

Wiederholung: Objektorientierte Vererbung

Grundlegende Aspekte objektorientierter Vererbung werden wiederholt an Hand von Folien aus

**U. Kastens: Vorlesung SWE, 1998/99 (aktualisiert),
<http://ag-kastens.upb.de/lehre/material/swei>**

- 18 - 20: Klassen und Objekte
- 78 - 82b: Vererbung

Folgender Abschnitt wird ebenso zur Wiederholung angeraten:

- 73 - 77b: Objektorientierte Abstaktion, Schnittstellen

Vorlesung Software-Entwicklung II SS 2004 / Folie 130x

Ziele:

Hinweis auf Folien-Material

in der Vorlesung:

- Klassen und Objekte
- Vererbung
- Objektorientierte Abstaktion, Schnittstellen

nachlesen:

Text

9. Parallele Prozesse Grundbegriffe (1)

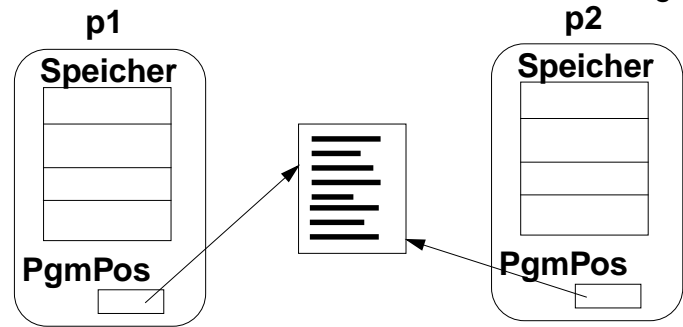
Prozess:

Ausführung eines sequentiellen Programmstückes in dem zugeordneten Speicher (Adressraum).

Zustand des Prozesses:

Speicherinhalt und Programmposition

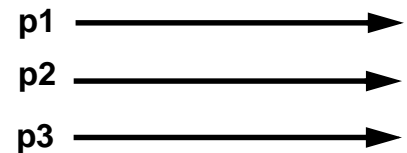
Zustände zweier Prozesse zu demselben Programm



sequentieller Prozess: Die Operationen werden eine nach der anderen ausgeführt.

Parallele Prozesse: mehrere Prozesse, die **gleichzeitig auf mehreren Prozessoren** ausgeführt werden;

- keine Annahme über relative Geschwindigkeit
- **unabhängige** Prozesse oder
- **kommunizierende** Prozesse:
p1 liefert Daten für p2 oder
p1 wartet bis eine Bedingung erfüllt ist



Vorlesung Software-Entwicklung II SS 2004 / Folie 131

Ziele:

Grundbegriffe Paralleler Prozesse verstehen

in der Vorlesung:

- Ampel-Beispiel zeigen,
- Begriffe erklären: Prozess, Adressraum, sequentiell, parallel

nachlesen:

Judy Bishop: Java lernen, 2.Aufl., Abschnitt 13.1

Verständnisfragen:

Geben Sie Beispiele für

- sequentielle Abläufe,
- parallele Prozesse

aus dem Alltag und in Rechnern.

Grundbegriffe (2)

verzahnte Prozesse: mehrere Prozesse, die stückweise **abwechselnd auf einem Prozessor** ausgeführt werden

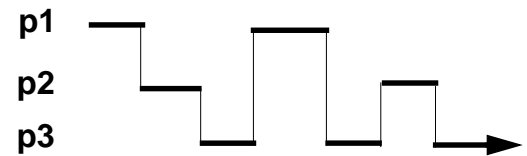
Prozessumschaltung durch den Scheduler des Betriebssystems, der JVM oder durch die Prozesse selbst.

Ziele:

- Prozessor auslasten, blockierte Prozesse nicht bearbeiten
- Gleichzeitigkeit simulieren

nebenläufige Prozesse:

Prozesse, die parallel oder verzahnt ausgeführt werden können



Vorlesung Software-Entwicklung II SS 2004 / Folie 131a

Ziele:

Grundbegriffe Paralleler Prozesse verstehen

in der Vorlesung:

Begriffe erklären:

- verzahnt (interleaving),
- nebenläufig,
- Prozessumschaltung

nachlesen:

Judy Bishop: Java lernen, 2.Aufl., Abschnitt 13.1

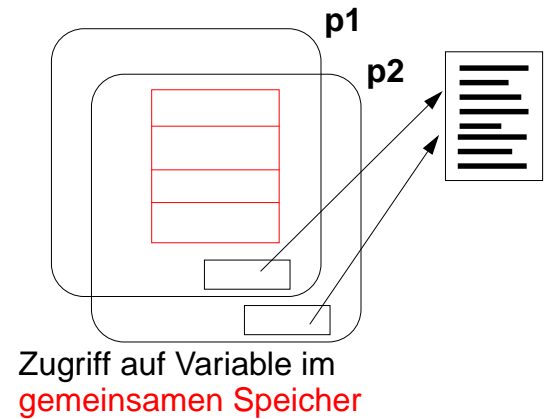
Verständnisfragen:

- In welchen Situationen veranlasst die Prozessverwaltung eine Prozessumschaltung?

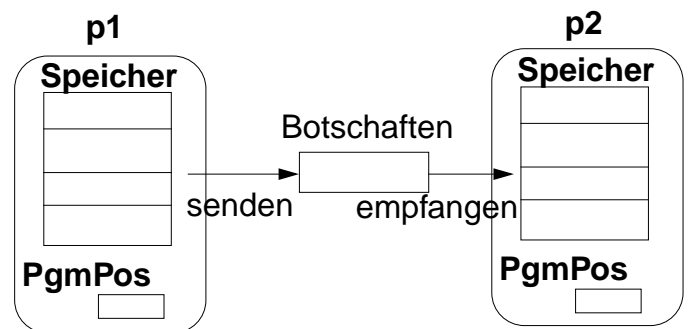
Grundbegriffe (3)

Threads („Fäden“ der Programmausführung):
Prozesse, die parallel oder verzahnt in
gemeinsamem Speicher ablaufen.

