Objektorientierte und Formale Programmierung
-- Java Grundlagen --

10. Threads und Sockets

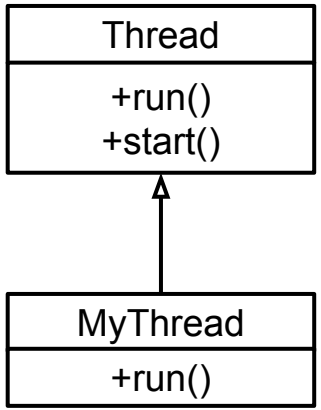
10.1. Threads

- Unsere bisherigen Programme arbeiteten rein sequentiell, die Anweisungen wurden eine nach der anderen ausgeführt
- Manchmal sollte ein Programm aber auch nebenläufig arbeiten können, d.h. mehrere Dinge (scheinbar) gleichzeitig tun, z.B.:
  - Ausgabe von Bild und Ton einer Multimedia-Anwendung
  - gleichzeitige Darstellung mehrerer Animationen
  - Bearbeitung längerer Aufgaben (z.B. Drucken) im ”Hintergrund“, während mit dem Programm weitergearbeitet wird
- Ein Thread ist eine Aktivität (d.h. Ausführung von Programmcode), die nebenläufig zu anderen Aktivitäten ausgeführt wird. Alle Threads einer Programmausführung arbeiten dabei auf denselben Daten

10.1. Threads

Threads in Java

- Ein Java-Programm startet immer mit genau einem Thread (main-Thread), der die Methode **`main()`** abarbeitet
- Für weitere Threads steht die Klasse **`Thread`** zur Verfügung, von dieser Klasse muß eine Unterklasse definiert werden
- Die wichtigste Operationen von Thread sind:
  - **`void run()`**: wird beim Start des Threads ausgeführt
    - muß in der Unterklasse überschrieben werden mit dem Code, den der Thread ausführen soll
    - der Thread endet, wenn **`run()`** zurückkehrt
  - **`void start()`**: startet den Thread
    - **`start()`** kehrt sofort zum Aufrufer zurück
    - der Thread führt nebenläufig seine **`run()`**-Methode aus



10.1. Threads

Beispiel WorkerThread (🌶️) : Berechnung im Hintergrund

```
class WorkerThread {
  public static void main(String[] s) {
    // Erstellen Sie ein Array von Threads, die gleichzeitig ausgeführt werden:
    Thread t = new WorkThread();  // Erzeuge ein neues Thread-Objekt
    t.start();  // Führe die run-Methode des Objekts in einem neuen Thread aus
  }
}

class WorkThread extends Thread {
  public void run() {  // wird nebenläufig zum Aufrufer ausgeführt
    System.out.println("Working ...\n");
    double v = 1.000000001;
    for (double i=0; i<5000000000.0; i++) { v *= v; }  // komplexe Berechnung
    System.out.println("Done!\n");
  }
}
```

10.1. Threads
Synchronisation von Threads

- Eine Hintergrund-Berechnung wie im Beispiel ist nur möglich, wenn das Ergebnis nicht zum Weiterarbeiten benötigt wird
- Andernfalls kann mit der Methode **`join()`** auf das Ende des Threads gewartet werden, wenn das Ergebnis gebraucht wird
  Dieses kann / muß in Attributen des Thread-Objekts gespeichert werden
- In vielen Fällen ist auch eine weitergehende Synchronisation der Threads erforderlich:
  - wechselseitiger Ausschluß von Methoden:
    verhindert gleichzeitige Ausführung durch mehrere Threads
  - Warten auf Ereignisse, die andere Threads auslösen

10.1. Threads

Beispiel WorkerThreadJoin (🌶🌶)

```
class WorkerThreadJoin {
  public static void main(String[] s) {
    // Erstellen Sie ein Array von Threads, die gleichzeitig ausgeführt werden
    // und benutzen Sie join() für jedes Thread:

  }
}

class WorkThread extends Thread {
  public void run() {  // wird nebenläufig zum Aufrufer ausgeführt
    System.out.println("Working ...\n");
    double v = 1.000000001;
    for (double i=0; i<5000000000.0; i++) { v *= v; }  // komplexe Berechnung
    System.out.println("Done!\n");
  }
}
```

