

Java

Collections & Maps
Threads
Kritische Abschnitte
Koordinierung



- Package: `java.util`
- Gemeinsame Schnittstelle: `Collection`
- Datenstrukturen
 - Menge
 - Schnittstelle: `Set`
 - Implementierungen: `HashSet`, `TreeSet`, ...
 - Liste
 - Schnittstelle: `List`
 - Implementierungen: `LinkedList`, `ArrayList`, ...
 - Warteschlange
 - Schnittstelle: `Queue`
 - Implementierungen: `PriorityQueue`, `ArrayBlockingQueue`, ...

Tutorial



The Java Tutorials, Trail: Collections

<http://docs.oracle.com/javase/tutorial/collections/index.html>



Generische Datenstrukturen

- Beispiel
 - Deklaration mittels generischem Datentyp (hier: `E`)

```
public interface List<E> extends Collection<E> [...]
```
 - Instanziierung: `E` durch zu verwaltenden Datentyp ersetzen, z. B. `String`

```
Collection<String> list = new LinkedList<String>();
```
- Vorteile
 - Implementierung der Datenstruktur kann Datentyp-unabhängig erfolgen
 - Typsicherheit bei Verwendung der Datenstruktur
- Einschränkung des Datentyps
 - Keine Einschränkung

```
Collection<?> c;
```
 - Einschränkung auf (Unter-)Klasse bzw. Schnittstelle

```
Collection<? extends Serializable> c;
```



Traversieren von Datenstrukturen

Iterator (`java.util.Iterator`)

- Methoden
 - `hasNext()` Überprüfung auf letztes Element
 - `next()` Voranschreiten zum nächsten Element
 - `remove()` Entfernen des aktuellen Elements

Beispiel

```
// Liste erzeugen und mit Elementen füllen
List<String> list = [...];

// Liste durchlaufen
Iterator<String> iter = list.iterator();
while(iter.hasNext()) {
    String s = iter.next();
    System.out.println(s);
}
```

- Erweiterte Iteratoren verfügbar, z. B. `java.util.ListIterator`



For-Each-Schleife

■ Bemerkungen

- Anwendbar auf alle Datenstrukturen, die die Schnittstelle `java.lang.Iterable<E>` implementieren, sowie Arrays
- Rein lesender Zugriff auf die Datenstruktur
[→ Nicht geeignet für das Einfügen von Elementen in ein Array.]

■ Beispiele

```
// Liste erzeugen und mit Elementen füllen
List<String> list = [...];
```

```
// Liste durchlaufen
for(String s: list) {
    System.out.println(s);
}
```

```
String[] array = { "A", "B", "C" };
```

```
// Array durchlaufen
for(String s: array) {
    System.out.println(s);
}
```



- Allgemeine Schnittstelle für Datenstrukturen zur Verwaltung von Schlüssel-Wert-Paaren

■ Eigenschaften

- Maximal ein Wert pro Schlüssel (→ keine Duplikate)
- Interner Aufbau bestimmt durch gewählte Implementierung
 - HashMap
 - TreeMap
 - ...

■ Beispiel

```
Map<String, Integer> telBook = new HashMap<String, Integer>();
telBook.put("Alice", 123456789);
telBook.put("Bob", 987654321);
[...]
```

```
Integer aliceNumber = telBook.get("Alice");
System.out.println("Alice's number: " + aliceNumber);
```



Algorithmen-Bibliothek

■ Verfügbare Algorithmen (Beispiele)

- Maximums- (`max()`) bzw. Minimumsbestimmung (`min()`)
- Sortieren (`sort()`)
- Überprüfung auf Existenz gemeinsamer Elemente (`disjoint()`)
- Erzeugung zufälliger Permutationen (`shuffle()`)

■ Beispiel

- Implementierung

```
Integer[] values = { 1, 2, 3, 4, 5, 6 };
```

```
List<Integer> list = new ArrayList<Integer>(values.length);
Collections.addAll(list, values);
```

```
System.out.println("Before: " + list);
Collections.shuffle(list);
System.out.println("After: " + list);
```

- Ausgabe eines Testlaufs

```
Before: [1, 2, 3, 4, 5, 6]
After:  [4, 2, 1, 6, 5, 3]
```



Überblick

Java

Collections & Maps
Threads
Kritische Abschnitte
Koordinierung



Variante 1: Unterklasse von `java.lang.Thread`

■ Vorgehensweise

1. Unterklasse von `Thread` erstellen
2. `run()`-Methode überschreiben
3. Instanz der neuen Klasse erzeugen
4. An dieser Instanz die `start()`-Methode aufrufen

