Nebenläufigkeit: Synchronisation kritischer Abschnitte

- Verstehen wieso synchronisiert werden muss
- Den Begriff *Kritischer Abschnitt* und *Mutual Exclusion* verstehen
- Verstehen dass auch eine einzelne Java-Anweisung während ihrer Durchführung vom Scheduler abgebrochen werden kann
- `synchronized` auf ein Objekt richtig definieren können
- `synchronized` einer Methode richtig einsetzen können
- Wissen was gesperrt und nicht gesperrt wird, wenn `synchronized` verwendet wird
- `synchronized` am Beispiel eines `ChatServers` richtig einsetzen können

Problem welches Synchronisation notwendig macht

Wenn Threads gemeinsam benutzte Objekte, Ressourcen oder Variablen gleichzeitig ändern, können Inkonsistenzen auftreten:

BankAccount

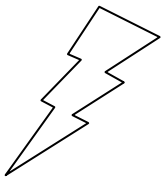
```
double amount = 0.0;

public void book(double amount) {
  double tmp = this.amount;
  tmp+=amount;
  this.amount = tmp;
}
```

Thread1 bucht fünfmal 100 auf das Konto

Thread2 bucht fünfmal -100 auf das Konto

Mögliche Ausgabe

```
T1 reads 0.0
T2 reads 0.0
T1 writes 100.0
T1 reads 100.0
T2 writes -100.0
T2 reads -100.0
T1 writes 200.0
T1 reads 200.0
…
```



Kritischer Abschnitt

Zusammengesetzte Operationen, die gemeinsame Variablen lesen und verändern. Dabei können Inkonsistenzen auftreten

Thread muss kritischen Abschnitt unter gegenseitigem Ausschluss (engl. *Mutual Exclusion*) ausführen können. Das bedeutet dass

- zu jedem Zeitpunkt höchstens ein Thread solch einen kritischen Abschnitt ausführen kann und
- andere Threads, welche im Objekt mit der Ausführung eines kritischen Abschnittes beginnen wollen, müssen warten

Konkret

```
public class AccountOperationMain
{
  public static void main(String[] args) throws InterruptedException {
    BankAccount account = new BankAccount();
    OperationThread t1 = new OperationThread("T1", account, 100);
    OperationThread t2 = new OperationThread("T2", account, -100);
    t1.start(); t2.start();
    t1.join(); t2.join();
    System.out.println("Final account balance " + account.getAmount());
  }
}
```

Thread erhält zu manipulierendes Objekt im Konstruktor

```
public class BankAccount
{
  private double amount = 0.0;

  public void book(double amount) {
    System.out.println(Thread.currentThread().getName() +
      " reads " + amount);
    double tmp = this.amount;
    tmp+=amount;
    System.out.println(Thread.currentThread().getName() +
      " writes " + tmp);
    this.amount = tmp;
  }

  …
}
```

```
public class OperationThread
  extends Thread
{

  private BankAccount account = null;
  private double amount = 0.0;

  public OperationThread(String name,
    BankAccount account, double amount) {
    setName(name);
    this.account = account;
    this.amount = amount;
  }

  @Override
  public void run() {
    for (int i = 0; i < 5; i++)
      account.book(amount);
  }
}
```

1. Lösung: Monitor auf Objekt

```
synchronized (account) {
```

account.book(amount);

Monitor ist die Kapselung eines kritischen Abschnittes mittel automatisch verwalteter Sperre

- Sperre wird beim Betreten des Monitors gesetzt und beim Verlassen wieder zurückgenommen
- Ist sie beim Betreten des Monitors bereits gesetzt, muss gewartet werden

Monitor darf nur von jeweils einem Thread zur selben Zeit durchlaufen werden

```
}
```

Vor Eintritt in den Monitor wird kontrolliert, ob Objekt gesperrt ist. Wenn ja, muss gewartet werden

```
@Override
public void run() {
  for (int i = 0; i < 5; i++)
    synchronized (account) {
      account.book(amount);
    }
  }
}
```

ACHTUNG: Positionierung von **synchronized** wichtig!!!

