XI: Nebenläufigkeit

- Threads. Definition, Verwendung.
- Synchronisation, Race conditions.
- Deadlock.
- Vermeidung von Deadlock durch Warten.

Nebenläufige Programmierung

Von Nebenläufigkeit (*concurrency*) spricht man, wenn mehrere Kontrollflüsse (Programm) gleichzeitig ablaufen und miteinander kommunizieren. Ein Programm zusammen mit Speicherbereichen in denen es abläuft bezeichnet man als *Prozess*.

Ein *Thread* (“Faden”) ist ein Prozess, der einen eigenen Keller (*stack*), aber keine eigene Halde (*heap*), hat.

Laufen mehrere Threads gleichzeitig ab, so haben sie also Zugriff auf dieselben Objekte, aber jeder Thread hat seine eigenen lokalen Variablen.

Der gemeinsame Zugriff auf die Objekte in der Halde erlaubt die Kommunikation zwischen den Threads.

Die Threads laufen in Wahrheit nicht gleichzeitig, sondern der Reihe nach, jeweils für eine kurze Zeitscheibe (*time slice*). Die Auswirkungen dieses Unterschiedes sind aber im wesentlichen unerheblich. Bei modernen Dual Core Prozessoren können Sie u.U. sogar echt simultan ablaufen.

Anwendungen von Threads

Viele professionelle Anwendungen verwenden inzwischen Threads oder ähnliche Konzepte:

- Datenbanken. Zugriffe laufen in eigenen Threads ab.
- Webserver. Anfragen werden jeweils in eigenen Threads verarbeitet.
- Fensterorientierte Benutzeroberflächen. Ein Thread für die Fenster, ein weiterer Thread für die eigentliche Anwendung.
- ...

Unser erster Thread

```
import java.util.Date;

public class Gruesser extends Thread {
    final static int DELAY = 1000;
    private String botschaft;
    Gruesser(String s) {
        botschaft = s;
    }

    public void run() {
        try {
            while (true) {
                Date jetzt = new Date();
                System.out.println(jetzt + " " + botschaft);
                sleep(DELAY);
            }
        }
```

Unser erster Thread

```
        catch (InterruptedException e) {}}

    public static void main(String[] args) {
        Gruesser th1 = new Gruesser("Guten Tag.");
        Gruesser th2 = new Gruesser("Auf Wiedersehen.");
        th1.start();
        th2.start();
    }}
```

Erklärung

- Die Klasse `Thread` stellt die Methoden `run` und `start` zur Verfügung. Ein Thread wird durch Erben von `Thread` definiert. Dabei soll man `run` überschreiben und `start` erben.
- Der Effekt von `start` ist, den Thread in einem eigenen Speicherbereich zu starten durch Aufruf der `run` Methode.
- Die `run`-Methode soll man nicht selbst aufrufen. Der Effekt wäre nur die Abarbeitung der Befehle im Rumpf von `run` ohne das Starten eines Threads.
- Das Abfangen der `InterruptedException` wird später erklärt.
- Die Methode `sleep` versetzt einen Thread für eine gegebene Zahl von Millisekunden in Schlaf.

Effekt

```
Wed Jan 16 14:23:46 CET 2008 Auf Wiedersehen.
Wed Jan 16 14:23:46 CET 2008 Guten Tag.
Wed Jan 16 14:23:47 CET 2008 Auf Wiedersehen.
Wed Jan 16 14:23:47 CET 2008 Guten Tag.
Wed Jan 16 14:23:48 CET 2008 Auf Wiedersehen.
Wed Jan 16 14:23:48 CET 2008 Guten Tag.
Wed Jan 16 14:23:49 CET 2008 Auf Wiedersehen.
Wed Jan 16 14:23:49 CET 2008 Guten Tag.
Wed Jan 16 14:23:50 CET 2008 Auf Wiedersehen.
Wed Jan 16 14:23:50 CET 2008 Guten Tag.
Wed Jan 16 14:23:51 CET 2008 Auf Wiedersehen.
Wed Jan 16 14:23:51 CET 2008 Guten Tag.
Wed Jan 16 14:23:52 CET 2008 Auf Wiedersehen.
Wed Jan 16 14:23:52 CET 2008 Guten Tag.
Wed Jan 16 14:23:53 CET 2008 Auf Wiedersehen.
Wed Jan 16 14:23:53 CET 2008 Guten Tag.
Wed Jan 16 14:23:54 CET 2008 Auf Wiedersehen.
```