Die **Prozessumschaltung** ist besonders einfach
und **schnell**,
daher auch **leichtgewichtige Prozesse**



Im Gegensatz zu
Prozessen im verteilten Speicher auf
verschiedenen Rechnern,
kommunizieren über Botschaften, statt
über Variable im Speicher



Vorlesung Software-Entwicklung II SS 2004 / Folie 131b

Ziele:

Grundbegriffe Paralleler Prozesse verstehen

in der Vorlesung:

Begriffe erklären:

- Thread,
- gemeinsamer Speicher,
- verteilter Speicher,
- Botschaft (message)

nachlesen:

Judy Bishop: Java lernen, 2.Aufl., Abschnitt 13.1

Anwendungen Paralleler Prozesse

- **Benutzungsoberflächen:**
Die Ereignisse werden von einem speziellen Systemprozess weitergegeben.
Aufwendige Berechnungen sollten nebenläufig programmiert werden, damit die Bedienung der Oberfläche nicht blockiert wird.
- **Simulation** realer Abläufe:
z. B. Produktion in einer Fabrik
- **Animation:**
Veranschaulichung von Abläufen, Algorithmen; Spiele
- **Steuerung** von Geräten:
Prozesse im Rechner überwachen und steuern externe Geräte, z. B. Montage-Roboter
- **Leistungssteigerung** durch Parallelrechner:
mehrere Prozesse bearbeiten gemeinsam die gleiche Aufgabe,
z. B. paralleles Sortieren großer Datenmengen.

Vorlesung Software-Entwicklung II SS 2004 / Folie 132

Ziele:

Vielfalt der Parallelverarbeitung erkennen

in der Vorlesung:

Es werden Beispiele zu den Anwendungsarten erläutert und einige Prozesse in Applets vorgeführt.

nachlesen:

Judy Bishop: Java lernen, 2.Aufl., Abschnitt 13.1

Verständnisfragen:

- Geben Sie weitere Beispiele von parallelen Prozessen in Anwendungen.

Java Threads erzeugen (1)

Prozesse, die nebenläufig, gemeinsam im Speicher des Programms bzw. Applets ablaufen.

Es gibt 2 Techniken Threads zu erzeugen. Erste Technik:

Eine Benutzerklasse implementiert das Interface **Runnable**:

```
class Aufgabe implements Runnable
{ ...
  public void run ()                vom Interface geforderte Methode run
  {...}                             das als Prozess auszuführende Programmstück
  Aufgabe (...) {...}               Konstruktor
}
```

Der Prozess wird als Objekt der vordefinierten Klasse Thread erzeugt:

```
Thread auftrag = new Thread (new Aufgabe (...));
```

Erst folgender Aufruf startet dann den Prozess:

```
auftrag.start();    Der neue Prozess beginnt, neben dem hier aktiven zu laufen.
```

Diese Technik (das Interface **Runnable** implementieren) sollte man anwenden, wenn der **abgespaltene Prozess nicht weiter beeinflusst** werden muss; also einen Auftrag erledigt (Methode **run**) und dann terminiert.

Vorlesung Software-Entwicklung II SS 2004 / Folie 133

Ziele:

Deklaration von Prozessklassen verstehen.

in der Vorlesung:

3 Programmierschritte:

- Klasse mit der Methode run deklarieren
- Prozessobjekte erzeugen
- Ausführung der Prozessobjekte starten

Falls in der Benutzerklasse weitere Objektmethoden benötigt würden, wären sie schlecht zugänglich. Dann sollte man die andere Variante verwenden.

nachlesen:

Judy Bishop: Java lernen, 2.Aufl., Abschnitt 13.2

Verständnisfragen:

- Die Klasse Thread hat Klassen- und Objektmethoden. Welche können in der run-Methode auf welche Weise aufgerufen werden?

Java Threads erzeugen (2)

Zweite Technik:

Die Benutzerklasse wird als Unterklasse der vordefinierten Klasse `Thread` definiert:

```
class DigiClock extends Thread
{ ...
  public void run ()           überschreibt die Thread-Methode run
  {...}                       das als Prozess auszuführende Programmstück
  DigiClock (...) {...}      Konstruktor
}
```

Der Prozess wird als Objekt der Benutzerklasse erzeugt (es ist auch ein `Thread`-Objekt):

```
Thread clock = new DigiClock (...);
```

Erst folgender Aufruf startet dann den Prozess:

```
clock.start();           der neue Prozess beginnt neben dem hier aktiven zu laufen
```

Diese Technik (Unterklasse von `Thread`) sollte man anwenden, wenn der abgespaltene Prozess weiter beeinflusst werden soll; also weitere Methoden der Benutzerklasse definiert und von aussen aufgerufen werden, z. B. zum vorübergehenden Anhalten oder endgültigem Terminieren.

Vorlesung Software-Entwicklung II SS 2004 / Folie 134

Ziele:

Deklaration von Prozesklassen verstehen.

in der Vorlesung:

3 Programmierschritte:

- Klasse mit der Methode `run` deklarieren
- Prozessobjekte erzeugen
- Ausführung der Prozeßobjekte starten

Gegenüberstellung zur Variante mit Interface `Runnable`.

nachlesen:

Judy Bishop: Java lernen, 2.Aufl., Abschnitt 13.2

Verständnisfragen:

- Die Klasse `Thread` hat Klassen- und Objektmethoden. Welche können in der `run`-Methode auf welche Weise aufgerufen werden?

10. Unabhängige parallele Prozesse

Beispiel: Digitale Uhr als Prozess im Applet (1)

Der Prozess soll in jeder Sekunde Datum und Uhrzeit als Text aktualisiert anzeigen.

