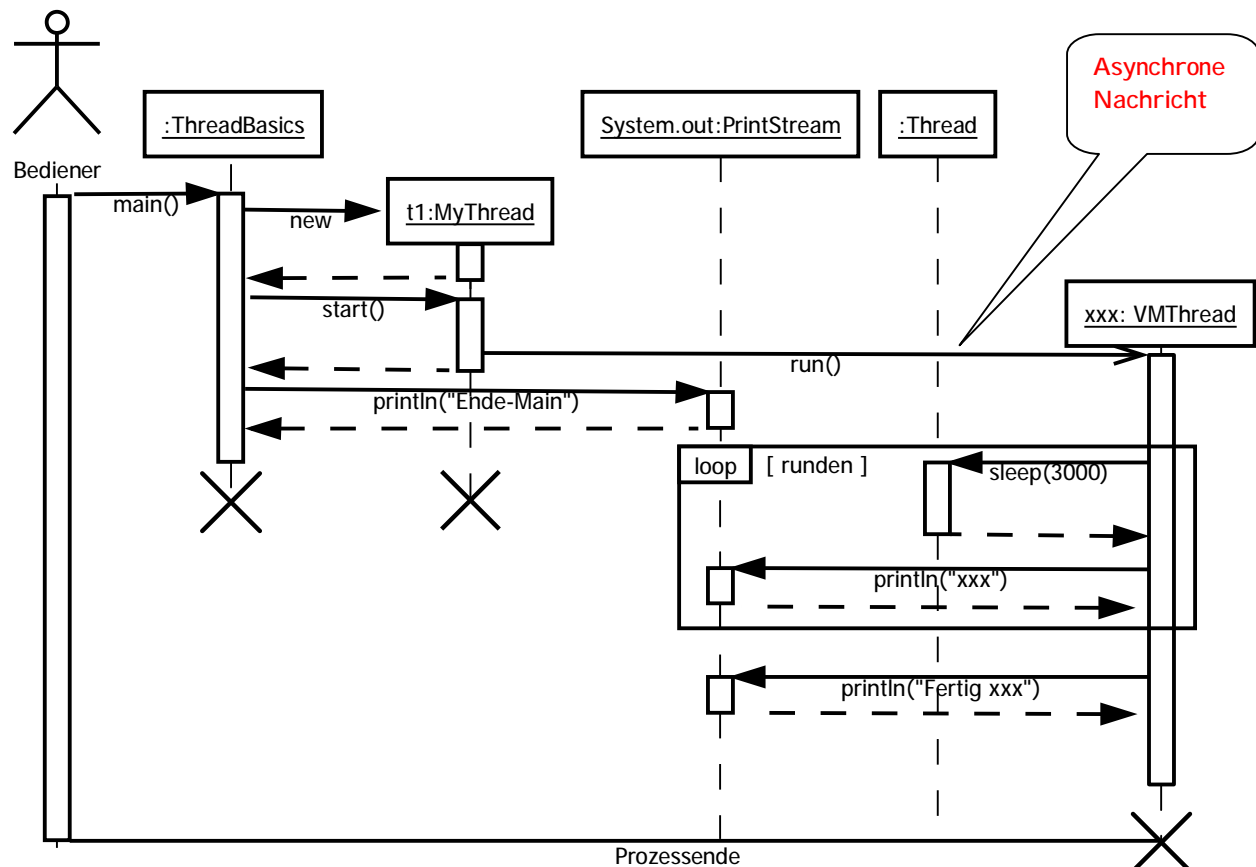
Folgende Abbildung zeigt den Lebenszyklus eines Threads in Form eines UML-Zustandsdiagramms:



Erzeugen von Threads

Möglichkeit 1	Möglichkeit 2
<ul style="list-style-type: none"> Eine eigene Klasse schreiben die von Thread erbt Methode public void run() überschreiben Objekt dieser Klasse mit new erzeugen Thread mit objekt.start() starten <pre> class MyThread extends Thread { private String text; private int runden, warteZeit; public MyThread(String text, int runden, int wZeit) { this.text = text; this.runden = runden; this.warteZeit = wZeit; } public void run() { try { // try-catch ist wegen sleep() noetig for (int i = 0; i < runden; i++) { Thread.sleep(warteZeit); System.out.println(text); } System.out.println("Fertig " + text); } catch (InterruptedException E) {} } } public class ThreadBasics { public static void main(String[] args) { MyThread t1 = new MyThread("xxx", 2, 3000); t1.start(); } } </pre>	<ul style="list-style-type: none"> Eine eigene Klasse schreiben, die die Methode public void run() des Interface Runnable implementiert Objekt dieser Klasse mit new erzeugen neuen Thread erzeugen und Objekt der eigenen „Runnable“-Klasse übergeben. Thread mit objekt.start() starten  <pre> classDiagram class Runnable { <<interface>> +run() } class MyRunnable { +run() } class Thread { +Thread() +Thread(target: Runnable) +start() +run() +getId(): long +isAlive(): boolean +join() +sleep(millis: long) +currentThread(): Thread +interrupt() +isInterrupted(): boolean } Runnable < -- MyRunnable Thread --> MyRunnable : Thread(target: Runnable) </pre>
<ul style="list-style-type: none"> Ein Thread-Objekt repräsentiert den mittels start() erzeugten Aktivitätsträger, der die run-Methode parallel zum aufrufenden Thread abarbeitet. Ein Thread terminiert, sobald die intern ausgeführte run-Methode endet. Die Lebensdauer des Thread-Objekts und die Abarbeitungsdauer der asynchron ausgeführten run-Methode ist nicht identisch! Das Thread-Objekt erlaubt Abfragen von Threadinformationen mittels getId() und isAlive() Javaprogramme besitzen immer mehrere Threads <ul style="list-style-type: none"> thread "main" und mindestens noch einen weiteren Thread "gc" den "Garbage Collector" evtl. GUI-Thread für Event-Handling 	