■ Beispiel

```
class MWThreadTest extends Thread {
    public void run() {
        System.out.println("Test");
    }
}
```

```
Thread test = new MWThreadTest();
test.start();
```

Variante 2: Implementieren von `java.lang.Runnable`

■ Vorgehensweise

1. `run()`-Methode der `Runnable`-Schnittstelle implementieren
2. `Runnable`-Objekt erstellen
3. Instanz von `Thread` mit Hilfe des `Runnable`-Objekts erzeugen
4. Am neuen `Thread`-Objekt die `start()`-Methode aufrufen

■ Beispiel

```
class MWRunnableTest implements Runnable {
    public void run() {
        System.out.println("Test");
    }
}
```

```
Runnable test = new MWRunnableTest();
Thread thread = new Thread(test);
thread.start();
```



Pausieren von Threads

`sleep(), yield()`

■ Ausführung für einen bestimmten Zeitraum aussetzen

■ Mittels `sleep()`-Methoden

```
static void sleep(long millis);
static void sleep(long millis, int nanos);
```

- Legt den aktuellen Thread für `millis` Millisekunden (und `nanos` Nanosekunden) „schlafen“
- Achtung: Es ist nicht garantiert, dass der Thread exakt nach der angegebenen Zeit seine Ausführung fortsetzt

■ Ausführung auf unbestimmte Zeit aussetzen

■ Mittels `yield()`-Methode

```
static void yield();
```

- Aussetzen der eigenen Ausführung zugunsten anderer Threads
- Keine Informationen über die Dauer der Pause



Beenden von Threads

`return, interrupt(), join()`

■ Regulär

- `return` aus der `run()`-Methode
- Ende der `run()`-Methode

■ Abbruch nach expliziter Anweisung

- Aufruf der `interrupt()`-Methode (durch einen anderen Thread)

```
public void interrupt();
```

■ Führt zu

- einer `InterruptedException`, falls sich der Thread gerade in einer unterbrechbaren blockierenden Operation befindet
- einer `ClosedByInterruptException`, falls sich der Thread gerade in einer unterbrechbaren I/O-Operation befindet
- dem Setzen einer `Interrupt-Status-Variable`, die mit `isInterrupted()` abgefragt werden kann, sonst.

■ Auf die Terminierung eines Threads warten mittels `join()`-Methode

```
public void join() throws InterruptedException;
```



Veraltete Methoden

- Als *deprecated* markierte Thread-Methoden
 - `stop()`: Thread-Ausführung stoppen
 - `destroy()`: Thread löschen (ohne Aufräumen)
 - `suspend()`: Thread-Ausführung anhalten
 - `resume()`: Thread-Ausführung fortsetzen
 - ...
- Gründe
 - `stop()` gibt alle Locks frei, die der Thread gerade hält
 - Gefahr von Inkonsistenzen
 - `destroy()` und `suspend()` geben keine Locks frei
 - Gefahr von Deadlocks