Applet

Tue Mar 30 18:18:47 CEST 1999

Applet started.

```
class DigiClock extends Thread
{ public void run ()
  { while (running)
    { line.setText(new Date().toString());          Datum schreiben
      try { sleep (1000); } catch (Exception ex) {}    Pause
    }
  }
}
// Methode, die den Prozeß von außen terminiert:
public void stopIt () { running = false; }
private boolean running = true;                    Zustandsvariable

public DigiClock (Label t) {line = t;} Label zum Beschreiben übergeben
private Label line;
}
```

Prozess als **Unterklasse** von **Thread**, weil er

- durch Aufruf von `stopIt` terminiert und
- durch weitere **Thread**-Methoden unterbrochen werden soll.

Vorlesung Software-Entwicklung II SS 2004 / Folie 135

Ziele:

Erstes vollständiges Prozessbeispiel

in der Vorlesung:

Erläuterungen zur

- Ausführung bis zur Terminierung von außen,
- `stopIt`-Methode,
- Begründung der Variante "Unterklasse von Thread".

Applet vorführen Digital Clock Process

Beispiel: Digitale Uhr als Prozess im Applet (2)

Der Prozess wird in der `init`-Methode der `Applet`-Unterklasse erzeugt:

```
public class DigiApp extends Applet
{   public void init ()
    {   Label clockText = new Label ("-----");
        add (clockText);

        clock = new DigiClock (clockText);           Prozess erzeugen
        clock.start();                               Prozess starten
    }

    public void start ()   { clock.resume(); }         Prozess fortsetzen
    public void stop ()   { clock.suspend(); }        Prozess anhalten
    public void destroy () { clock.stopIt(); }        Prozess terminieren

    private DigiClock clock;
}
```

Prozesse, die in einem Applet gestartet werden,

- sollen angehalten werden (`suspend`, `resume`), solange das Applet nicht sichtbar ist (`stop`, `start`),
- müssen terminiert werden (`stopIt`), wenn das Applet entladen wird (`destroy`).

Andernfalls belasten Sie den Rechner, obwohl sie nicht mehr sichtbar sind.

Vorlesung Software-Entwicklung II SS 2004 / Folie 136

Ziele:

Prozess aus Applet starten

in der Vorlesung:

Erläuterungen zum Starten, Anhalten, Fortsetzen und Terminieren von Prozessen aus Applets.

nachlesen:

Judy Bishop: Java lernen, 2.Aufl., Abschnitt 13.2

Übungsaufgaben:

Ändern Sie die Klassen dieses Beispiels, so daß `DigiClock` nicht Unterklasse von `Thread` ist sondern `Runnable` implementiert.

Verständnisfragen:

- Begründen Sie weshalb die gezeigte Lösung besser ist als eine in der `DigiClock` `Runnable` implementiert.

Wichtige Methoden der Klasse Thread

public void run ();

wird überschrieben mit der Methode, die die auszuführenden Anweisungen enthält

public void start ();

startet die Ausführung des Prozesses

public void suspend ();

hält den angegebenen Prozess an: `clock.suspend();`

public void resume ();

setzt den angegebenen Prozess fort: `clock.resume();`

public void join () throws InterruptedException;

der aufrufende Prozess wartet bis der angegebene Prozess terminiert ist:

```
try { auftrag.join(); } catch (Exception e){}
```

public static void sleep (long millisec) throws InterruptedException;

der aufrufende Prozess wartet mindestens die in Millisekunden angegebene Zeit:

```
try { Thread.sleep (1000); } catch (Exception e){}
```

public final void stop () throws SecurityException;

nicht benutzen! Terminiert den Prozess u. U. in einem inkonsistenten Zustand

Vorlesung Software-Entwicklung II SS 2004 / Folie 137

Ziele:

Übersicht zu Thread-Methoden

in der Vorlesung:

- Erläuterungen zu den Methoden
- Veranschaulichung durch graphische Darstellung der Abläufe der beteiligten Prozesse
- Verweise auf Beispiele

nachlesen:

Judy Bishop: Java lernen, 2.Aufl., Abschnitt 13.2

Übungsaufgaben:

Veranschaulichen Sie die Wirkung der Methoden durch graphische Darstellung der Abläufe der beteiligten Prozesse.

Verständnisfragen:

- An welchen Methodenaufrufen sind zwei, an welchen ist nur ein Prozess beteiligt?

Beispiel: Ampelsimulation (1)

Aufgabe:

- Auf einer Zeichenfläche mehrere Ampeln darstellen.
- Der Prozess jeder Ampel durchläuft die Ampelphasen und zeichnet die Lichter entsprechend.
- Vereinfachungen:
1 Ampel; keine Einstellung der Phasendauer; keine Grünanforderung für Fußgänger.

Lösung:

- Ampel-Objekte aus einer Unterklasse (`setOfLights`) von `Thread` erzeugen (Methode 2).
- Ampel-Objekte zeichnen auf gemeinsamer Zeichenfläche (`Canvas`-Objekt).
- `run`-Methode wechselt die Ampelphasen und ruft eine Methode zum Zeichnen der Lichter auf (`drawLights`).
- Eine weitere Methode (`draw`) zeichnet die unveränderlichen Teile des Ampel-Bildes.
- Die `paint`-Methode der Zeichenfläche ruft beide Zeichenmethoden der Ampel-Objekte auf (Delegation der Zeichenaufgabe).

Vorlesung Software-Entwicklung II SS 2004 / Folie 138

Ziele:

Techniken für unabhängige Prozesse an nicht-trivialer Aufgabe zeigen

in der Vorlesung:

- Aufgabe erläutern
- Vereinfachung und Erweiterbarkeit begründen
- mit dem statischen Ampel-Programm Änderungen erläutern
- 2 Zeichenmethoden begründen

nachlesen:

Judy Bishop: Java lernen, 2.Aufl., Abschnitt 13.2, Ex.13.2

Übungsaufgaben:

Vergleichen Sie die Aufgabe mit der des statischen Ampel-Applets.