Effekt

```
Wed Jan 16 14:23:54 CET 2008 Guten Tag.
Wed Jan 16 14:23:55 CET 2008 Auf Wiedersehen.
Wed Jan 16 14:23:55 CET 2008 Guten Tag.
```

Nicht nur `run` aufrufen!

```
public static void main(String[] args) {
    Gruesser th1 = new Gruesser("Guten Tag.");
    Gruesser th2 = new Gruesser("Auf Wiedersehen.");
    th1.run();
    th2.run();
}
```

Effekt:

```
Wed Jan 16 14:25:20 CET 2008 Guten Tag.
Wed Jan 16 14:25:21 CET 2008 Guten Tag.
Wed Jan 16 14:25:22 CET 2008 Guten Tag.
Wed Jan 16 14:25:23 CET 2008 Guten Tag.
Wed Jan 16 14:25:24 CET 2008 Guten Tag.
Wed Jan 16 14:25:25 CET 2008 Guten Tag.
```

Es kommt gar nicht zur Ausführung des zweiten Threads.

Threads unterbrechen

Ruft man bei einem Thread die Methode `interrupt()` auf, so kann dessen Aufmerksamkeit erlangt werden:

Die Methode `isInterrupted()` liefert dann `true` zurück;

befindet sich der Thread “im Schlaf” (aufgrund von `sleep`), so wird die Ausnahme `InterruptedException` geworfen.

Man kann dies dazu nutzen, den Thread auf Wunsch von außen vernünftig zu beenden. Beispiel: Mehrere Threads suchen in unterschiedlichen Datenbanken. Hat einer das Gesuchte gefunden, so kann er die anderen unterbrechen und auffordern, nach ordentlichem Verlassen der Datenbank zu terminieren.

Beispiel

In `Gruesser`:

```
    public void run() {
        try {
            while (true) {
                if (isInterrupted())
                    throw new InterruptedException();
                Date now = new Date();
                System.out.println(now + " " + message);
                sleep(DELAY);
            }
        }
        catch (InterruptedException e) {
        System.out.println("Fertig (" + message + ")");}
    }
```

Außerdem:

```
class WatchDog extends Thread{
```

Beispiel

```
private Thread t;
WatchDog(Thread t) { this.t = t; }
public void run() {
    try {
        sleep(10000);
        t.interrupt();
    }
    catch (InterruptedException e) {}
}}
```

In `main`;

```
WatchDog w = new WatchDog(th2);
th1.start();
th2.start();
w.start();
```

Effekt

```
Tue Jul 06 19:49:37 CEST 2004 Guten Tag.
Tue Jul 06 19:49:37 CEST 2004 Auf Wiedersehen.
Tue Jul 06 19:49:38 CEST 2004 Guten Tag.
Tue Jul 06 19:49:38 CEST 2004 Auf Wiedersehen.
Tue Jul 06 19:49:39 CEST 2004 Guten Tag.
Tue Jul 06 19:49:39 CEST 2004 Auf Wiedersehen.
Tue Jul 06 19:49:40 CEST 2004 Guten Tag.
Tue Jul 06 19:49:40 CEST 2004 Auf Wiedersehen.
Tue Jul 06 19:49:41 CEST 2004 Guten Tag.
Tue Jul 06 19:49:41 CEST 2004 Auf Wiedersehen.
Fertig (Auf Wiedersehen.)
Tue Jul 06 19:49:42 CEST 2004 Guten Tag.
Tue Jul 06 19:49:43 CEST 2004 Guten Tag.
Tue Jul 06 19:49:44 CEST 2004 Guten Tag.
Tue Jul 06 19:49:45 CEST 2004 Guten Tag.
```

Synchronisation

Greifen mehrere Threads auf dasselbe Objekt modifizierend zu, so muss sichergestellt werden, dass sie sich nicht gegenseitig in die Quere kommen.