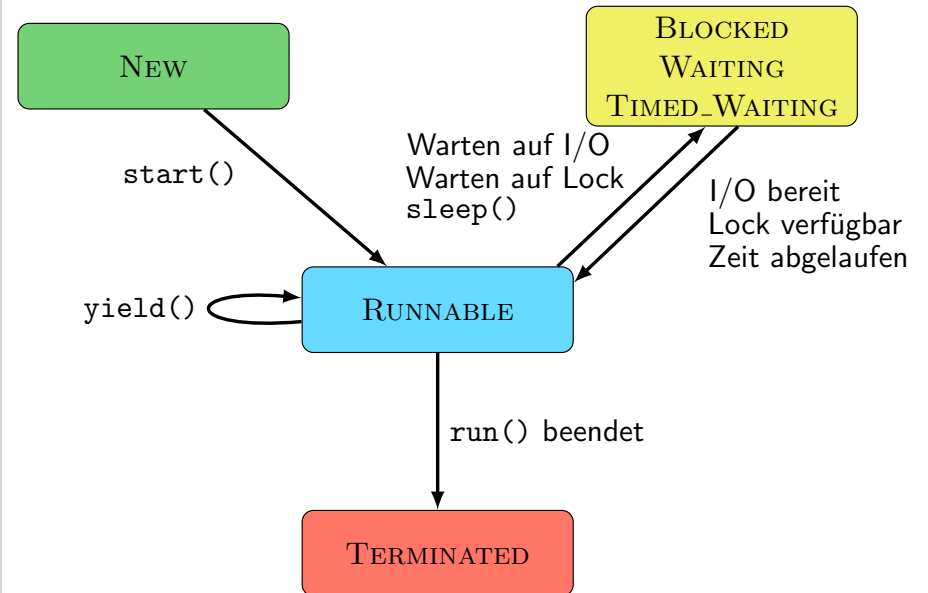
Weitere Informationen

Java Thread Primitive Deprecation

<http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html>



Thread-Zustände in Java



Überblick

Java

Collections & Maps
Threads
Kritische Abschnitte
Koordinierung



Identifizierung kritischer Abschnitte

Beispiel

```
public class MWCounter implements Runnable {
    public int a = 0;

    public void run() {
        for(int i = 0; i < 1000000; i++) {
            a = a + 1;
        }
    }

    public static void main(String[] args) throws Exception {
        MWCounter value = new MWCounter();
        Thread t1 = new Thread(value);
        Thread t2 = new Thread(value);

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

        t1.join();
        t2.join();
        System.out.println("Expected a = 2000000, " +
            "but a = " + value.a);
    }
}
```



- Ergebnisse einiger Durchläufe: 1732744, 1378075, 1506836
- Was passiert, wenn `a = a + 1` ausgeführt wird?

```
LOAD a into Register
ADD 1 to Register
STORE Register into a
```

- Mögliche Verzahnung wenn zwei Threads T_1 und T_2 beteiligt sind

0. $a = 0$;

1. T_1 -LOAD: $a = 0$, $Reg_1 = 0$

2. T_2 -LOAD: $a = 0$, $Reg_2 = 0$

3. T_1 -ADD: $a = 0$, $Reg_1 = 1$

4. T_1 -STORE: $a = 1$, $Reg_1 = 1$

5. T_2 -ADD: $a = 1$, $Reg_2 = 1$

6. T_2 -STORE: $a = 1$, $Reg_2 = 1$

⇒ Die drei Operationen müssen jeweils **atomar** ausgeführt werden!



- Code, der zu jedem Zeitpunkt nur von einem einzigen Thread ausgeführt wird, muss nicht synchronisiert werden

- Synchronisieren nötig, falls Atomizität erforderlich

1. Der Aufruf einer (komplexen) Methode muss atomar erfolgen

- Eine Methode enthält mehrere Operationen, die auf einem konsistenten Zustand arbeiten müssen

- Beispiele:

- „`a = a + 1`“

- Listen-Operationen (`add()`, `remove()`,...)

2. Zusammenhängende Methodenaufrufe müssen atomar erfolgen

- Methodenfolge muss auf einem konsistenten Zustand arbeiten

- Beispiel:

```
List list = new LinkedList();
[...]
int lastObjectIndex = list.size() - 1;
Object lastObject = list.get(lastObjectIndex);
```



- Standardansatz in Java

- Kennzeichnung eines kritischen Abschnitts mittels `synchronized`-Block
- Verknüpfung eines kritischen Abschnitts mit einem *Sperrobjekt*
- Ein Sperrobjekt kann nur von jeweils einem Thread gehalten werden

```
public void foo() {
    [...] // unkritische Operationen
    synchronized(<Sperrobjekt>) {
        [...] // kritischer Abschnitt
    }
    [...] // unkritische Operationen
}
```

- Hinweise

- Jedes `java.lang.Object` kann als Sperrobjekt dienen
- Ein Thread kann dasselbe Sperrobjekt mehrfach halten (rekursive Sperre)

- Mögliche Lösung für das Zähler-Beispiel

```
synchronized(this) { a = a + 1; }
```



- Explizites Lock

- Standardimplementierung: `java.util.concurrent.locks.ReentrantLock`
- Verwendung analog zu anderen Programmiersprachen

```
Lock lock = new ReentrantLock();
```

```
lock.lock();
[...] // kritischer Abschnitt
lock.unlock();
```

- Semaphor

- Standardimplementierung: `java.util.concurrent.Semaphore`
- Initialisierung des Zählers im Konstruktor

```
Semaphore sem = new Semaphore(1);
```

```
sem.acquire();
[...] // kritischer Abschnitt
sem.release();
```



- Standardansatz per `synchronized`-Block
 - Betreten und Verlassen des kritischen Abschnitts in derselben Methode
 - Nachteile
 - Keine Timeouts beim Warten auf ein Sperrobjekt möglich
 - Keine alternativen Semantiken für das Anfordern und Freigeben von Sperrobjekten (z. B. zur Implementierung von Fairness) definierbar
- Explizites Lock
 - Eigene Implementierung der Lock-Schnittstelle möglich
 - Methoden für bedingtes Belegen des Locks

```
boolean tryLock();
boolean tryLock(long time, TimeUnit unit);
```