Prozessklasse der Ampelsimulation

```

class SetOfLights extends Thread
{ /* Jedes Objekt dieser Klasse zeichnet eine Ampel an vorgegebener
   Position x auf ein gemeinsames Canvas-Objekt area. */

   private Canvas area;           // gemeinsame Zeichenfläche
   private int x;                 // X-Koordinate dieser Ampel
   private int light = 0;        // aktuelle Ampelphase

   public SetOfLights (Canvas area, int x)
   { this.area = area; this.x = x;}

   public void run ()
   { while (running)              // drei Ampelphasen wiederholen
     { for (light = 0; light < 3; light++)
       { drawLights ();           // neuen Zustand zeichnen
         try { sleep(500); }      // nach fester Dauer Zustand wechseln
         catch (InterruptedException e) { }
       }
     }
   }

   public void stopIt () { running = false; } // Prozess von außen anhalten
   private boolean running = true;

   public void draw (Graphics g) {...} // unveränderliche Bildteile zeichnen
   public void drawLights ()    {...}  // veränderliche Bildteile zeichnen
}

```

© 2004 bei Prof. Dr. Uwe Kastens

Vorlesung Software-Entwicklung II SS 2004 / Folie 139

Ziele:

Muster für Prozesse, die auf gemeinsamer Zeichenfläche zeichnen

in der Vorlesung:

- Canvas-Objekt übergeben statt Unterklasse von Canvas!
- Objektvariable begründen
- Methoden run und stopIt erläutern
- Der Prozeß zeichnet nur die veränderlichen Bildteile.

nachlesen:

Judy Bishop: Java lernen, 2.Aufl., Abschnitt 13.2, Ex.13.2

Übungsaufgaben:

Vergleichen Sie diese Klasse mit der Klasse `LightsCanvas` der statischen Applet-Version

Verständnisfragen:

- Wie implementieren Sie einstellbare Phasendauern?
- Wie erweitern Sie die run-Methode um die Grünanforderung für Fußgänger?

Zeichenmethoden der Ampelsimulation

```

class SetOfLights extends Thread
{
    ...
    public void draw (Graphics g)
    {
        // unveränderliche Teile des Bildes relativ zu x zeichnen und schreiben:
        g.drawOval(x-8, 10, 30, 68);           // der Ampelumriss
    }

    public void drawLights ()
    {
        // veränderliche Teile des Bildes zeichnen:
        Graphics g = area.getGraphics();//das Graphics-Objekt der Zeichenfläche

        if (light == 0) g.setColor(Color.red);           // die 4 Lichter
        else g.setColor(Color.lightGray);
        g.fillOval(x, 15, 15, 15);

        if (light == 1) g.setColor(Color.yellow);
        else g.setColor(Color.lightGray);
        g.fillOval(x, 35, 15, 15);

        if (light == 2) g.setColor(Color.green);
        else g.setColor(Color.lightGray);
        g.fillOval(x, 55, 15, 15);

        g.setColor(Color.green); g.fillOval(x, 85, 15, 15);
    }
}

```

© 2004 bei Prof. Dr. Uwe Kastens

Vorlesung Software-Entwicklung II SS 2004 / Folie 140

Ziele:

Trennung zwischen statischen und veränderlichen Bildteilen zeigen

in der Vorlesung:

Erläuterungen dazu

nachlesen:

Judy Bishop: Java lernen, 2.Aufl., Abschnitt 13.2, Ex. 13.2

Verständnisfragen:

Wie implementieren Sie die Grünanforderung für Fußgänger?

Applet-Klasse der Ampelsimulation

```

public class TrafficLight extends Applet
{ private Canvas area; // gemeinsame Zeichenfläche
  private int lightsPosition = 105; // X-Koordinate der nächsten Ampel
  private SetOfLights lights; // zunächst nur eine Ampel

  public void init ()
  { setLayout (new BorderLayout ());
    area = // das Canvas-Objekt zum Zeichnen der Ampeln
    new Canvas() // anonyme Unterklasse mit 1 Objekt:
    { public void paint (Graphics g) // Zeichnen an Ampel-Objekt delegieren
      { lights.draw (g); // zeichne statischen Teil
        lights.drawLights (); // zeichne veränderlichen Teil
      }
    };
    add (area, BorderLayout.CENTER);

    lights = new SetOfLights (area, lightsPosition); // 1 Ampel-Objekt
    area.repaint (); // erstes Zeichnen anstoßen
    lights.start (); // Prozess starten
  }

  // Prozesse des Applets anhalten, fortsetzen, beenden:
  public void stop () { lights.suspend (); }
  public void start () { lights.resume (); }
  public void destroy () { lights.stopIt (); }
}

```

© 2004 bei Prof. Dr. Uwe Kastens

Vorlesung Software-Entwicklung II SS 2004 / Folie 141

Ziele:

Ein Schema für die Hauptklasse bei zeichnenden Prozessen

in der Vorlesung:

Erläuterungen zur

- Erzeugung der gemeinsamen Zeichenfläche
- Aufteilung des Zeichnens von statischen und veränderlichen Bildteilen

nachlesen:

Judy Bishop: Java lernen, 2.Aufl., Abschnitt 13.2, Ex. 13.2

Übungsaufgaben:

Vergleichen Sie diese Klasse mit der entsprechenden Klasse der statischen Applet-Version

Verständnisfragen:

- Wie wird das Zeichnen der verschiedenen Bildteile ausgelöst?
- Wie erweitern Sie die Implementierung auf mehrere Ampeln?
- Begründen Sie die Implementierung der Methoden `start`, `stop` und `destroy`.
- Warum kann hier eine anonyme Klasse benutzt werden?

11. Monitore, Synchronisation: Gegenseitiger Ausschluss

Wenn mehrere Prozesse die **Werte gemeinsamer Variablen verändern**, kann eine ungünstige Verzahnung (oder echte Parallelausführung) zu inkonsistenten Daten führen.

Z. B. zwei Prozesse benutzen lokale Variable `tmp` und eine gemeinsame Variable `konto`:

```

p   tmp = konto;                               konto = tmp+10;
q                                     tmp = konto; konto = tmp+10;

```

Kritische Abschnitte: zusammengesetzte Operationen, die gemeinsame Variablen lesen und/oder verändern.

