Einführung in die Programmierung mit Java

Teil 12: Nebenläufigkeit

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

19. Dezember 2017

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 12: Nebenläufigkeit

1. Grundlagen
2. Threadunterbrechung
3. Synchronisation
   - Race condition
   - Deadlock
4. Zusammenfassung

Nebenläufigkeit vs. Paralleles Rechnen

• **Paralleles Rechnen**: Ziel ist *schnellere* Ausführung deterministischer Programme durch gleichzeitige Verwendung mehrerer Prozessoren.
Computer mit mehreren Kernen und Cloud-Architekturen sind inzwischen Standard und ermöglichen paralleles Rechnen.

• Dagegen bedeutet **Nebenläufigkeit** (engl. *Concurrency*) nicht-deterministische Berechnungen durch zufällig abwechselnd ausgeführte interagierende Prozesse.

Beispiel Reaktion auf verschiedene externe Ereignisse:
UniworX Webserver verarbeitet scheinbar gleichzeitig viele Benutzeranfragen.

Dies muss nicht unbedingt parallel erfolgen: auf einem einzelnen Prozessorkern wird einfach ständig abgewechselt.

Threads

Ein Programm zusammen mit dem Speicherbereichen, in denen es abläuft, bezeichnet man als **Prozess**. Ein Programm kann auch aus mehreren interagierenden Prozessen bestehen.

Ein **Thread** ("Faden") ist ein Prozess, der einen eigenen **Keller** (*stack*, Speicher für lokale Variablen), aber **keine** eigene **Halde** (*heap*, Speicher für Objekte), hat.

Gleichzeitig ablaufende Threads eines nebenläufigen Programms können über die gemeinsame Halde interagieren:

- 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.

Scheinbare Gleichzeitigkeit

Üblicherweise gibt es mehr Threads als Prozessorkerne.
Das Betriebssystem bzw. die Laufzeitumgebung kümmert sich darum, alle Threads auf alle verfügbaren Prozessorkerne zu verteilen.

Dabei gibt es verschiedene Strategien. Meist werden alle Threads regelmäßig unterbrochen, um alle Threads scheinbar gleichmäßig auszuführen.

Je nach dem Verhältnis zwischen Threads und Kernen laufen mehrere Threads in Wahrheit nicht gleichzeitig, sondern der Reihe nach, jeweils für eine kurze Zeitscheibe (*time slice*). Die Auswirkungen dieses Unterschiedes sind für uns aber meistens unerheblich.

Anwendungen von Threads

Viele professionelle Anwendungen verwenden Threads:

oder recht 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.
- ...

Thread und Runnable

Zur nebenläufigen Ausführung erzeugen wir für jeden Thread ein Objekt der Klasse `Thread`.
Im Konstruktor `Thread(Runnable r)` übergeben wir ein Objekt, welches das Interface `Runnable` implementiert:

```
public interface Runnable {
    public abstract void run();
}
```

Nach dem Erstellen des neuen Thread-Objekts müssen wir den Thread noch mit der Methode `void start()` starten:

```
Runnable mytask = ...
Thread thread = new Thread(mytask);
thread.start();
// more code
```

Ausführung wird mit den Befehlen nach `// more code` fortgesetzt. *Gleichzeitig* wird in einem neuen Thread jedoch auch der Code der `run`-Methode des Objekts `mytask` ausgeführt!

Unsere ersten Threads — Variante A

```
import java.util.Date;

public class Gruesser implements Runnable{
  final static int DELAY = 1000;
  private String botschaft;

  public Gruesser(String botschaft) {
      this.botschaft = botschaft;
  }
  @Override
  public void run() {
      try {
          while (true) {
              Date jetzt = new Date();
              System.out.println(jetzt + " " + botschaft);
              Thread.sleep(DELAY);
          }
      } catch (InterruptedException e) {}
  }
```

```
  public static void main(String[] args) {
      Gruesser g1 = new Gruesser("Guten Tag.");
      Thread th1 = new Thread(g1);
      Thread th2 = new Thread(new Gruesser("Auf Wiedersehen."));
      th1.start();
      th2.start();
      System.out.println("Grüßer gestartet!");
  }
}
```

