Überblick

Java
- Collections & Maps
- Threads
- Synchronisation
- Koordinierung

Collections — Übersicht

- Package: `java.util`
- Gemeinsame Schnittstelle: `Collection`
- Datenstrukturen
  - Menge
    - Schnittstelle: `Set`
    - Implementierungen: `HashSet`, `TreeSet`, ...
  - Liste
    - Schnittstelle: `List`
    - Implementierungen: `LinkedList`, `ArrayList`, `Vector`, ...
  - 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 fuellen
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`

Traversieren von Datenstrukturen

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 dazu geeignet, um Elemente in ein Array einzufügen.]
- Beispiele

```
// Liste erzeugen und mit Elementen fuellen
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);
}
```

Maps

`java.util.Map<K,V>`

- 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

`java.util.Collections`

- 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

Was ist ein Thread?

- Aktivitätsträger mit eigenem Ausführungskontext
  - Instruktionszähler
  - Register
  - Stack
- Alle Threads laufen im gleichen Adressbereich
  - Arbeit auf lokalen Variablen
  - Kommunikation mit anderen Threads
- Vorteile
  - Ausführen paralleler Algorithmen auf einem Multiprozessorrechner
  - Durch das Warten auf langsame Geräte (z. B. Netzwerk, Benutzer) wird nicht das gesamte Programm blockiert
- Nachteile
  - Komplexe Semantik
  - Fehlersuche schwierig

Erzeugung von Threads in Java `java.lang.Runnable`

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

- Vorgehensweise
  1. Die `run()`-Methode der `Runnable`-Schnittstelle implementieren
  2. Objekt der neuen Klasse erzeugen, das `Runnable` implementiert
  3. Instanz von `Thread` erzeugen, dem Konstruktor dabei das `Runnable`-Objekt mitgeben
  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();
```

Erzeugung von Threads in Java `java.lang.Thread`

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

- Vorgehensweise
  1. Unterklasse von `java.lang.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();
```

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 wieder aufwacht
- Ausführung auf unbestimmte Zeit aussetzen
  - Mittels `yield()`-Methode

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

  - Gibt die Ausführung zugunsten anderer Threads auf
  - Keine Informationen über die Dauer der Pause

Beenden von Threads

`return, interrupt()`

- Regulär
  - `return` aus der `run()`-Methode
  - Ende der `run()`-Methode
- Abbruch nach expliziter Anweisung
  - Aufruf der `interrupt()`-Methode

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

  - Wird (normalerweise) von außen aufgerufen
  - 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 IO-Operation befindet
    - dem Setzen einer Interrupt-Status-Variable, die mit `isInterrupted()` abgefragt werden kann, sonst.

Auf das Beenden von Threads warten

`join()`

- Mittels `join()`-Methode

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

- Beispiel

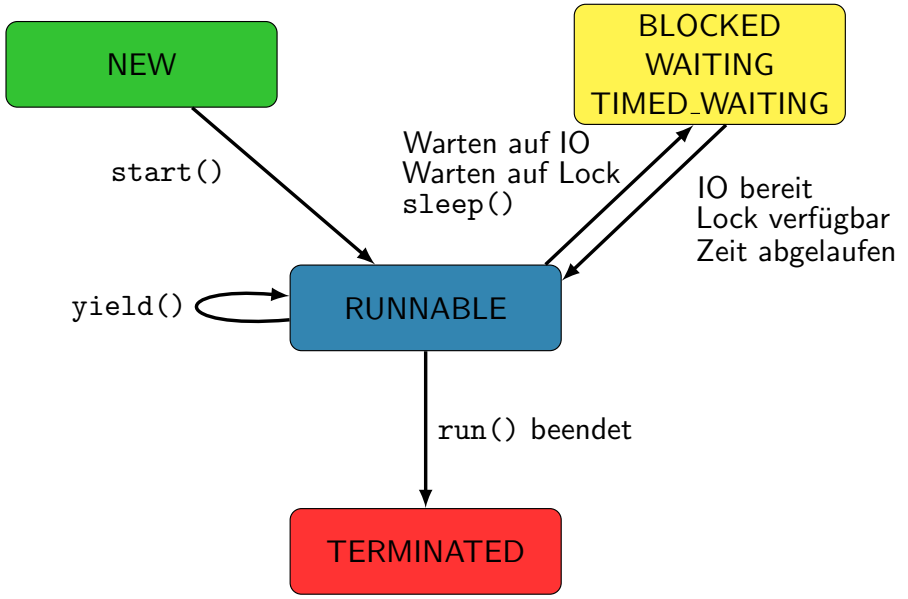
```
MWWorker worker = new MWWorker();    // implementiert Runnable
Thread workerThread = new Thread(worker);
workerThread.start();

[...]

try {
    workerThread.join();
    worker.printResult();
} catch(InterruptedException ie) {
    // Unterbrechungsbehandlung fuer join()
}
```