Prozesse müssen kritische Abschnitte unter **gegenseitigem Ausschluss (mutual exclusion)** ausführen:

D. h. **zu jedem Zeitpunkt darf höchstens ein Prozess** solch einen kritischen Abschnitt für bestimmte Variablen ausführen. Andere Prozesse, die für die gleichen Variablen mit der Ausführung eines kritischen Abschnitts beginnen wollen, müssen warten.

Vorlesung Software-Entwicklung II SS 2004 / Folie 142

Ziele:

Grundbegriff "Gegenseitiger Ausschluss" verstehen

in der Vorlesung:

Begründung, weshalb Inkonsistenzen durch Verzahnte Ausführung entstehen können.

Verständnisfragen:

- Geben Sie weitere Beispiele für kritische Abschnitte.
- Warum könnte auch das Ausgeben der Werte zweier Variable ein kritischer Abschnitt sein?

Gegenseitiger Ausschluss durch Monitore

Monitor:

Ein Modul, der Daten und Operationen darauf kapselt.

Die **kritischen Abschnitte** auf den Daten werden als **Monitor-Operationen** formuliert.

Prozesse rufen Monitor-Operationen auf, um auf die Daten zuzugreifen.

Die Monitor-Operationen werden unter gegenseitigem Ausschluss ausgeführt.

Monitore in Java:

Methoden einer Klasse, die kritische Abschnitte auf Objektvariablen implementieren, können als **synchronized** gekennzeichnet werden:

```
class Bank
{   synchronized public void abbuchen (...)   {...}
    synchronized public void überweisen (...) {...}
    ...
    private int[] konten;
}
```

Jedes Objekt der Klasse wirkt dann als **Monitor** für seine Objektvariablen:

Aufrufe von synchronized Methoden von mehreren Prozessen für dasselbe Objekt werden unter **gegenseitigem Ausschluss** ausgeführt.

Zu jedem Zeitpunkt kann höchstens 1 Prozess mit **synchronized** Methoden auf Objektvariable zugreifen.

Vorlesung Software-Entwicklung II SS 2004 / Folie 143

Ziele:

Das Konzept "Monitor" verstehen

in der Vorlesung:

Erläuterungen dazu

- Monitor als Zusammenfassung kritischer Abschnitte
- Monitor-Klassen in Java mit synchronized Methoden
- gegenseitiger Ausschluss immer für jedes Objekt individuell
- gegenseitiger Ausschluss nur für Aufrufe von synchronized Methoden untereinander

Verständnisfragen:

Geben Sie Beispiele für Monitor-Methoden, die *nicht* unter gegenseitigem Ausschluss zu laufen brauchen.

Beispiel: gegenseitiger Ausschluss durch Monitor

```

class Bank
{
    synchronized public void abbuchen (String name) // kritischer Abschnitt
    {
        int vorher = konto;
        System.out.println (name + " Konto vor : " + vorher);
        // kritische Verwendung der Objektvariablen konto:
        konto = vorher - 10;
        System.out.println (name + " Konto nach: " + konto);
    }
    private int konto = 100;
}

class Kunde extends Thread
{
    Kunde (String str, Bank b) { name = str; meineBank = b; }
    public void run ()
    {
        for (int i = 1; i <= 2; i++) meineBank.abbuchen(name);
    }
    private Bank meineBank; private String name;
}

class BankPgm
{
    public static void main (String [] args)
    {
        Bank b = new Bank(); // Objekt als Monitor für seine Objektvariable
        new Kunde("ich", b).start(); new Kunde("du ", b).start();
    }
}

```

Vorlesung Software-Entwicklung II SS 2004 / Folie 144

Ziele:

Ein erstes vollständiges Beispiel für einen Monitor.

in der Vorlesung:

- beteiligte Objekte zeigen
- verschiedene Abläufe zeigen
- Abläufe zeigen, für den Fall, daß die Methode *nicht* `synchronized` definiert wird

Übungsaufgaben:

Implementieren Sie das Programm. Entfernen Sie das `synchronized`, und fügen Sie vor der Zuweisung an die Variable `konto` einen `sleep`-Aufruf ein.

12. Bedingungssynchronisation im Monitor

Bedingungssynchronisation:

Ein Prozess wartet, bis eine Bedingung erfüllt ist - verursacht durch einen anderen Prozess;
z. B. erst dann in einen Puffer schreiben, wenn er nicht mehr voll ist.

Bedingungssynchronisation im Monitor:

Ein Prozess, der in einer kritischen Monitor-Operation auf eine Bedingung wartet, muss den Monitor freigeben, damit andere Prozesse den Zustand des Monitors ändern können.

Bedingungssynchronisation in Java:

Vordefinierte Methoden der Klasse `Object` zur Bedingungssynchronisation:
(Sie müssen aus `synchronized` Methoden heraus aufgerufen werden.)

- `wait()` **blockiert den aufrufenden Prozess und gibt den Monitor frei**
- das Objekt, dessen `synchronized` Methode er gerade ausführt.
- `notifyAll()` **weckt alle in diesem Monitor blockierten Prozesse;**
sie können weiterlaufen, sobald der Monitor frei ist.
- `notify()` weckt einen blockierten Prozess;
ist für unser Monitorschema nicht brauchbar

Nachdem ein blockierter Prozess geweckt wurde, muß er die **Bedingung, auf die er wartet, erneut prüfen** - sie könnte schon durch schnellere Prozesse wieder invalidiert sein:

```
while (avail<n) try { wait(); } catch (InterruptedException e) {}
```

Vorlesung Software-Entwicklung II SS 2004 / Folie 145

Ziele:

Das Prinzip "Bedingungssynchronisation" verstehen

in der Vorlesung:

- Erläuterungen an Beispielen für Wartebedingungen
- Erläuterung der Methoden `wait` und `notifyAll`