Das nachfolgende UML-Sequenzdiagramm zeigt den zeitlichen Ablauf des Beispielprogramms der vorangegangenen Seite:



Es können auch mehrere Threads der gleichen Klasse erzeugt werden. So erzeugt z.B. die folgende Erweiterung des main-Programms nebenstehende Ausgaben:

```

public class ThreadBasics {
    public static void main(String[] args) {
        MyThread t1 = new MyThread("xxx", 2, 3000);
        MyThread t2 = new MyThread("yyy", 5, 1000);

        t1.start();
        t2.start();

        system.out.println("main: Ausführungszustand t1: " + t1.isAlive());
        try {
            system.out.println("main: warten auf Ende t2 ...");
            t2.join();
        } catch (InterruptedException e) {}
        system.out.println("Main-Ende");
    }
}

```

```

main: Ausführungszustand t1: true
main: warten auf Ende t2 ...
yyy
yyy
xxx
yyy
yyy
yyy
Fertig yyy
Main-Ende
xxx
Fertig xxx

```

Unterbrechen / Beenden eines Threads

Ein direkter Abbruch eines Threads von außen ist nicht zulässig. Das folgende Beispiel zeigt das saubere Beenden eines Threads:

```
public static void main(String[] args) {
    MyThread2 t = new MyThread2();
    t.start();
    // irgendwas tun ...
    // jetzt soll Thread t
    // beendet werden:
    t.interrupt();
}
```

D.h. ein Thread ist dann beendet, wenn die `run()`-Methode zu Ende ist.

Mit der `interrupt()`-Methode und entsprechender Implementierung in der `run()`-Methode kann ein Thread „schnell“ beendet werden.

```
// Alternative 1 zur Bearbeitung interrupt()
class MyThread2 extends Thread {
    public void run() {
        while (!isInterrupted()) {
            System.out.println("tue was");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                // Interrupt-Flag nochmal setzen
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

```
// Alternative 2: noch einfacher!
class MyThread2 extends Thread {
    public void run() {
        try {
            while (!isInterrupted()) {
                System.out.println("tue was");
                Thread.sleep(100);
            }
        } catch (InterruptedException e) {}
    }
}
```

Ablaufsteuerung mittels `synchronized`

Mittels des Schlüsselworts `synchronized` können Methoden/Blöcke eines Objekts bzw. einer Klasse für die „gleichzeitige“ Ausführung aus mehreren Threads gesperrt werden. Die damit geschützten Programmteile werden garantiert von einem 2. Thread erst betreten, wenn der 1. Thread den kritischen Abschnitt wieder verlassen hat. „Race conditions“ (d.h. ein Ergebnis ist vom zeitlichen Verhalten der Einzeloperationen abhängig) können dadurch verhindert werden.

<code>synchronized</code>	object lock	class lock
Methode	<code>synchronized void op1(){ //code }</code>	<code>static synchronized void op1() { //code }</code>
Block	<code>synchronized(object){ //code }</code>	<code>synchronized (Klasse.class) { //code }</code>

Ermitteln sie die zu schützenden Objekte (Anzahl) und wählen dann die passende Variante aus.

Kooperation und Koordination von Threads

Die Klasse `Object` stellt Methoden zur Verfügung mit denen Threads kooperieren bzw. Threads sich koordinieren können:

- `wait()` wartet auf den Eintritt eines Ereignisses.
- `notify()` benachrichtigt einen wartenden Thread von einem Ereignis.
- `notifyAll()` benachrichtigt alle wartenden Threads, die auf den Eintritt eines Ergebnisses warten.

Object
<code>+equals(obj:Object): boolean</code> <code>+getClass(): Class</code> <code>+toString(): String</code> <code>+wait()</code> <code>+notify()</code> <code>+notifyAll()</code>

Achtung:

Diese Methodenaufrufe müssen aus einem `synchronized`-Block/-Methode erfolgen!