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
    → kann zu Inkonsistenzen führen
  - `destroy()` und `suspend()` geben keine Locks frei
- Weitere Informationen

Java Thread Primitive Deprecation
http://docs.oracle.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html

Thread-Zustände in Java



Java
- Collections & Maps
- Threads
- Synchronisation
- Koordinierung

Korrektheit nebenläufiger Programme

- Hauptaugenmerk liegt meist auf zwei Prinzipien
  - Sicherheit (*Safety*)
    - „Es passiert niemals etwas Schlechtes“
    - Beispiele:
      - Korrekte Berechnungen
      - Korrekte (Zwischen-)Zustände
      - Korrekte Ergebnisse
      - ...
  - Lebendigkeit (*Liveness*)
    - „Es passiert überhaupt irgendetwas“
    - Beispiele:
      - Keine Deadlocks
      - Stetiger Programm-Fortschritt
      - ...
- Maßnahmen
  - Synchronisation
  - Koordinierung

Synchronisationsbedarf: 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);
    }
}
```

Probleme mit Multithreading: Beispiel

- Ergebnis 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 beteiligt sind

0. a = 0;

1.	`T1-load:  a = 0, Reg1 = 0`	
2.		`T2-load:  a = 0, Reg2 = 0`
3.	`T1-add:   a = 0, Reg1 = 1`	
4.	`T1-store: a = 1, Reg1 = 1`	
5.		`T2-add:   a = 1, Reg2 = 1`
6.		`T2-store: a = 1, Reg2 = 1`

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

Synchronisation in Java

- Grundprinzip
  - Vor Betreten eines kritischen Abschnitts muss ein Thread ein *Sperrobjekt* anfordern
  - Beim Verlassen des kritischen Abschnitts wird das Sperrobjekt wieder freigegeben
  - Ein Sperrobjekt wird zu jedem Zeitpunkt von nur maximal einem Thread gehalten
- Hinweise
  - In Java kann jedes Objekt als Sperrobjekt dienen
  - Ein Thread kann das selbe Sperrobjekt mehrfach halten (rekursive Sperre)

Das Schlüsselwort `synchronized` — Übersicht

- Schutz von kritischen Abschnitten per `synchronized`-Block

```
public void foo() {
    [...] // unkritische Operationen

    synchronized(<Sperrobjekt>) {
        [...] // zu schuetzender Code (kritischer Abschnitt)
    }

    [...] // unkritische Operationen
}
```

- Ausweitung eines `synchronized`-Blocks auf die komplette Methode

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

- Verbesserung für Zähler-Beispiel

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

Das Schlüsselwort `synchronized` — Nachteile

- Die Anforderung des Sperrobjekts zu Beginn des kritischen Abschnitts (`lock()`) und seine Freigabe am Ende (`unlock()`) sind
  - nur im Java-Byte-Code sichtbar
  - nicht trennbar [→ Vorteil: kein `lock()` ohne `unlock()`.]

```
synchronized(<Sperrobjekt>) {
    [...] // zu schuetzender Code (kritischer Abschnitt)
}
```

- 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

→ Lösung: Alternative Synchronisationsvarianten (siehe später)

Wann muss synchronisiert werden?

- Atomare Aufrufe 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);
```

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

Synchronisierte Datenstrukturen `java.util.Collections`

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

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

- 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 (siehe vorherige Folie)

Java
- Collections & Maps
- Threads
- Synchronisation
- Koordinierung

Koordinierung

- Synchronisation alleine nicht ausreichend
  - Jeder Thread „lebt in seiner eigenen Welt“
  - Threads haben keine Möglichkeit sich abzustimmen
- Koordinierung unterstützt
  - Verwaltung von gemeinsam genutzten Betriebsmitteln
  - Rollenverteilung (z. B. Produzent/Konsument)
  - Gemeinsame Behandlung von Problemsituationen
  - ...