Beispiel: Monitor zur Vergabe von Ressourcen

Aufgabe: Eine Begrenzte Anzahl gleichartiger Ressourcen wird von einem Monitor verwaltet. Prozesse fordern einige Ressourcen an und geben sie später zurück.

```
class Monitor
{ private int avail;                // Anzahl verfügbarer Ressourcen
  Monitor (int a) { avail = a; }

  synchronized void getElem (int n, int who)    // n Elemente abgeben
  { System.out.println("Client"+who+" needs "+n+",available "+avail);

    while (avail < n)                        // Bedingung in Warteschleife prüfen
    { try { wait(); } catch (InterruptedException e) {}
      }                                        // try ... catch nötig wegen wait()
    avail -= n;
    System.out.println("Client"+who+" got "+n+", available "+avail);
  }

  synchronized void putElem (int n, int who)   // n Elemente zurücknehmen
  { avail += n;
    System.out.println("Client"+who+" put "+n+",available "+avail);
    notifyAll();                             // Alle Wartenden können die Bedingung erneut prüfen
  }
}
```

© 2004 bei Prof. Dr. Uwe Kastens

Vorlesung Software-Entwicklung II SS 2004 / Folie 146

Ziele:

Ein vollständiges Monitor-Beispiel verstehen

in der Vorlesung:

- Das Vergabe-Schema (put, get) erläutern
- Zusammenspiel von wait() und notifyAll erläutern
- Zeigen, weshalb die Schleife um den wait-Aufruf nötig ist

nachlesen:

Judy Bishop: Java lernen, 2.Aufl., Abschnitt 13.3

Übungsaufgaben:

Leser/Schreiber-Problem implementieren

Verständnisfragen:

- Warum steht der Aufruf wait() in einer Schleife?
- Warum wird notifyAll() statt notify() verwendet?

Beispiel: Prozesse und Hauptprogramm zu SWE-146

```
import java.util.Random;

class Client extends Thread
{ private Monitor mon; private Random rand;
  private int ident, rounds, max;
  Client (Monitor m, int id, int rd, int avail)
  { ident = id; rounds = rd; mon = m; max = avail;
    rand = new Random();          // Neuer Zufallszahlengenerator
  }

  public void run ()
  { while (rounds > 0)
    { int m = Math.abs(rand.nextInt()) % max + 1;
      mon.getElem (m, ident);    // m Elemente anfordern
      try { sleep (Math.abs(rand.nextInt()) % 1000 + 1);}
          catch (InterruptedException e) {}
      mon.putElem (m, ident);    // m Elemente zurückgeben
      rounds--;
    }
  }
}
```

```
class TestMonitor
{ public static void main (String[] args) {
  int avail = 20;
  Monitor mon = new Monitor (avail);
  for (int i = 0; i < 5; i++)
    new Client(mon,i, 4,avail).start();
} }
```

Vorlesung Software-Entwicklung II SS 2004 / Folie 147

Ziele:

Fortsetzung von [Folie 146](#)

in der Vorlesung:

Erläuterungen dazu

Schema: Gleichartige Ressourcen vergeben

Ein **Monitor** verwaltet eine endliche Menge von $k \geq 1$ **gleichartigen Ressourcen**.

Prozesse fordern jeweils unterschiedlich viele (n) Ressourcen an, $1 \leq n \leq k$, und geben sie nach Gebrauch wieder zurück.

Die **Wartebedingung** für das Anfordern ist „Sind n Ressourcen verfügbar?“

Zu jedem Zeitpunkt zwischen Aufrufen der Monitor-Methoden gilt:
„Die Summe der freien und der vergebenen Ressourcen ist k .“

Die Monitor-Klasse auf SWE-146 implementiert dieses Schema.

Beispiele:

- Walkman-Vergabe an Besuchergruppen im Museum (siehe [Java lernen, 13.5])
- Taxi-Unternehmen; $n = 1$

auch **abstrakte Ressourcen**, z. B.

- das Recht, eine Brücke begrenzter Kapazität (Anzahl Fahrzeuge oder Gewicht) zu befahren,

auch **gleichartige Ressourcen mit Identität**:

Der Monitor muß dann die Identitäten der Ressourcen in einer Datenstruktur verwalten.

- Nummern der vom Taxi-Unternehmen vergebenen Taxis
- Adressen der Speicherblöcke, die eine Speicherverwaltung vergibt

Vorlesung Software-Entwicklung II SS 2004 / Folie 148

Ziele:

Schema zur Vergabe gleichartiger Ressourcen verstehen

in der Vorlesung:

- Aufgabe erläutern
- Monitor von Folie 146 als Schema erläutern
- Varianten besprechen

nachlesen:

Judy Bishop: Java lernen, 2.Aufl., Abschnitt 13.3, 13.5

nachlesen:

- Vorlesung "Konzepte und Methoden der System-Software"
- G. R. Andrews: Concurrent Programming, Addison-Wesley, 1991

Übungsaufgaben:

Verständnisfragen:

- Geben Sie zu den Beispielen die jeweilige Ausprägung des Schemas an: Bedeutung der Ressourcen, sind sie konkret oder abstrakt, ist ihre Identität relevant; Bedeutung der Operationen, wer führt sie aus, wieviele Ressourcen werden in einer Operation angefordert.
- Geben Sie weitere Beispiele.