Ausgabe

```
mhofmann@avila:~/W17_EIP/Vorlesung$ java Gruesser
Grüßer gestartet!
Mon Dec 18 10:26:21 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:26:21 CET 2017 Guten Tag.
Mon Dec 18 10:26:22 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:26:22 CET 2017 Guten Tag.
Mon Dec 18 10:26:23 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:26:23 CET 2017 Guten Tag.
Mon Dec 18 10:26:24 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:26:24 CET 2017 Guten Tag.
```

Erklärung Greeter

- Der Effekt von `start` ist, den Thread mit eigenem Keller (Stack) zu starten. Der neue Thread startet mit dem Aufruf der `run` Methode.
- Der ursprüngliche, startende Thread läuft auch weiter!
- Die Methode `sleep` versetzt einen Thread für eine gegebene Zahl von Millisekunden in Schlaf.
  Dabei kann eine `InterruptedException` auftreten, welche wir hier einfach mit einem leeren Handler ignorieren — das sollte man nicht machen, und wir kommen deshalb im nächsten Abschnitt auch gleich darauf zurück!

Unser erster Thread — Variante B

Alternativ kann man auch von `Thread` erben, da diese selbst `Runnable` implementiert:

```
import java.util.Date;
public class Gruesser extends Thread {
    final static int DELAY = 1000;
    private String botschaft;

    public Gruesser(String s) {
        botschaft = s;
    }
    public void run() {
        try {
            while (true) {
                Date jetzt = new Date();
                System.out.println(jetzt + " " + botschaft);
                sleep(DELAY);        // sleep wurde geerbt.
            }
        } 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();
        System.out.println("Grüßer gestartet!");
    }
}
```

Zum Starten braucht man nur ein Objekt zu erzeugen, da `Gruesser` durch die Vererbung bereits selbst zu einem `Thread` geworden ist.

VORSICHT:

Ein neuer Thread wird nur mit `start` gestartet, also niemals einfach nur die `run`-Methode selbst aufrufen!
Der Effekt wäre nur die Abarbeitung der Befehle im Rumpf von `run` ohne Starten eines neuen Threads, also ohne Nebenläufigkeit.

Effekt

```
Grüßer gestartet!
Mon Dec 18 10:30:56 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:30:56 CET 2017 Guten Tag.
Mon Dec 18 10:30:57 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:30:57 CET 2017 Guten Tag.
Mon Dec 18 10:30:58 CET 2017 Auf Wiedersehen.
...
Mon Dec 18 10:31:24 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:31:24 CET 2017 Guten Tag.
Mon Dec 18 10:31:25 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:31:25 CET 2017 Guten Tag.
Mon Dec 18 10:31:26 CET 2017 Guten Tag.
Mon Dec 18 10:31:26 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:31:27 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:31:27 CET 2017 Guten Tag.
Mon Dec 18 10:31:28 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:31:28 CET 2017 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();
    System.out.println("Grüßer gestartet!");
}
```

Effekt bei Ausgabe

```
Mon Dec 18 10:33:38 CET 2017 Guten Tag.
Mon Dec 18 10:33:39 CET 2017 Guten Tag.
Mon Dec 18 10:33:40 CET 2017 Guten Tag.
Mon Dec 18 10:33:41 CET 2017 Guten Tag.
Mon Dec 18 10:33:42 CET 2017 Guten Tag.
Mon Dec 18 10:33:43 CET 2017 Guten Tag.
Mon Dec 18 10:33:44 CET 2017 Guten Tag.
Mon Dec 18 10:33:45 CET 2017 Guten Tag.
```

Es kommt gar nicht zur Ausführung des zweiten Threads!

Thread-Erstellung

- Ein eigener Thread wird wahlweise definiert duch...
  1. Erzeugung eines normalen `Thread`-Objekts mit Übergabe eines Objekts des Interface `Runnable` im Konstruktor; oder durch
  2. Erben von `Thread`.

  In beiden Fällen soll man Methode `void run()` überschreiben.
- Ein Thread mit eigenem Keller (Stack) wird dann durch Aufruf von `void start()` gestartet.
- Der neue Thread arbeitet dann die Methode `void run()` ab. Dies geschieht nebenläufig zu dem ursprünglichen Thread, in dem `start` aufgerufen wurde.