Als Beispiel betrachten wir folgende Klasse für Bankkonten:

```
class BankAccount {
    private double balance;
    BankAccount() {balance = 0;}
    public void deposit(double amount) {
        System.out.println("Depositing " + amount);
        double newBalance = balance + amount;
        System.out.println("New balance is: " + newBalance);
        balance = newBalance;    }
    public void withdraw(double amount) {
        System.out.println("Withdrawing " + amount);
        double newBalance = balance - amount;
        System.out.println("New balance is: " + newBalance);
        balance = newBalance;}}
```

Synchronisation

Wir betrachten einen Thread, der 10-Mal hintereinander 100 EUR einzahlt:

```
class DepositThread extends Thread {
    private BankAccount account;
    private double amount;
    final static int DELAY = 100;
    DepositThread(BankAccount account, double amount) {
        this.account = account;
        this.amount = amount;}
    public void run() {
        try {
            for (int i = 0; i <= 10; i++) {
                if (isInterrupted()) throw new InterruptedExceptio
                account.deposit(amount);
                sleep(DELAY);
            }}
        catch(InterruptedException e){}
    }}
```

Synchronisation

...und einen Thread, der 10-Mal hintereinander 100EUR abhebt.

```
class WithdrawalThread extends Thread {
    private BankAccount account;
    private double amount;
    final static int DELAY = 100;
    WithdrawalThread(BankAccount account, double amount) {
        this.account = account;
        this.amount = amount; }
    public void run() {
        try {
            for (int i = 0; i <= 10; i++) {
                if (isInterrupted()) throw new InterruptedExceptio
                account.withdraw(amount);
                sleep(DELAY);
            }}
        catch(InterruptedException e){}
 }}
```

Synchronisation

Jetzt lassen wir beide Threads parallel laufen:

```
public static void main(String[] args) {
    BankAccount account = new BankAccount();
    DepositThread th1 = new DepositThread(account,100);
    WithdrawalThread th2 = new WithdrawalThread(account,100);
    th1.start();
    th2.start();}
```

Ein paarmal geht es gut:

```
Depositing 100.0
New balance is: 100.0
Withdrawing 100.0
New balance is: 0.0
Depositing 100.0
New balance is: 100.0
Withdrawing 100.0
New balance is: 0.0
```

Synchronisation

aber plötzlich:

```
New balance is: 0.0
Depositing 100.0
New balance is: 200.0
Depositing 100.0
New balance is: 100.0
Withdrawing 100.0
New balance is: 0.0
Withdrawing 100.0
New balance is: -100.0
```

Dieser Effekt passiert nicht jedesmal und auch nicht immer in derselben Weise!

Erklärung

Eine Meldung `New balance is: ...` wurde bereits ausgegeben, der neue Kontostand aber noch nicht ins Konto übertragen.

Die nächste Einzahlung bezieht sich dann auf den alten Kontostand und erhöht diesen.

Bei nebenläufiger Programmierung muss man bedenken, dass die einzelnen Statements nebenläufiger Threads beliebig verzahnt abgearbeitet werden können. Nur die Reihenfolge von Statements innerhalb eines Threads ist durch den Kontrollfluss vorgegeben.

Hängt das Programmergebnis von der Art und Weise der Verzahnung ab, so liegt eine *Race Condition* vor.

Race Conditions sind gefürchtet, da sie zu schwer vorhersagbarem und schwer reproduzierbarem Programmverhalten führen.

Eine Race Condition manifestiert sich oft nur sehr selten (etwa 1 Mal pro 1000 Programmabläufe) und ist daher durch Testen kaum aufzuspüren.

Das Schlüsselwort `synchronized`

Im Beispiel kann man dadurch Abhilfe schaffen, dass man die Methoden `deposit` und `withdrawal` mit dem Schlüsselwort `synchronized` kennzeichnet.

```
public synchronized void deposit(double amount) { . . . }
public synchronized void withdraw(double amount) { . . . }
```

Achtung: Es ist keine Lösung, `deposit` irgendwie “atomar” zu schreiben:

```
System.out.println("Depositing " + amount + "\n" +
"New balance is: " + balance+=amount):
```

Ein zusammengesetztes Statement wie letzteres wird nämlich in mehrere Bytecode - Befehle übersetzt und die können auch wieder mit anderen Befehlen verzahnt werden.