Schema: Beschränkter Puffer

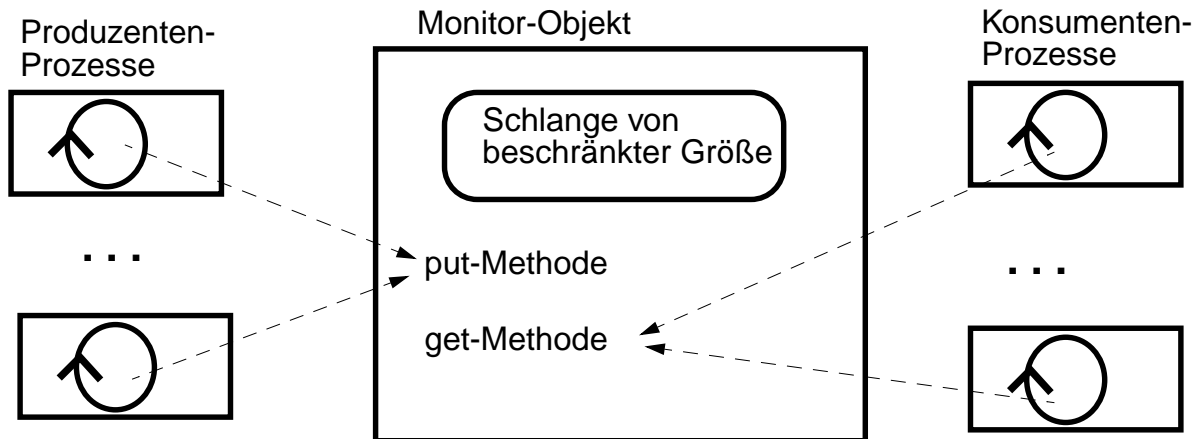
Ein Monitor speichert Elemente in einem **Puffer von beschränkter Größe**.

Produzenten-Prozesse liefern einzelne Elemente,
Konsumenten-Prozesse entnehmen einzelne Elemente.

Der Monitor stellt sicher, daß die Elemente in der gleichen **Reihenfolge** geliefert und entnommen werden. (Datenstruktur: Schlange)

Die **Wartebedingungen** lauten:

- in den Puffer **schreiben**, nur wenn er **nicht voll** ist,
- **aus dem Puffer nehmen**, nur wenn er **nicht leer** ist.



Vorlesung Software-Entwicklung II SS 2004 / Folie 149

Ziele:

Synchronisationsschema verstehen

in der Vorlesung:

- Schema erläutern
- mehrere, symmetrische Wartebedingungen
- Hinweis auf Schlangen-Implementierung

Verständnisfragen:

- Wo wird welche Wartebedingung verändert?

Monitor-Klasse für Beschränkte Puffer

```

class Buffer
{ private Queue buf;      // Schlange der Länge n zur Aufnahme der Elemente
  public Buffer (int n) {buf = new Queue(n); }

  synchronized public void put (Object elem)
  {
    // ein Produzenten-Prozess versucht, ein Element zu liefern
    while (buf.isFull())           // warten bis der Puffer nicht voll ist
      try {wait();} catch (InterruptedException e) {}
    buf.enqueue (elem); // Wartebedingung der get-Methode hat sich verändert
    notifyAll();           // jeder blockierte Prozess prüft seine Wartebedingung
  }

  synchronized public Object get ()
  {
    // ein Konsumenten-Prozess versucht, ein Element zu nehmen
    while (buf.isEmpty())         // warten bis der Puffer nicht leer ist
      try {wait();} catch (InterruptedException e) {}
    Object elem = buf.first();
    buf.dequeue(); // Wartebedingung der put-Methode hat sich verändert
    notifyAll(); // jeder blockierte Prozess prüft seine Wartebedingung
    return elem;
  }
}

```

© 2004 bei Prof. Dr. Uwe Kastens

Vorlesung Software-Entwicklung II SS 2004 / Folie 150

Ziele:

Systematischer Umgang mit Wartebingungen

in der Vorlesung:

Erläuterungen dazu

- Veränderung der Wartebedingungen
- `notifyAll` begründen
- Zustandsübergänge bei `notifyAll` in der `get`-Operation am Beispiel erläutern

Verständnisfragen:

- Welche Zustände bezüglich der Erfüllung der Wartebedingungen sind möglich?
- Was bedeutet das für mehrere wartende Prozesse?
- Geben Sie detailliert an, wie mehrere wartende Prozesse auf den Aufruf `notifyAll()` reagieren.

13. Verklemmungen

Ein **einzelner Prozess ist verklemmt** (in einer **Deadlock**-Situation), wenn er auf eine **Bedingung wartet, die nicht mehr wahr werden kann**.

Mehrere Prozesse sind untereinander verklemmt (in einer Deadlock-Situation), wenn sie **zyklisch aufeinander warten**;

d. h. die Wartebedingung eines Prozesses kann nur von einem anderen, auch wartenden Prozess erfüllt werden.

Verklemmung bei Ressourcenvergabe können eintreten, wenn folgende Bedingungen gelten:

1. Jeder Prozess **fordert mehrmals nacheinander Ressourcen an**.
2. Jeder Prozess benötigt bestimmte Ressourcen **exklusiv** für sich allein.
3. Die **Relation** „Prozess p benötigt eine Ressource die Prozess q“ hat ist **zyklisch**.

Beispiel: Dining Philosophers (SWE-152)

Vorlesung Software-Entwicklung II SS 2004 / Folie 151

Ziele:

Den Zustand "Verklemmung" verstehen

in der Vorlesung:

Erläuterungen dazu am Beispiel der Dining Philosophers, Folie 152

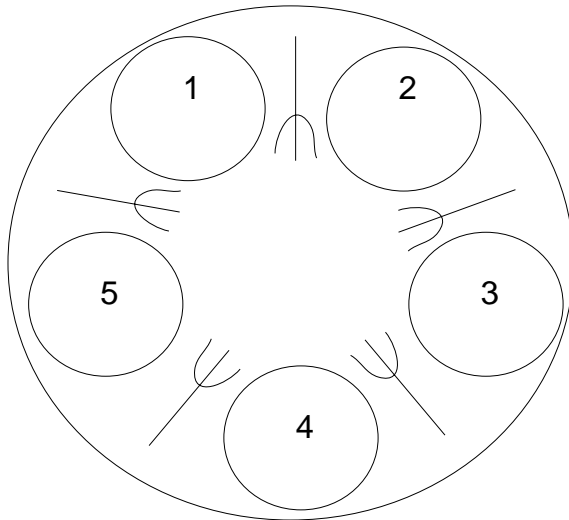
Verständnisfragen:

- Begründen Sie, daß für Verklemmungen bei Resourcevergabe jede der drei Bedingungen notwendig ist.
- Argumentieren Sie umgekehrt: Mein Prozesssystem wird nicht verklemmen, weil ...