2. Lösung: Monitor auf Methode

```
public synchronized void book (double amount) {
```

Komplette Methode wird durch Monitor geschützt

Als Sperre wird der `this`-Zeiger verwendet. Gesperrt wird demnach jenes Objekt, auf welches Methode aufgerufen wird

```
}
```

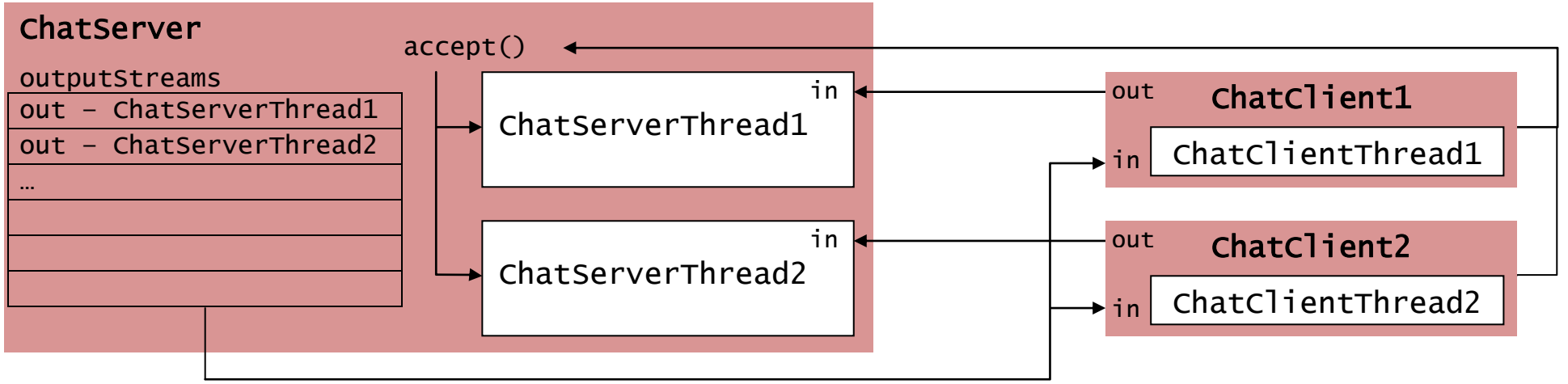
ACHTUNG: Würde die Klasse `Bankaccount` noch eine nicht synchronisierte Methode `transfer()` enthalten, so könnte diese auch während einer aktiven Monitorsperre auf das Objekt gestartet werden!!!

Bemerkungen

- Seit *Java 5* existiert Schnittstelle `Lock`, mit der sich ein kritischer Block mit `lock()` und `unlock()` markieren lässt
- `ReentrantLock` ist die wichtigste Implementierung dafür
- Birgt mehr Möglichkeiten z.B. kann festgelegt werden, dass der am längsten wartende Thread weiterarbeiten darf

Beispiel: ChatServer und ChatClient

Quelle: http://www.vorlesungen.uos.de/informatik/b06/



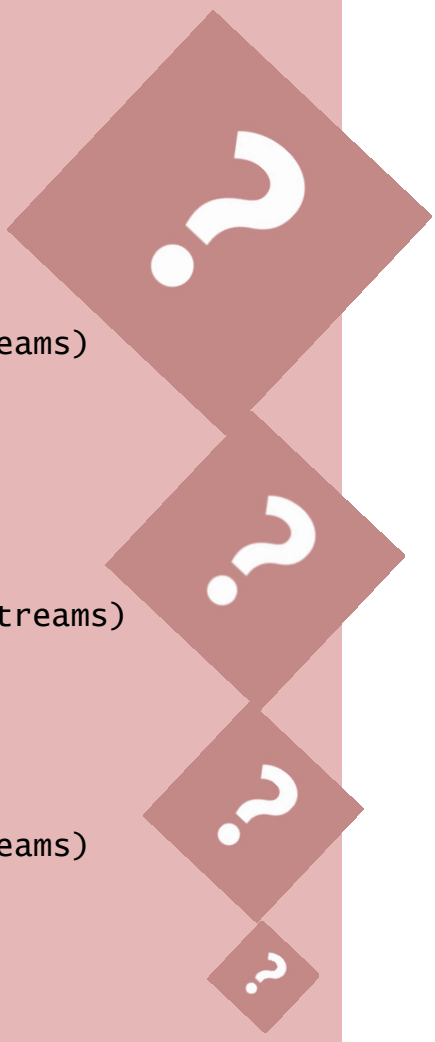
- `ChatServer` legt bei `accept()` `ChatServerThread` an welcher Kommunikation mit Client übernimmt
- Beendet Client die Kommunikation stirbt dieser Thread
- `ChatServer` enthält statische Variable `outputStreams` (`ArrayList`) welche alle OutputStreams der Threads verwaltet
- Meldet sich neuer Client an, wird sein OutputStream in `ArrayList` aufgenommen
- Erhält Thread vom Client Nachricht, wird diese über alle OutputStreams an alle Clients gesendet
- `ChatClient` baut Verbindung zum Server auf
- `ChatClient` legt `ChatClientThread` an, der Nachrichten die Server an InputStream schickt, in Konsole ausgibt
- `ChatClient` liest Nachricht von Konsole und schreibt diese in den OutputStream
- Wird in Konsole `[Strg]+Z` gedrückt wird Socket geschlossen, und Thread stirbt