10.1. Threads

Beispiel: Bankkonto (1/2)

```
class Konto {
  public Konto(double saldo) { this.saldo = saldo; }
  public double getSaldo() { return saldo; }
  public boolean abheben(double betrag) {
    double neuerSaldo = getSaldo() - betrag;
    boolean ok = true;
    if (neuerSaldo < 0) {
      // Bei Ueberziehung: Anfrage an Schufa (über Netzwerk)
      ok = frageSchufa(neuerSaldo);  // kann dauern ...
    }
    if (ok)
      saldo = neuerSaldo;  // Buchung durchführen
    return ok;
  }
  private double saldo = 0.0;
}
```

10.1. Threads
Beispiel: Bankkonto (2/2)

```
class Banking extends Thread {
  Konto konto; double betrag;   // Eingabedaten für den Thread
  Banking(Konto k, double b) { konto = k; betrag = b; }
  public void run() {
    konto.abheben(betrag);
    System.out.println("Kontostand: " + konto.getSaldo());
  }
}
class Bankkonto {
  public static void main(String args[]) {
    Konto konto = new Konto(10);  // Konto mit 10 EUR
    for (int i=0; i<3; i++) {  // dreimal 10 EUR abheben
      Banking t = new Banking(konto, 10); t.start();
    }
  }
}
```

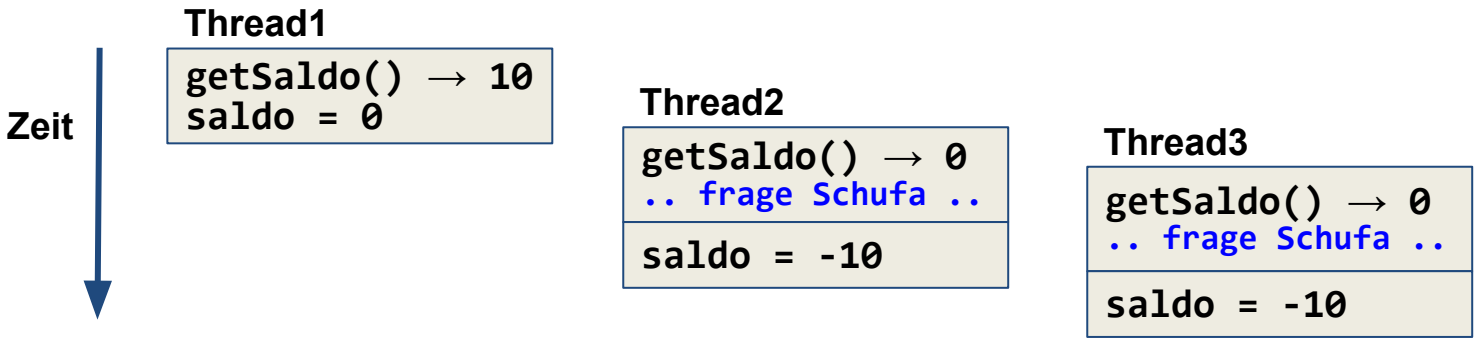
10.1. Threads

Beispiel: Bankkonto Ausgabe

```
Kontostand: 0.0
Kontostand: -10.0
Kontostand: -10.0
```

Warum hier zweimal derselbe Kontostand?

Problem: zeitliche Verzahnung in **`abheben()`**:



Lösung: wechselseitiger Ausschluß für **`abheben()`**

10.1. Threads
Synchronisation mit Monitor

- Kritischer Bereich ist ein Programmteil, der nur von einem Prozess zur gleichen Zeit durchlaufen werden darf
- Monitor dient zur Kapselung eines kritischen Bereichs und zur Synchronisation nebenläufiger Prozesse
- Sperre („Lock“)
  - Sperre wird beim Betreten des Monitors gesetzt und beim Verlassen zurückgenommen
  - Sperre beim Eintritt in den Monitor von anderem Prozess gesetzt, so muss der aktuelle Prozess warten
  - Freigabe der Sperre beim Verlassen des Monitors