Synchronisationsaufgabe: Dining Philosophers

Abstraktion von Prozesssystemen, in denen die Prozesse **mehrere Ressourcen exklusiv** benötigen. (Erstmals formuliert von E. Dijkstra 1968.)

5 Philosophen sitzen an einem gedeckten Tisch. Jeder beschäftigt sich abwechselnd mit denken und essen. Wenn ein Philosoph Hunger verspürt, versucht er die beiden Gabeln neben seinem Teller zu bekommen, um damit zu essen. Nachdem er gesättigt ist, legt er die Gabeln ab und verfällt wieder in tiefgründig philosophisches Nachdenken.



Abstraktes Programm für die Philosophen:

```
wiederhole
  denken
  nimm rechte Gabel
  nimm linke Gabel
  essen
  gib rechte Gabel
  gib linke Gabel
```

Es gibt einen Verklemmungszustand:

Für alle Prozesse i aus $\{1, \dots, 5\}$ gilt:
Prozess i hat seine rechte Gabel
und wartet auf seine linke Gabel.

Vorlesung Software-Entwicklung II SS 2004 / Folie 152

Ziele:

Typische Verklemmungssituation verstehen

in der Vorlesung:

- Beispiel erläutern
- Präzise Charakterisierung der Verklemmungssituation
- Die drei Bedingungen von Folie 151 zeigen
- Beispiel als Resourcevergabeschema verallgemeinern
- Eine Animation der Philosophen (Applet wird aus dem Internet geladen)

Verständnisfragen:

- Erläutern Sie an der Aufgabe die drei Bedingungen von Folie 151.

Dining Philosophers, Lösung (1)

Lösungsprinzip: Ressourcen (Gabeln) **zugleich anfordern** - nicht nacheinander.

Ein **Monitor** verwaltet die Gabeln als **Ressourcen mit Identität**.

Monitor-Operation: Dem Prozess i die linke und rechte Gabel geben.

Wartebedingung dafür: Die beiden Gabeln für den i -ten Prozess sind frei.

```
class ResourcePairs
{
    // nur Paare aus „benachbarten“ Ressourcen werden vergeben
    private boolean[] avail = {true, true, true, true, true};

    synchronized void getPair (int i)
    { while (!(avail[i % 5] & avail[(i+1) % 5]))
      try {wait();} catch (InterruptedException e) {}

      avail[i % 5] = false; avail[(i+1) % 5] = false;
    }

    synchronized void putPair (int i)
    { avail[i % 5] = true; avail[(i+1) % 5] = true;
      notifyAll();
    }
}
```

Der Verklemmungszustand „Jeder Prozess hat **eine** Gabel.“ kann nicht eintreten.

Vorlesung Software-Entwicklung II SS 2004 / Folie 153

Ziele:

Verklemmung systematisch vermeiden

in der Vorlesung:

- Bedingung (1) von Folie 151 wird invalidiert
- Der auf Folie 152 charakterisierte Zustand kann nicht eintreten
- Monitor systematisch entwickeln

Verständnisfragen:

Warum kann der auf Folie 152 charakterisierte Zustand nicht eintreten?

Dining Philosophers, Lösung (2)

Lösungsprinzip: Die Aussage die den Verklemmungszustand charakterisiert negieren.

Für alle Prozesse i aus $\{1, \dots, 5\}$ gilt: Prozess i hat seine rechte Gabel und wartet.

Negation liefert Lösungsidee:

Es gibt einen Prozess i aus $\{1, \dots, 5\}$: Prozess i hat seine rechte Gabel nicht oder er wartet nicht.

Lösung: Einer der Prozesse, z. B. 1, nimmt erst die linke dann die rechte Gabel.

Der Verklemmungszustand „Jeder Prozess hat **eine** Gabel.“ kann nicht eintreten.

Die Bedingung (3) von SWE-151 „zyklisches Warten“ wird invalidiert.

Ein **Monitor** verwaltet die Gabeln als **Ressourcen mit Identität**.

Monitor-Operation: Dem Prozess i die i -te Gabel (**einzeln!**) geben.

Wartebedingung dafür: Die i -te Gabel ist frei.

Vorlesung Software-Entwicklung II SS 2004 / Folie 154

Ziele:

Lösungsalternative kennenlernen

in der Vorlesung:

- Lösungsalternative herleiten
- kritischer Vergleich mit Lösung (1)

Verständnisfragen:

- Begründen Sie anhand der Rolle des Monitors, weshalb die Lösung (1) sicherer ist.

Zusammenfassung von SWE II

allgemeine Begriffe und Konzepte

Java Programmierung

Graphische Bedienungsoberflächen

Komponenten-Bibliothek
 hierarchische Objektstrukturen
 Eingabekomponenten
 Observer-Muster
 Reaktion auf Ereignisse, Folgen

AWT, Swing
 LayoutManager
 Button, TextField, ...
 Applets

Parallele Prozesse

Grundbegriffe
 parallel, verzahnt, nebenläufig

Java-Threads, Prozess-Objekte
 (1) implements Runnable, (2) extends Thread
 Thread-Methoden, Threads in Applets
 Monitor-Objekte in Java
 synchronized Methoden
 Methoden wait, notifyAll

Monitore
 gegenseitiger Ausschluss
 Bedingungssynchronisation im Monitor

Schema: Vergabe gleichartiger Ressourcen
 Schema: Beschränkter Puffer
 Schema: Ressourcenvergabe

Verklemmung

Vorlesung Software-Entwicklung II SS 2004 / Folie 155

Ziele:

Die Themen aus SWE II im Überblick sehen

in der Vorlesung:

- Auf die Begriffe, Konzepte, Techniken hinweisen
- Trennen zwischen allgemeinen Konzepten und Java-spezifischen Techniken
- Auf entsprechende Zusammenfassungen von SWE I verweisen.

Übungsaufgaben:

Wiederholen Sie die angesprochenen Themen.

Verständnisfragen:

Sammeln Sie die Verständnisfragen zu den Themen und beantworten Sie sie.