- Semaphor
 - Fairness-Bedingung konfigurierbar
 - Methoden für bedingtes Dekrementieren des Zählers

```
boolean tryAcquire([int permits, ] [long timeout]);
```



- Ersatzschreibweise für einen methodenweiten `synchronized`-Block
- Sperrobjekt
 - Statische Methoden: `Class`-Objekt der entsprechenden Klasse
 - Sonst: `this`

```
class MWExample {
    synchronized public void foo() {
        [...] // kritischer Abschnitt
    }

    synchronized public void bar() {
        [...] // kritischer Abschnitt
    }
}
```

- Beachte
 - Alle `synchronized`-Methoden einer Klasse nutzen dasselbe Sperrobjekt
 - Ansatz nur sinnvoll, falls Methoden tatsächlich in Konflikt stehen



- Klasse `java.util.Collections`
 - Statische Wrapper-Methoden für `Collection`-Objekte
 - Synchronisation kompletter Datenstrukturen
- Methoden

```
static <T> List<T> synchronizedList(List<T> list);
static <K,V> Map<K,V> synchronizedMap(Map<K,V> map);
static <T> Set<T> synchronizedSet(Set<T> set);
[...]
```

- Beispiel

```
List<String> list = new LinkedList<String>();
List<String> syncList = Collections.synchronizedList(list);
```

- Beachte
 - Synchronisiert **alle** Zugriffe auf eine Datenstruktur
 - Löst Fall 1, jedoch nicht Fall 2 von Folie 2-18



- Ansatz
 - Ersatz-Klassen für problematische Datentypen
 - Atomare Varianten häufig verwendeter Operationen
 - Operation für atomares *Compare-and-Swap* (CAS)
- Verfügbare Klassen
 - Versionen für primitive Datentypen: `Atomic{Boolean,Integer,Long}`
 - Arrays: `AtomicIntegerArray`, `AtomicLongArray`
 - Referenzen: `AtomicReference`, `AtomicReferenceArray`
 - ...
- Beispiel

```
AtomicInteger ai = new AtomicInteger(47);
int newValueA = ai.incrementAndGet();
int newValueB = ai.getAndIncrement();
int oldValue = ai.getAndSet(4);
boolean success = ai.compareAndSet(oldValue, 7);
```



Java

Collections & Maps
 Threads
 Kritische Abschnitte
 Koordination



- Problemstellung
 - Rollenverteilung zwischen Threads (z. B. Produzent/Konsument)
 - Threads müssen sich abstimmen, um eine gemeinsame Aufgabe zu lösen
 → Mechanismen zur Koordination erforderlich
- Standardansatz in Java
 - Ein Thread wartet darauf, dass ein Ereignis eintritt
 - Der Thread wird mittels einer *Synchronisationsvariable* benachrichtigt
- Hinweise
 - Jedes `java.lang.Object` kann als Synchronisationsvariable dienen
 - Um andere Threads per Synchronisationsvariable zu benachrichtigen, muss ein Thread innerhalb eines `synchronized`-Blocks dieser Variable sein
- Methoden
 - `wait()` Auf eine Benachrichtigung warten
 - `notify()` Benachrichtigung an **einen** wartenden Thread senden
 - `notifyAll()` Benachrichtigung an **alle** wartenden Threads senden



Koordinierung in Java

Beispiel

■ Variablen

```
Object syncObject = new Object(); // Synchronisations-Variablen
boolean flag = false;           // Ereignis-Flag
```

■ Auf Erfüllung der Bedingung wartender Thread

```
synchronized(syncObject) {
    while(!flag) {
        syncObject.wait();
    }
}
```

■ Bedingung erfüllender Thread

```
synchronized(syncObject) {
    flag = true;
    syncObject.notify();
}
```



Koordinierung in Java

Alternativen

Bedingungsvariable

Semaphore

■ Variablen

```
Lock lock = new ReentrantLock();
Condition cnd = lock.newCondition();
boolean flag = false;
Semaphore sem = new Semaphore(0);
```

■ Auf Erfüllung der Bedingung wartender Thread

```
lock.lock();
while(!flag) {
    condition.await();
}
lock.unlock();
sem.acquire();
```

■ Bedingung erfüllender Thread

```
lock.lock();
flag = true;
condition.signal();
lock.unlock();
sem.release();
```