Wenn man ohnehin nur `run` implementiert und sonst keine Methoden von `Thread` überschreibt, dann empfiehlt es sich, lediglich das Interface `Runnable` zu implementieren.
Die dynamischen Methoden von `Thread` kann man trotzdem noch über die statische Methode `Thread.currentThread()` aufrufen.

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, die Datenbank ordentlich zu verlassen und zu terminieren.

Beispiel für Unterbrechung

In der Klasse `Gruesser` schreiben wir:

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

Ein von aussen ausgelöster Interrupt setzt lediglich `isInterrupted` auf `true`; es sei denn, der Thread schläft gerade, dann wird eine Exception geworfen.

Außerdem:

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

In der `main`-Methode:

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

Ausgabe mit WatchDog

```
Mon Dec 18 10:42:00 CET 2017 Guten Tag.
Mon Dec 18 10:42:01 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:42:01 CET 2017 Guten Tag.
Mon Dec 18 10:42:02 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:42:02 CET 2017 Guten Tag.
Mon Dec 18 10:42:03 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:42:03 CET 2017 Guten Tag.
Fertig (Auf Wiedersehen.)
Mon Dec 18 10:42:04 CET 2017 Guten Tag.
Mon Dec 18 10:42:05 CET 2017 Guten Tag.
Mon Dec 18 10:42:06 CET 2017 Guten Tag.
Mon Dec 18 10:42:07 CET 2017 Guten Tag.
```

Effekt

```
Grüßer gestartet!
Mon Dec 18 10:30:56 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:30:56 CET 2017 Guten Tag.
Mon Dec 18 10:30:57 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:30:57 CET 2017 Guten Tag.
Mon Dec 18 10:30:58 CET 2017 Auf Wiedersehen.
...
Mon Dec 18 10:31:24 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:31:24 CET 2017 Guten Tag.
Mon Dec 18 10:31:25 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:31:25 CET 2017 Guten Tag.
Mon Dec 18 10:31:26 CET 2017 Guten Tag.
Mon Dec 18 10:31:26 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:31:27 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:31:27 CET 2017 Guten Tag.
Mon Dec 18 10:31:28 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:31:28 CET 2017 Guten Tag.
```

Effekt

```
Grüßer gestartet!
Mon Dec 18 10:30:56 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:30:56 CET 2017 Guten Tag.
Mon Dec 18 10:30:57 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:30:57 CET 2017 Guten Tag.
Mon Dec 18 10:30:58 CET 2017 Auf Wiedersehen.
...
Mon Dec 18 10:31:24 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:31:24 CET 2017 Guten Tag.
Mon Dec 18 10:31:25 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:31:25 CET 2017 Guten Tag.
Mon Dec 18 10:31:26 CET 2017 Guten Tag.
Mon Dec 18 10:31:26 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:31:27 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:31:27 CET 2017 Guten Tag.
Mon Dec 18 10:31:28 CET 2017 Auf Wiedersehen.
Mon Dec 18 10:31:28 CET 2017 Guten Tag.
```

Verzahnung

```
public class Concurrent implements Runnable {
  final static int DELAY = 10;
  private String botschaft;

  public Concurrent(String s) {
      botschaft = s;
  }

  public void run() {
    try {
      while (true) {
          System.out.print(botschaft);
          sleep(DELAY);
    } } catch (InterruptedException e) {}
  }
```

Verzahnung

```
public static void main(String[] args) {
  new Thread(new Concurrent("A")).start();
  new Thread(new Concurrent("B")).start();
  new Thread(new Concurrent("C")).start();
  new Thread(new Concurrent("D")).start();
}
```

Ausgabe

```
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDACBDACBDACBD
ACBDACBDACBDACBDACBDACBDADCBADCBADCBADCBADCBADCBADCB
ACDBACDBACDBACDBACDBCDABCDABCDBACBDACBDACDBADBCADBAC
DBACDABCDABCDBACDABCDBACDBACDBACDBACDBACDBACDBACDBAC
```

Reihenfolge der Abarbeitung der Einzelschritte *verschiedener* Threads ist nicht vorhersagbar. Die Möglichkeiten der **Verzahnung** der Einzelschritte wird schnell immens, so dass auch das Gesamtergebnis nicht vorhersagbar/testbar wird.