Wie wirkt `synchronized`?

Jedes Objekt ist mit einem *Monitor* ausgestattet. Dieser Monitor beinhaltet eine Boole'sche Variable und eine Warteschlange für Threads.

Ruft ein Thread eine `synchronized` Methode auf, so wird zunächst geprüft, ob die assoziierte Boole'sche Variable des Objekts `true` (="frei") ist. Falls ja, so wird die Variable auf `false` (="besetzt") gesetzt. Falls nein, so wird der aufrufende Thread *blockiert* und in die Warteschlange eingereiht.

Verlässt ein Thread eine `synchronized` Methode, so wird zunächst geprüft, ob sich wartende Threads in der Schlange befinden. Falls ja, so darf deren erster die von ihm gewünschte Methode ausführen. Falls nein, so wird die mit dem Objekt assoziierte Boole'sche Variable auf `true` (="frei") gesetzt.

Vergleich: An jedem Objekt hängt ein Mikrofon. Nur, wer es in der Hand hält, kann bei dem Objekt synchronisierte Methoden aufrufen.

Terminologie

Führt ein Thread gerade eine synchronisierte Methode bei einem Objekt aus, so sagt man, dieser Thread “halte das Lock” dieses Objekts. (*holds the objects’ lock*).

Der Monitor stellt sicher, dass zu jedem Zeitpunkt immer nur ein Thread das Lock eines Objekts halten kann.

Nur eine von i.a. mehreren synchronisierten Methoden eines Objekts kann also jeweils zu einem gegebenen Zeitpunkt ausgeführt werden.

Deadlock

Das Problem der Race Condition ist somit behoben; aber dafür handeln wir uns gleich das nächste ein: Warten zwei Threads gegenseitig auf die Freigabe von Locks, so kann keiner der beiden Threads weiterarbeiten. Es liegt eine *Verklemmung* (engl.: *Deadlock*) vor.

Vergleich: Zur Vermeidung von Verklemmungen darf man bei Stau nicht in eine Kreuzung einfahren. Täte man es doch, so könnte bei vier Kreuzungen, die ein Quadrat bilden, eine Verklemmung entstehen.

Beispiel

```
public synchronized void deposit(double amount) {
    System.out.println("Depositing " + amount);
    double newBalance = balance + amount;
    System.out.println("New balance is: " + newBalance);
    balance = newBalance;
}
public synchronized  void withdraw(double amount) {
    while (balance < amount)
        ; /* tue nichts */
    System.out.println("Withdrawing " + amount);
    double newBalance = balance - amount;
    System.out.println("New balance is: " + newBalance);
    balance = newBalance;}}
```

Wird die Programmzeile `/* tue nichts */` erreicht, so verklemmt sich das Gesamtsystem, da ja jeder Thread, der versucht `deposit` aufzurufen, sofort blockiert wird.

Abhilfe: `wait` und `notifyAll`

Jedes Objekt (also die Klasse `Object` bietet die Methoden `wait` und `notifyAll` an.

Wird `wait` aufgerufen, so wird der ausführende Thread in den *Wartezustand* versetzt.

Das Lock des Objekts wird dadurch wieder frei.

Wird `notifyAll` aufgerufen, so werden alle Prozesse, die auf das Objekt warten, in den blockierten Zustand versetzt und können so bei nächster Gelegenheit das Lock erhalten.

Zustandsmodell

Jeder Thread kann einen von vier Zuständen innehaben:

- Laufend (*running*)
- Bereit (*ready*)
- blockiert (*blocked*)
- wartend (*waiting*)

Jedes Objekt beinhaltet eine eingebaute Boole'sche Variable, eine Schlange von blockierten Threads und eine Menge von wartenden Threads.

Außerdem gibt es eine zentrale Schlange von *bereiten* Prozessen.