Koordinierungsbedarf: Das „Philosophen-Problem“

- Erkenntnisse
  - Das Leben eines Philosophen beschränkt sich auf zwei Tätigkeiten: Denken und Essen (abwechselnd)
  - Zum Essen benötigt man Messer **und** Gabel
- Experiment
  - 4 Philosophen werden an einem runden Tisch platziert
  - Zwischen 2 Philosophen liegt jeweils ein Messer oder eine Gabel
  - Messer und Gabel können nicht gleichzeitig genommen werden
  - Da nicht ausreichend Besteck für alle vorhanden ist, legt jeder Philosoph sein Besteck nach dem Essen wieder zurück
- Problem, falls (zum Beispiel) alle Philosophen zuerst das rechte und danach erst das linke Besteckteil nehmen wollen: **Deadlock**

→ Koordinierung notwendig

Koordinierung in Java — Übersicht

- Grundprinzip
  - Ein Thread wartet darauf, dass eine Bedingung wahr wird oder ein Ereignis eintritt
  - Der Thread wird mittels einer *Synchronisationsvariable* benachrichtigt
- Beachte
  - Jedes Java-Objekt kann als Synchronisationsvariable dienen
  - Um andere Threads über eine Synchronisationsvariable zu benachrichtigen, muss sich ein Thread innerhalb eines `synchronized`-Blocks dieser Variable befinden
- Methoden
  - `wait()` auf eine Benachrichtigung warten
  - `notify()` Benachrichtigung an **einen** wartenden Thread senden
  - `notifyAll()` Benachrichtigung an **alle** wartenden Thread senden

Koordinierung in Java — Beispiel

- Variablen

```
Object syncObject = new Object(); // Synchronisations-Variable
boolean condition = false;        // Bedingung
```

- Auf Erfüllung der Bedingung wartender Thread

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

- Bedingung erfüllender Thread

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

Explizite Locks — `java.util.concurrent.locks.*`

- Allgemeine Schnittstelle: `Lock`
  - Lock anfordern

```
void lock();
void lockInterruptibly() throws InterruptedException;
boolean tryLock();
boolean tryLock(long time, TimeUnit unit)
                           throws InterruptedException;
```

  - Lock freigeben

```
void unlock();
```

  - `Condition`-Variable für dieses Lock erzeugen

```
Condition newCondition();
```

- Implementierung: `ReentrantLock`

```
Lock lock = new ReentrantLock();
lock.lock();
[...]
lock.unlock();
```

Bedingungsvariablen — `java.util.concurrent.locks.Condition`

- Schnittstelle: `Condition`
- Auf Signal (= Erfüllung der Bedingung) warten

```
void await() throws InterruptedException; // vgl. wait()
void awaitUninterruptibly();
boolean await(long time, TimeUnit unit)
                                    throws InterruptedException;
boolean awaitUntil(Date deadline) throws InterruptedException;
```

- Signalisieren

```
void signal();    // analog zu notify()
void signalAll(); // analog zu notifyAll()
```

- Beachte

  Ein Thread, der `await*()` oder `signal*()` aufruft, muss das zugehörige Lock halten (vgl. `wait()` und `notify*()` innerhalb `synchronized`-Block)

Semaphoren

java.util.concurrent.Semaphore

- Konstruktoren

```
Semaphore(int permits);
Semaphore(int permits, boolean fair);
```

- Semaphore belegen (= herunter zählen)

```
acquire([int permits]) throws InterruptedException;
acquireUninterruptibly([int permits]);
tryAcquire([int permits, ] [long timeout]);
```

- Semaphore freigeben (= herauf zählen)

```
release([int permits]);
```

- Beispiel

```
Semaphore s = new Semaphore(1);
s.acquireUninterruptibly();
[...]
s.release();
```