Verzahnung

```
public static void main(String[] args) {
  new Thread(new Concurrent("A")).start();
  new Thread(new Concurrent("B")).start();
  new Thread(new Concurrent("C")).start();
  new Thread(new Concurrent("D")).start();
}
```

Ausgabe

```
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDACBDACBDACBD
ACBDACBDACBDACBDACBDACBDADCBADCBADCBADCBADCBADCBADCB
ACDBACDBACDBACDBACDBCDABCDABCDBACBDACBDACDBADBCADBAC
DBACDABCDABCDBACDABCDBACDBACDBACDBACDBACDBACDBACDBAC
```

Reihenfolge der Abarbeitung der Einzelschritte *verschiedener* Threads ist nicht vorhersagbar. Die Möglichkeiten der **Verzahnung** der Einzelschritte wird schnell immens, so dass auch das Gesamtergebnis nicht vorhersagbar/testbar wird.

Verzahnung

```
public static void main(String[] args) {
  new Thread(new Concurrent("A")).start();
  new Thread(new Concurrent("B")).start();
  new Thread(new Concurrent("C")).start();
  new Thread(new Concurrent("D")).start();
}
```

Ausgabe

```
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDACBDACBDACBD
ACBDACBDACBDACBDACBDACBDADCBADCBADCBADCBADCBADCBADCB
ACDBACDBACDBACDBACDBCDABCDABCDBACBDACBDACDBADBCADBAC
DBACDABCDABCDBACDABCDBACDBACDBACDBACDBACDBACDBACDBAC
```

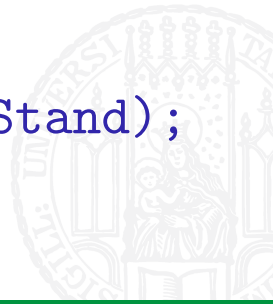
Reihenfolge der Abarbeitung der Einzelschritte *verschiedener* Threads ist nicht vorhersagbar. Die Möglichkeiten der **Verzahnung** der Einzelschritte wird schnell immens, so dass auch das Gesamtergebnis nicht vorhersagbar/testbar wird.

Synchronisation

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

Beispiel Wir betrachten wieder folgende Klasse für Bankkonten:

```
class Bankkonto {
    private double kontostand;
    Bankkonto() {kontostand = 0;}
    public void einzahlen(double amount) {
        System.out.println("Einzahlen von" + betrag);
        double neuerStand = kontostand + betrag;
        System.out.println("Neuer Kontostand: " + newKontostand)
        kontostand = neuerStand;    }
    public void abheben(double betrag) {
        System.out.println("Abheben von " + betrag);
        double neuerStand = kontostand - betrag;
        System.out.println("Neuer Kontostand: "+ neuerStand);
        kontostand = neuerStand;
}   }
```



Synchronisation

Dieser Thread zahlt 10-Mal hintereinander 100€ ein:

```
class EinzahlenThread extends Thread {
    private Bankkonto konto;
    private double betrag;
    final static int DELAY = 100;
    EinzahlenThread(Bankkonto konto, double betrag) {
        this.konto = konto;
        this.betrag = betrag;   }
    public void run() {
        try {
            for (int i = 0; i <= 10; i++) {
                if (isInterrupted())
                    throw new InterruptedException();
                konto.einzahlen(betrag);
                sleep(DELAY);
        }   }
        catch(InterruptedException e){}
}   }
```

Synchronisation

. . . und dieser hebt 10-Mal hintereinander 100€ ab:

```
class AbhebeThread extends Thread {
    private Bankkonto konto;
    private double betrag;
    final static int DELAY = 100;
    AbhebeThread(Bankkonto konto, double betrag) {
        this.konto = konto;
        this.betrag = betrag;   }
    public void run() {
        try {
            for (int i = 0; i <= 10; i++) {
                if (isInterrupted())
                    throw new InterruptedException();
                konto.abheben(betrag);
                sleep(DELAY);
        }   }
        catch(InterruptedException e){}
}   }
```