10.1. Threads
Synchronisation mit Monitor

- Nutzung des Monitors mit Hilfe des Schlüsselworts Synchronized
  - Schutz einer kompletten Methode durch Sperre mit **`this`**-Pointer
  - Schutz eines Codeabschnitts durch Angabe einer Objektvariable

```
public synchronized void method() {
   // anweisungen
}
```

```
synchronized(einObjekt) {
   // anweisungen
}
```

```
public void method() {
  synchronized(this) {
        // anweisungen
  }
}
```

10.1. Threads

Beispiel Monitor: Hochzählen eines gemeinsamen Zählers

```
public class Listing extends Thread {
  static int cnt = 0;
  public static void main(String[] args) {
    Thread t1 = new Listing();
    Thread t2 = new Listing();
    t1.start();
    t2.start();
  }
  public void run() {
    while (true) System.out.print(cnt++" ");
  }
}
```

Mögliche Ausgabe: 1 2 3 4 5 7 8 9 10 6 11 12 …

Operation **`System.out.print(cnt++);`** ist nicht atomar

Unterbrechung während der Ausführung möglich

10.1. Threads

Beispiel Monitor: Hochzählen eines gemeinsamen Zählers

```
public class SyncListing extends Thread {
  static int cnt = 0;
  public static void main(String[] args) {
    Thread t1 = new SyncListing();
    Thread t2 = new SyncListing();
    t1.start();
    t2.start();
  }
  public void run() {
    while (true) {
      synchronized (getClass()) {      // getClass() gibt die Laufzeitklasse
        System.out.print(cnt+++" ");   // dieses Objekts zurück.
      }
    }
  }
}
```

10.1. Threads

Anwendung von synchronized auf eine Methode

- Zugriff auf ein Objekt selbst wird synchronisiert
- Mehr als ein Thread wird das Objekt zur gleichen Zeit verwenden
- Beispiel „Zählerobjekt“:
  - Kapselung des Zählers
  - Auf Anforderung aktuellen Zählerstand liefern und internen Zähler inkrementieren

10.1. Threads
Beispiel (1/3)

```
class Counter {
  int cnt;
  public Counter(int cnt) {  this.cnt = cnt; }
  public int nextNumber() {
    int ret = cnt;
    // zeitaufwändige Berechnung um langwierige Operation zu simulieren:
    double x = 1.0, y, z;
    for (int i= 0; i < 9000000; ++i) {
      x = Math.sin((x*i%35)*1.13);
      y = Math.log(x+10.0);
      z = Math.sqrt(x+y);
    }
    cnt++;
    return ret;
  }
}
```

10.1. Threads
Beispiel (2/3)

```
public class Listing extends Thread {
  private String name; private Counter counter;
  public Listing(String name, Counter counter) {
    this.name = name; this.counter = counter;
  }
  public static void main(String[] args) {
    Thread[] t = new Thread[5];
    Counter cnt = new Counter(10);
    for (int i = 0; i < 5; ++i) {
      t[i] = new Listing("Thread-"+i, cnt); t[i].start();
    }
  }
  public void run() {
    while (true)
      System.out.println(counter.nextNumber()+" for "+name);
  }
}
```

10.1. Threads
Beispiel (3/3)

- Ergebnis: doppelte Zahlenwerte:
- Markierung der Methode **`nextNumber()`** als synchronized macht diese zu einem Monitor.
  Code in der Methode wird als atomares Programmfragment behandelt. Unterbrechung des kritischen Abschnitts durch anderen Thread ist nicht möglich.

```
public synchronized int nextNumber() { // ...
```

```
10 for Thread-2
11 for Thread-4
10 for Thread-0
10 for Thread-1
11 for Thread-2
11 for Thread-3
12 for Thread-4
13 for Thread-0
14 for Thread-1
15 for Thread-2
16 for Thread-3
...
```