ChatServer

```
public class ChatServer
{
  public static final int PORT = 65535;
  protected static ArrayList<PrintStream> outputStreams =
    new ArrayList();
  public static void main(String[] args) {
    ServerSocket server = null;
    try {
      server = new ServerSocket(PORT);
      System.out.println("Chat server started");
      while (true) {
        Socket client = server.accept();
        try {
          new ChatServerThread(client).start();
        } catch (IOException e) {
          System.out.println(e.getMessage());
        }
      }
    } catch (IOException e) {
      System.out.println(e.getMessage());
    } finally {
      try { server.close(); } catch (Exception e1) { ; }
    }
  }
}
```

ChatServerThread

```
public class ChatServerThread extends Thread
{
  private Socket client = null;
  private BufferedReader in = null;
  private PrintStream out = null;

  public ChatServerThread(Socket client) throws IOException {
    (Initialisierung von BufferedReader und PrintStream siehe hinten)
    …
  }

  @Override
  public void run() {
    try {
      ChatServer.outputStreams.add(out);

      String name = in.readLine();
      System.out.println(name + " signed in");
      for (PrintStream outs: ChatServer.outputStreams)
        outs.println(name + " signed in");

      while (true) {
        String line = in.readLine();
        if (line == null)
          break;
        for (PrintStream outs: ChatServer.outputStreams)
          outs.println(name + ": " + line);
      }

      ChatServer.outputStreams.remove(out);
      System.out.println(name + " signed out");
      for (PrintStream outs: ChatServer.outputStreams)
        outs.println(name + " signed out");
    } catch (IOException e) {
      System.out.println(e.getMessage());
      if (out != null)
        ChatServer.outputStreams.remove(out);
    } finally {
      try { client.close(); } catch (Exception e1) { ; }
    }
  }
}
```



- `in.readln()` liefert `null`, falls Client Verbindung abbricht
- Zu jedem Zeitpunkt darf nur ein Thread `ArrayList` verwenden deshalb muss **synchronized** verwendet werden, aber wie ? ? ?

ChatClient

```
public class ChatClient
{
  public static final int PORT = 65535;

  public static void main(String[] args) {
    Socket client = null;
    try {
      client = new Socket(args[1], PORT);

      BufferedReader in = new BufferedReader(
        new InputStreamReader(client.getInputStream()));
      PrintStream out = new PrintStream(client.getOutputStream());

      BufferedReader consoleIn =
        new BufferedReader(new InputStreamReader(System.in));

      // sending the name of the client to the server
      out.println(args[0]);

      new ChatClientThread(in).start();

      while (true) {
        String line = consoleIn.readLine();
        if (line == null)
          // pressed [Ctrl]+Z to sign out
          break;
        out.println(line);
      }
    } catch (IOException e) {
      System.out.println(e.getMessage());
    } finally {
      try { client.close(); } catch (Exception e1) { ; }
    }
  }
}
```

- `BufferedReader` und `PrintStream` erlauben das einfache Empfangen und Senden von Strings über eine Socketverbindung
- Schließen des Sockets führt dazu, dass Server bei `readln()` ein `null` erhält, seine Schleife und sein Thread wird beendet
- `ChatClientThread` erhält vom Server in diesem Fall ein `SocketException` und wird ebenfalls beendet (siehe hinten)

ChatClientThread

```
public class ChatClientThread extends Thread
{
  private BufferedReader in = null;
  public ChatClientThread(BufferedReader in) {
    this.in = in;
  }
  @Override
  public void run() {
    try {
      while (true) {
        String line = in.readLine();
        System.out.println(line);
      }
    } catch (SocketException e) {
      System.out.println("Ignore exception");
    } catch (IOException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

- Wird [Strg]+Z gedrückt oder ChatServer antwortet nicht weil er ausgeschaltet wurde, so wird Socket geschlossen und SocketException tritt auf
- In diesem Fall wird ChatClientThread beendet
- Exception kann in diesem Fall ignoriert werden