Synchronisation

Jetzt lassen wir beide Threads parallel laufen:

```
public static void main(String[] args) {
    Bankkonto konto = new Bankkonto();b
    EinzahlenThread th1 = new EinzahlenThread(konto,100);
    AbhebenThread th2 = new AbhebenThread(konto,100);
    th1.start();
    th2.start();}
```

Ein paarmal geht es gut:

```
Einzahlen von 100.0
Neuer Kontostand: 100.0
Abheben von 100.0
Neuer Kontostand: 0.0
Einzahlen von 100.0
Neuer Kontostand: 100.0
Abheben von 100.0
Neuer Kontostand: 0.0
```

Synchronisationsfehler

. . . aber plötzlich:

```
Neuer Kontostand: 0.0
Einzahlen von 100.0
Neuer Kontostand: 200.0
Einzahlen von 100.0
Neuer Kontostand: 100.0
Abheben von 100.0
Neuer Kontostand: 0.0
Abheben von 100.0
Neuer Kontostand: -100.0
```

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

Erklärung

Die Methoden `abheben` und `einzahlen` werden von den nebenläufigen Threads aufgerufen und ausgeführt.
Es kann vorkommen, dass der abhebende Thread an der kommentierten Stelle unterbrochen wird:

```
            // *** UNTERBRECHUNG HIER ***
        System.out.println("Neuer Kontostand: "+ neuerStand);
        kontostand = neuerStand;
    }
```

Vor der Fortsetzung der Ausführung wird nun der einzahlende Thread aufgerufen und führt `einzahlen` vollständig aus.
Nach der Fortsetzung steht in der *lokalen Variable* `neuerStand` ein veralteter Wert; welcher fälschlicherweise zurückgeschrieben wird!

Race condition

- Versuchen mehrere Threads gleichzeitig auf ein Objekt zuzugreifen, mindestens einer davon schreibend, so spricht man von einer **Race condition** (dt. Wettlauf oder kritische Verzahnung).
- Da die Verzahnung der Threads beliebig ist, führt dies zu schwer vorhersagbarem und schwer reproduzierbarem Programmverhalten.
- Race conditions sind unter allen Umständen zu vermeiden!
- Bei neueren Rechnerarchitekturen und Compilern können Race Conditions sogar zu Programmverhalten führen, welches *nicht* durch Verzahnung erklärt werden kann. Siehe “Java Memory Model”.

Abhilfe gegen Race Conditions: Synchronisation.

Das Schlüsselwort synchronized

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

```
public synchronized void einzahlen(double betrag) { ... }
public synchronized void abheben(double betrag) { ... }
```

Achtung:
Es ist keine Lösung, `einzahlen` irgendwie "atomar" zu schreiben:

```
System.out.println("Einzahlen von " + betrag + "\n" +
"Neuer Kontostand: " + kontostand+=betrag):
```

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 (bei einem Objekt) 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

Terminologie

- Führt ein Thread gerade eine synchronisierte Methode bei einem Objekt aus, so sagt man, dieser Thread "hält 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.

Synchronisierte Methoden

Der häufigste Fall ist die komplette Synchronisation eines Methodenrumpfes:

```
public synchronized EinTyp methname(Arg a) {
  \\ Rumpf der Methode methname
}
```

Dies ist jedoch nahezu gleich zu:

```
public EinTyp methname(Arg a) {
    synchronized(this) {
      \\ Rumpf der Methode methname
    }
}
```

Es ist also auch möglich, nur einzelne Programm-Blöcke zu synchronisieren.

Blöcke synchronisieren

Synchronisation einzelner Programm-Blöcke:

```
synchronized(lock) {
  // synchronisierter Code
}
```

- Hier ist `lock` das Objekt, dessen Lock verwendet wird.
- Es darf ein beliebiges Objekt verwendet werden.
- Ein Thread darf einen synchronisierten Bereich nur dann betreten, wenn das Lock des Lock-Objekts frei ist — ansonsten ist der Thread blockiert, bis das Lock frei ist.