10.1. Threads
`wait` und `notify`

- **`wait`** und **`notify`** sind Synchronisationsprimitive der Klasse **`Object`**
  Das Objekt besitzt Warteliste von Threads, die unterbrochen wurden und auf ein Ereignis warten, um fortgesetzt zu werden. **`wait`** und **`notify`** dürfen nur innerhalb eines synchronized-Blocks aufgerufen werden
- Aufruf von **`wait`**
  - Nimmt bereits gewährten Sperren zurück
  - Stellt den Prozess, der den Aufruf von wait verursachte, in die Warteliste des Objekts
- Aufruf von **`notify`**
  - entfernt einen (beliebigen) Prozess aus der Warteliste des Objekts
  - Stellt die aufgehobenen Sperren wieder her

10.1. Threads

Beispiel Producer/Consumer für Fließkommazahlen (1/3)

```
class Producer extends Thread {
  private Vector v;
  public Producer(Vector v) { this.v = v; }
  public void run() {
    String s;
    while (true) {
      synchronized (v) {
        s = "Wert "+Math.random();
        v.addElement(s);
        System.out.println("Produzent erzeugte "+s);
        v.notify();
      }
      try {
        Thread.sleep((int)(100*Math.random()));
      } catch (InterruptedException e) {  }
    }
  }
}
```

10.1. Threads
Beispiel Producer/Consumer für Fließkommazahlen (2/3)

```
class Consumer extends Thread {
  private Vector v;
  public Consumer(Vector v) { this.v = v; }
  public void run() {
    while (true) {
      synchronized (v) {
        if (v.size() < 1)
          try { v.wait(); } catch (InterruptedException e) {  }
        System.out.print("Konsument fand "+(String)v.elementAt(0));
        v.removeElementAt(0);
        System.out.println(" (verbleiben: "+v.size()+")");
      }
      try {
        Thread.sleep((int)(100*Math.random()));
      } catch (InterruptedException e) {  }
    }
  }
}
```

10.1. Threads

Beispiel Producer/Consumer für Fließkommazahlen (3/3)

```
public class ProdConListing {
  public static void main(String[] args) {
    Vector v = new Vector();
    Producer p = new Producer(v);
    Consumer c = new Consumer(v);
    p.start();
    c.start();
  }
}
```

Beispielausgabe

```
Produzent erzeugte Wert 0.6548096532111007
Konsument fand Wert 0.6548096532111007 (verbleiben: 0)
Produzent erzeugte Wert 0.6965546173953919
Produzent erzeugte Wert 0.6990053250441516
Produzent erzeugte Wert 0.9874467815778902
Konsument fand Wert 0.6965546173953919 (verbleiben: 2)
Produzent erzeugte Wert 0.019655027417308846
Konsument fand Wert 0.6990053250441516 (verbleiben: 2)
```

10.2. Sockets

Adressierung

- Zur Adressierung von Rechnern im Netz wird die Klasse **`InetAddress`** des Pakets **`java.net`** verwendet
- Ein **`InetAddress`**-Objekt enthält sowohl eine IP-Adresse als auch den symbolischen Namen des jeweiligen Rechners
- Die beiden Bestandteile können mit den Methoden **`getHostName`** und **`getHostAddress`** abgefragt werden.
- Mit Hilfe von **`getAddress`** kann die IP-Adresse auch direkt als **`byte`**-Array mit vier Elementen beschafft werden
  - **`String getHostName()`**
  - **`String getHostAddress()`**
  - **`byte[] getAddress()`**

10.2. Sockets
Adressierung

- Statische Methoden zum Erzeugen eines InetAddress-Objekts:
  - `public static InetAddress getByName(String host) throws UnknownHostException`
  - `public static InetAddress getLocalHost() throws UnknownHostException`
- **`getByName`** erwartet einen String mit der IP-Adresse oder dem Namen des Hosts als Argument
- **`getLocalHost`** liefert ein InetAddress-Objekt für den eigenen Rechner
- **`UnknownHostException`** wenn die Adresse nicht ermittelt werden kann (z.B. kein DNS-Server)

10.2. Sockets
Beispiel

```
import java.net.*;
public class Lookup {
  public static void main(String[] args) {
    if (args.length != 1) {
      System.err.println("Usage: java Lookup <host>");
      System.exit(1);
    }
    try {  // Get requested address
      InetAddress addr = InetAddress.getByName(args[0]);
      System.out.println(addr.getHostName());
      System.out.println(addr.getHostAddress());
    }
    catch (UnknownHostException e) {
      System.err.println(e.toString());
      System.exit(1);
    }
  }
}
```