Zustandsübergänge

- Nach Ablauf einer Zeitscheibe wird ein laufender Thread “bereit” gemacht und der erste “bereite” Thread “laufend” gemacht.
- Beim Versuch der Ausführung einer synchronisierten Methode auf einem Objekt, dessen Lock nicht frei ist, wird der aufrufende Thread in den Zustand “blockiert” versetzt und in die Warteschlange der durch das Objekt blockierten Threads eingereiht.
- Beim Aufruf der Methode `wait` bei einem Objekt wird der aufrufende Thread in den Wartezustand versetzt und in die Menge der wartenden Threads dieses Objekts eingefügt.
- Beim Verlassen einer synchronisierten Methode wird der nächste blockierte Thread in der zugehörigen Warteschlange “bereit” gemacht.
- Beim Aufruf von `notifyAll` bei einem Objekt werden alle “wartenden” Threads bei diesem Objekt in den Zustand “blockiert” versetzt und in die Warteschlange des Objektes eingereiht.

Beispiel

```
    public synchronized void deposit(double amount) {
        System.out.println("Depositing " + amount);
        double newBalance = balance + amount;
        System.out.println("New balance is: " + newBalance);
        balance = newBalance;
        this.notifyAll();
    }
    public synchronized  void withdraw(double amount)
       throws InterruptedException {
        while (balance < amount)
            this.wait();
        System.out.println("Withdrawing " + amount);
        double newBalance = balance - amount;
        System.out.println("New balance is: " + newBalance);
        balance = newBalance;
    }
}
```

Bemerkungen

- Warum werden wartende Threads nicht einfach blockiert? A: dann würden sie ständig bereitgestellt und sofort wieder blockiert.
- Bei Benutzung von `wait` das `notifyAll` nicht vergessen.
- `notifyAll` darf nur aufgerufen werden, wenn man das Lock des entsprechenden Objektes hält (sonst `IllegalMonitorState`). Das ist insbesondere dann der Fall, wenn der Aufruf in einer synchronisierten Methode stattfindet.
- Man kann auch Teilbereiche einer Methode durch das Objekt-Lock sichern: Dazu verwendet man die Syntax

  ```
  synchronized(x /* x ein Objekt */)
  {
   . . . /* nur mit Lock zu durchlaufender Bereich */
  }
  ```

  Hier ist `x` das Objekt, dessen Lock verwendet wird, zum Beispiel `this`.

Bemerkungen

- Es gibt auch die Methode `notify`. Hier wird nur ein beliebig ausgewählter wartender Thread in den Zustand “blockiert” versetzt. Horstmann rät von `notify` ab. Problem: es kann sein, dass der jeweils ausgewählte Thread immer wieder sofort “wartend” wird, aber ein anderer, der auch wartet, den Fortschritt sicherstellen würde.

Komplizierte Deadlocks

Nicht alle Deadlocks lassen sich durch `wait` und `notifyAll` verhindern:

Es gebe drei Konten: `account0`, `account1`, `account2` und drei Threads:

- `th0` überweist immer wieder jeweils EUR500 von `account1` und `account2` auf `account0`.
- `th1` überweist immer wieder jeweils EUR500 von `account0` und `account2` auf `account1`.
- `th0` überweist immer wieder jeweils EUR500 von `account0` und `account1` auf `account2`.

Man beginnt mit EUR1000 in allen drei Konten.

Nebenläufigkeit: Zusammenfassung

- Threads sind Programmstücke, die parallel mit dem Rest des Programms ausgeführt werden (nebenläufig).
- Mit der Methode `start` wird ein Thread gestartet. Seine `run`-Methode wird dann nebenläufig abgearbeitet.
- Threads können unterbrochen werden; dies kann mit `isInterrupted` festgestellt werden.
- Eine Race Condition liegt vor, wenn das beobachtbare Programmverhalten von der Reihenfolge und der Verzahnung abhängt, in der mehrere Threads abgearbeitet werden.
- Zu einem gegebenen Zeitpunkt kann bei einem Objekt nur eine seiner synchronisierten Methoden abgearbeitet werden. Diese wird dann als Ganzes abgearbeitet.
- Durch Aufruf von `wait` wird der aufrufende Thread blockiert, bis ein anderer Prozess bei dem entsprechenden Objekt `notifyAll` aufruft.

Nebenläufigkeit: Zusammenfassung

- Ein Deadlock liegt vor, wenn Threads gegenseitig aufeinander warten. Mit `wait` und `notifyAll` lassen sich Deadlocks in manchen Fällen vermeiden.