Fallstrick

Auch wenn alle Methoden eines Objektes synchronisiert sind, können Probleme auftreten, z.B. bei dem Lesen mehrerer Werte:

```
int x = obj.getSyncAttribute1();
int y = obj.getSyncAttribute2();
```

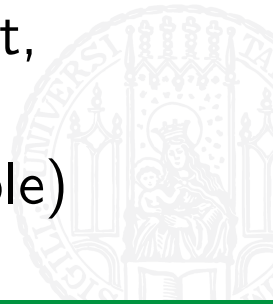
Zwischen diesen beiden Zuweisung *könnte* eine Unterbrechung stattfinden, in der sich `obj` ändert — `x` passt dann nicht mehr zu `y`!

Abhilfe durch Synchronisation

```
synchronized(obj) {
  int x = obj.getSyncAttribute1();
  int y = obj.getSyncAttribute2();
}
```

Während einer Unterbrechung werden andere Threads blockiert, welche `obj` durch synchronisierte Methoden abändern wollen.

Hinweis Die Verwendung unveränderlicher Objekte (immutable) kann solche Probleme von vornherein ausschließen.



Deadlock

Das Problem der Race Condition ist somit behoben; aber dafür handeln wir uns gleich das nächste Problem 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 Verklemmung

```
public synchronized void einzahlen(double betrag) {
    System.out.println("Einzahlen von " + betrag);
    double neuerStand = kontostand + betrag;
    System.out.println("Neuer Kontostand: " + neuerStand);
    kontostand = neuerStand;
}
public synchronized  void abheben(double betrag) {
    while (kontostand < betrag)
        ; /* tue nichts */
    System.out.println("Abheben von " + betrag);
    double neuerStand = kontostand - betrag;
    System.out.println("Neuer Kontostand: " + neuerStand);
    kontostand = neuerStand;}}
```

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

Abhilfe: `wait` und `notifyAll`

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

- Beide Methoden dürfen nur mit Objekten aufgerufen werden, deren Lock vom aktuellen Thread gehalten wird; d.h.
  1. `this.wait();` in synchronisierten Methoden
  2. `synchronized(obj) { ... obj.wait(); ... }`

  Ansonsten Ausnahme `IllegalMonitorStateException`
- Wird `wait` aufgerufen, so wird der ausführende Thread in den **Wartezustand** versetzt.
  Achtung: Das Lock des Objekts wird dadurch wieder frei!
- Wird `notifyAll` aufgerufen, so werden alle Threads, die auf das Objekt warten, in den blockierten Zustand versetzt und können so bei nächster Gelegenheit das Lock wieder erhalten.

Dies ist nützlich, wenn man darauf warten möchte, dass ein anderer Thread ein Objekt günstig abändert.

Zustandsmodell

Jeder Thread kann einen von fünf Zuständen innehaben:

- **laufend** (*running*) — wird momentan ausgeführt
- **bereit** (*ready*) — kann laufen, aber kein Kern frei
- **blockiert** (*blocked*) — sychronisierter Bereich nicht betretbar
- **schlafend** (*sleeping*) — `sleep` aufgerufen
- **wartend** (*waiting*) — `wait` aufgerufen

Jedes Objekt beinhaltet

1. eine eingebaute Boole'sche Variable zur Synchronisation,
2. eine Menge von blockierten Threads und
3. eine Menge von wartenden Threads.

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

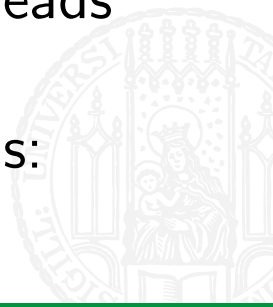
Zustandsübergänge I

- Nach Ablauf einer Zeitscheibe wird ein laufender Thread "bereit" gemacht und der erste "bereite" Thread "laufend" gemacht. Bei n-Kernen die ersten n "bereiten" Threads
- Beim Versuch der Ausführung einer synchronisierten Methode[1] auf einem Objekt, dessen Lock nicht frei ist, wird der aufrufende Thread in den Zustand "blockiert" versetzt und in die Menge der durch das Objekt blockierten Threads eingereiht.
- Beim Verlassen einer synchronisierten Methode[1] wird einer der blockierten Threads in der zugehörigen Menge "bereit" gemacht.