10.2. Sockets

Aufbau einer einfachen Socket-Verbindung

- Socket bezeichnet eine streambasierte Programmierschnittstelle zur Kommunikation zweier Rechner in einem TCP/IP-Netz
- Das Übertragen von Daten über eine Socket-Verbindung ähnelt dem Zugriff auf eine Datei:
  - Aufbau der Verbindung
  - Daten gelesen und/oder geschrieben.
  - Abbauen der Verbindung
- Erzeugen eines Sockets:
  - `public Socket(String host, int port) throws UnknownHostException, IOException`
  - `public Socket(InetAddress address, int port) throws IOException`

10.2. Sockets

Aufbau einer einfachen Socket-Verbindung

- Nachdem die Socket-Verbindung erfolgreich aufgebaut wurde, kann mit den beiden Methoden getInputStream und getOutputStream je ein Stream zum Empfangen und Versenden von Daten beschafft werden:
  - `public InputStream getInputStream() throws IOException`
  - `public OutputStream getOutputStream() throws IOException`
- Streams können direkt verwendet werden oder mit `Filterstreams` in bequemer zu verwendenden Streamtyp geschachtelt werden.
- Nach Ende der Kommunikation werden Eingabe- und Ausgabestreams als auch der Socket selbst mit close geschlossen

10.2. Sockets

Beispiel einer Socket-Verbindung

Verbindung zum DayTime-Service auf Port 13:

```
public class DayTime {
  public static void main(String[] args) {
    String hostname = "time.nist.gov";
    try {
      Socket sock = new Socket(hostname, 13);
      InputStream in = sock.getInputStream();
      int len;
      byte[] b = new byte[100];
      while ((len = in.read(b)) != -1) {  // Lese Zeit als Serie von Bytes von
        System.out.write(b, 0, len);      // time.nist.gov, Port 13
      }
      in.close();
      sock.close();
    } catch (IOException e) { System.err.println(e.toString()); }
  }
}
```

10.2. Sockets

Beispiel einer Socket-Verbindung

Echo / Lesen und Schreiben von Daten (1/3)

```
class EchoClient {
  public static void main(String[] args) {
    if (args.length != 1) {
      System.err.println("Usage: java EchoClient <host>");
      System.exit(1);
    }
    try {
      Socket sock = new Socket(args[0], 7);
      InputStream in = sock.getInputStream();
      OutputStream out = sock.getOutputStream();
      sock.setSoTimeout(300);  // Timeout setzen
      OutputThread th = new OutputThread(in);  // Ausgabethread erzeugen
      th.start();
```

10.2. Sockets

Beispiel einer Socket-Verbindung

Echo / Lesen und Schreiben von Daten (2/3)

```
// Schleife für Benutzereingaben
BufferedReader conin = new BufferedReader(
                              new InputStreamReader(System.in) );
String line = "";
while (true) {
  line = conin.readLine();     // Eingabezeile lesen
  if (line.equalsIgnoreCase("QUIT"))
    break;
  out.write(line.getBytes());  // Eingabezeile an ECHO-Server
  out.write('\r');
  out.write('\n');
  th.yield();  // Ausgabe abwarten
}
```

10.2. Sockets

Beispiel einer Socket-Verbindung

Echo / Lesen und Schreiben von Daten (3/3)

```
      // Programm beenden
      System.out.println("terminating output thread...");
      th.requestStop();
      th.yield();
      try { Thread.sleep(1000); } catch (InterruptedException e) { }
      in.close();
      out.close();
      sock.close();
    } catch (IOException e) {
      System.err.println(e.toString());
      System.exit(1);
    }

  }  // end main
}
```

10.2. Sockets

Beispiel einer Socket-Verbindung

OutputThread (1/2)

```
class OutputThread extends Thread {
  InputStream in;
  boolean stoprequested;

  public OutputThread(InputStream in) {
     super();
     this.in = in;
     stoprequested = false;
  }

  public synchronized void requestStop() {
     stoprequested = true;
  }
```

10.2. Sockets

Beispiel einer Socket-Verbindung

OutputThread (2/2)

```
  public void run() {
    int len;
    byte[] b = new byte[100];
    try {
      while (!stoprequested) {
        try {
          if ((len = in.read(b)) == -1) break;
          System.out.write(b, 0, len);
        } catch (InterruptedIOException e) {
          // nochmal versuchen
        }
      }
    } catch (IOException e) {
      System.err.println("OutputThread: " + e.toString());
    }
  }
}
```

10.2. Sockets
Server Sockets

- Klasse **`ServerSocket`** bietet Funktionen um auf einen eingehenden Verbindungswunsch zu warten und nach erfolgtem Verbindungsaufbau einen Socket zur Kommunikation mit dem Client zurückzugeben (`import java.net.serversocket;`)
  - `public ServerSocket(int port) throws IOException`
  - `public Socket accept() throws IOException`
- Konstruktor erzeugt einen **`ServerSocket`** für einen bestimmten Port, also einen bestimmten Typ von Server-Anwendung
- Anschließend wird die Methode accept aufgerufen, um auf einen eingehenden Verbindungswunsch zu warten
- **`accept`** blockiert so lange, bis sich ein Client bei der Server-Anwendung anmeldet
- Ist der Verbindungsaufbau erfolgreich, liefert accept ein Socket-Objekt, das wie bei einer Client-Anwendung zur Kommunikation mit der Gegenseite verwendet werden kann

10.2. Sockets

Beispiel Server Socket: Echo Server

```
class SimpleEchoServer {
  public static void main(String[] args) {
    try {
      ServerSocket echod = new ServerSocket(7);
      Socket socket = echod.accept();  // Warte auf Verbindung (Port 7)
      InputStream in = socket.getInputStream();
      OutputStream out = socket.getOutputStream();
      int c;
      while ((c = in.read()) != -1) {
        out.write((char)c);
        System.out.print((char)c);
      }
      socket.close();   // Verbindung beenden
      echod.close();
    } catch (IOException e) { /*...*/ }
  }
}
```

10.2. Sockets

Verbindungen zu mehreren Clients

Für jeden Client soll ein eigener Thread angelegt werden

- Hauptprogramm erzeugt **`ServerSocket`**
- Warten auf Verbindungswunsch in einer Schleife mit **`accept`**
- Nach dem Verbindungsaufbau erfolgt die weitere Bearbeitung in einem neuen Thread mit dem Verbindungs-Socket als Argument
  - Thread erledigt die gesamte Kommunikation mit dem Client
  - Beendet der Client die Verbindung, wird auch der zugehörige Thread beendet

10.2. Sockets

Beispiel Server für mehrere Clients (1/3)

```
class EchoServer {
  public static void main(String[] args) {
    int cnt = 0;
    try {
      System.out.println("Warte auf Verbindungen auf Port 7...");
      ServerSocket echod = new ServerSocket(7);
      while (true) {  // accept-Aufruf wird in eine Schleife verpackt und für
         Socket socket = echod.accept(); // jeden einkommenden Client-Request
         (new EchoClientThread(++cnt, socket)).start();  // wird ein eigener
      }                                                  // Thread EchoClientThread erzeugt
    } catch (IOException e) {
      System.err.println(e.toString());
      System.exit(1);
    }
  }
}
```

10.2. Sockets
Beispiel Server für mehrere Clients (2/3)

```
class EchoClientThread extends Thread {
  private int name;
  private Socket socket;

  public EchoClientThread(int name, Socket socket) {
    this.name = name;
    this.socket = socket;
  }

  //...
```

10.2. Sockets

Beispiel Server für mehrere Clients (3/3)

```
  public void run() {
    String msg = "EchoServer: Verbindung " + name;
    System.out.println(msg + " hergestellt");
    try {
      InputStream in = socket.getInputStream();
      OutputStream out = socket.getOutputStream();
      out.write((msg + "\r\n").getBytes());
      int c;
      while ((c = in.read()) != -1) {
        out.write((char)c);
        System.out.print((char)c);
      }
      System.out.println("Verbindung " + name + " wird beendet");
      socket.close();
    }
    catch (IOException e) { System.err.println(e.toString()); }
  }
}
```