[1] oder auch eines synchronisierten Blocks

Zustandsübergänge II

- Ein Aufruf von `sleep(`*n*`)` signalisiert dem OS, dass der Thread ungefähr *n*-Millisekunden als "schlafend" pausieren möchte. Nach dieser Zeit wird der Thread wieder "bereit" gemacht.
  Wichtig: Alle Locks werden weiterhin gehalten und können andere Threads während des Schlafs blockieren!
- Beim Aufruf der Methode *obj*`.wait()` wird der aufrufende Thread "wartend" gemacht und in die Menge der wartenden Threads des Objekts *obj* eingefügt. Das gehaltene Lock von *obj* wird solange freigegeben.
- Beim Aufruf von *obj*`.notifyAll()` bei einem Objekt werden alle "wartenden" Threads bei diesem Objekt in den Zustand "blockiert" versetzt und in die Menge der blockierten Threads des Objektes eingereiht.
- Ein Interrupt beendet "schlafend" und "wartend" ebenfalls: ein "schlafender" Thread wird "bereit" gemacht; ein "wartender" Thread wird "blockiert"



Beispiel

```
public synchronized void einzahlen(double betrag) {
    System.out.println("Einzahlen von " + betrag);
    double neuerStand = kontostand + betrag;
    System.out.println("Neuer Stand ist " + newKontostand);
    kontostand = neuerStand;
    this.notifyAll();
}

public synchronized void abheben(double betrag)
  throws InterruptedException {
    while (kontostand < betrag)
        this.wait();
    System.out.println("Abheben von " + betrag);
    double neuerStand = kontostand - betrag;
    System.out.println("Neuer Stand ist " + neuerStand);
    kontostand = neuerStand;
}
```

Bemerkungen

Jetzt wartet ein Thread mit einer Abhebungen bei zu geringem Kontostand; nach jeder Einzahlung durch andere Threads wird geprüft, ob die Abhebung nun möglich ist und ggf. weiter gewartet.

- `wait` meistens in Schleife aufrufen: nach dem Warten muss die erwartete Bedingung noch nicht gelten zu geringe Einzahlung; anderer wartender abhebender Thread war schneller, etc.
- Bei Benutzung von `wait` das `notifyAll` an anderer Stelle nicht vergessen!
- `wait`/`notifyAll` dürfen nur aufgerufen werden, wenn man das Lock des entsprechenden Objektes hält, also in synchronisierter Methode/Block; sonst `IllegalMonitorStateException`.
- Warum werden wartende Threads nicht einfach blockiert? *Antwort:* dann würden sie ständig bereitgestellt und sofort wieder blockiert.

Notify

Es gibt auch die Methode `notify`. Hier wird *einer* der wartenden Threads *zufällig* ausgewählt und in den Zustand "blockiert" versetzt. Von der Benutzung wird abgeraten.

Problem

Es kann sein, dass die Bedingung des jeweils zufällig ausgewählten Threads nicht erfüllt ist und dieser sofort wieder "wartend" wird, aber ein anderer wartender Thread den Fortschritt des Programms sicherstellen könnte. In diesem Fall muss auf ein erneutes `notify` von einem anderen Thread gewartet werden — und dann könnte erneut ein unpassender "wartender" Thread ausgewählt werden.

Komplizierte Deadlocks

Nicht alle Deadlocks lassen sich durch `wait` und `notifyAll` verhindern.
Es gebe drei Konten: `konto0`, `konto1`, `konto2` und drei Threads:

- `th0` überweist immer wieder jeweils EUR500 von `konto1` und `konto2` auf `konto0`.
- `th1` überweist immer wieder jeweils EUR500 von `konto0` und `konto2` auf `konto1`.
- `th2` überweist immer wieder jeweils EUR500 von `konto0` und `konto1` auf `konto2`.

Man beginnt mit EUR1000 in allen drei Konten.

Weitere Probleme bei Nebenläufigkeit

Bei nebenläufigen Berechnungen mit mehreren Threads können neben *Race Conditions* und *Deadlocks* u.a. auch noch folgende Probleme auftreten:

Livelock Es werdem zwar noch Threads ausgeführt und der Zustand der Objekte ändert sich, aber diese reagieren nur noch gegenseitig aufeinander, ohne einen echten Fortschritt zu erzielen. Z.B. wird ein Warten nur durch Erledigung unwichtiger Aufgaben unterbrochen.

Starvation Ein Spezialfall eines Livelocks: *Ein wichtiger Thread* bleibt dauerhaft blockiert/wartend, weil ständig andere Threads zum Vorzug kommen, diese aber nichts Wesentliches zum Fortschritt des Programms beitragen.

Livelocks entstehen häufig bei fehlerhaften Versuchen der Deadlock-Vermeidung: z.B. wenn alle Threads ein drohendes Deadlock gleichzeitig erkennen und alle zurücktreten und es dann später gleichzeitig erneut versuchen.

Nebenläufigkeit und GUIs

Problem

Wird der GUI-Thread mit Berechnungen ausgelastet, reagiert die GUI nicht mehr auf den Benutzer.

Größenänderung, Abbrechen-Knopf, etc.

Aufwändige Berechnungen sind also nebenläufig auszuführen.
Allerdings ist der JavaFX-Szenengraph nicht **Thread-sicher**, d.h. greift man aus einem anderen Thread auf den Szenengraph von JavaFX zu, können Synchronisationsfehler auftreten.

Unproblematisch, so lange nicht direkt gezeichnet wird.

Abhilfe

JavaFX bietet im Paket `javafx.concurrent` spezielle Versionen

Entwurfsmuster: Guarded-Block

Ein Thread wartet auf die Erfüllung einer Bedingung durch andere(n) Thread(s):

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

Threads, deren Ausführung möglicherweise eine Bedingung erfüllen können, benachrichtigen alle wartenden Threads:

```
synchronized(lock) {
  condition = true;
  lock.notifyAll();
}
```

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.
- *Race Condition*: zwei Threads greifen unsynchronisiert auf dasselbe Objekt zu, mindestens einer modifizierend. Immer zu vermeiden.
- Zu einem gegebenen Zeitpunkt kann bei einem Objekt nur eine seiner synchronisierten Methoden abgearbeitet werden. Diese wird dann als Ganzes abgearbeitet.
- `wait`: der aufrufende Thread wird blockiert, bis ein anderer Prozess bei dem entsprechenden Objekt `notifyAll` aufruft.
- Ein *Deadlock* liegt vor, wenn Threads gegenseitig aufeinander warten. Mit `wait` und `notifyAll` lassen sich Deadlocks in manchen Fällen vermeiden.

Probleme bei Nebenläufigkeit

Bei nebenläufigen Berechnungen mit mehreren Threads können u.a. folgende Probleme auftreten:

Race-Condition Zwei Threads versuchen gleichzeitig auf ein Objekt zuzugreifen, mindestens einer davon schreibend. Dadurch unvorhergesehenes und schwer reproduzierbares Programmverhalten. Immer zu vermeiden.

Deadlock Ein Thread wartet auf das Ergebnis eines anderen Threads, welche direkt oder indirekt selbst auf das Ergebnis des ersten Threads wartet. Beide warten aufeinander, die Berechnung kommt somit zum erliegen.

Durch den Einsatz von Synchronisation/Locks kann man Race-Conditions vermeiden, erhöht aber prinzipiell die Gefahr von Deadlocks.

Verschiedene Threads können sich gegenseitig beeinflussen.

Manchmal wird ein Thread schneller als ein anderer abgehandelt.

Da die Möglichkeiten der Verzahnung immens sind, ist das Gesamtergebnis der Berechnung kaum vorhersagbar/testbar.

Wichtige Methoden der Klasse Thread

Statisch

- `Thread.currentThread()` liefert eigenes Thread Objekt
- `Thread.sleep(long millis)` wartet Millisekunden ab

Dynamisch

Dynamische Methoden, Aufruf über `Thread` Objekt

- `thread.join()` wartet, bis `thread` beendet ist
- `thread.interrupt()` um Unterbrechung zu signalisieren
- `thread.isInterrupted()` ob ein Interrupt vorliegt

Wenn man nicht von `Thread` geerbt hat, kann man über die statische Methode `Thread.currentThread()` das Thread-Objekt erhalten und dann damit die dynamischen Methoden aufrufen.