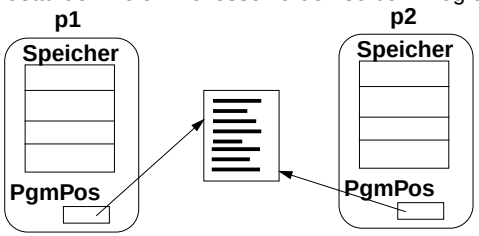
10. Parallele Prozesse
Grundbegriffe (1)

Prozess:
Ausführung eines sequentiellen Programmstückes in dem zugeordneten Speicher (Adressraum).
Zustand des Prozesses: Speicherinhalt und Programmposition

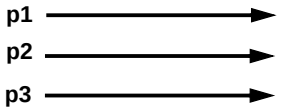
Zustände zweier Prozesse zu demselben Programm



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

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

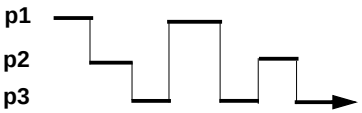
- keine Annahme über relative Geschwindigkeit
- **unabhängige** Prozesse oder
- **kommunizierende** Prozesse:
  p1 liefert Daten für p2 oder
  p1 wartet bis eine Bedingung erfüllt ist



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:

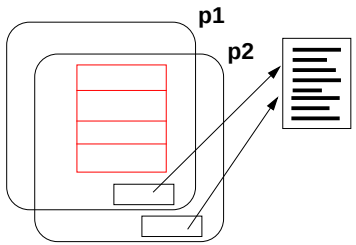
- Prozessor auslasten,
  blockierte Prozesse nicht bearbeiten
- Gleichzeitigkeit simulieren

nebenläufige Prozesse:
Prozesse, die parallel oder verzahnt ausgeführt werden können

Grundbegriffe (3)

Threads (“Fäden“ oder auch “Ausführungsstränge” der Programmausführung):
Prozesse, die parallel oder verzahnt in **gemeinsamem Speicher** ablaufen.

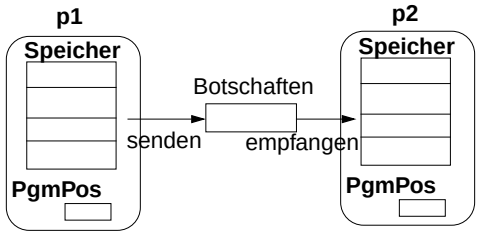
Die **Prozessumschaltung** ist besonders einfach und **schnell,**
daher auch **leichtgewichtige Prozesse**



Zugriff auf Variable im gemeinsamen Speicher

Im Gegensatz zu
Prozessen im verteilten Speicher auf verschiedenen Rechnern,

kommunizieren über Botschaften, statt über Variable im Speicher



Anwendungen Paralleler Prozesse

- **Benutzungsoberflächen**:
  Die Ereignisse werden von einem speziellen Systemprozess weitergegeben.
  Aufwändige 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.

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.

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.

11. Unabhängige parallele Prozesse

Beispiel: Digitale Uhr als Prozess im Applet (1)

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

```
class DigiClock extends Thread
{  public void run ()
   {  while (running)                          iterieren bis von außen terminiert wird
      {  line.setText(timeFormat.format(new Date()));     Datum schreiben
         try { sleep (1000); } catch (InterruptedException e) {}   Pause
      }
   }
                                               Methode, die den Prozeß von außen terminiert:
   public void stopIt () { running = false; }
   private boolean running = true;             Zustandsvariable
   public DigiClock (JLabel t) {line = t;}     Label zum Beschreiben übergeben
   private JLabel line;
   private DateFormat timeFormat = DateFormat.getTimeInstance(
                                   DateFormat.MEDIUM, Locale.GERMANY);
}
```

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

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

Beispiel: Digitale Uhr als Prozess im Applet (2)

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

```
public class DigiApp extends JApplet
{  public void init ()
   {  JLabel clockText = new JLabel ("--:--:--");
      getContentPane().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.

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 join () throws InterruptedException;`
der aufrufende Prozess wartet bis der angegebene Prozess terminiert ist:

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

`public static void sleep (long millisec) throws InterruptedException;`
der aufrufende Prozess wartet mindestens die in Millisekunden angegebene Zeit:

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

`public void suspend ();`
`public void resume ();`
hält den angegebenen Prozess an bzw. setzt ihn fort:

```
clock.suspend(); clock.resume();
```

unterbricht den Prozess u. U. in einem kritischen Abschnitt, nur für **`start`**/**`stop`** im Applet

`public final void stop () throws SecurityException;`
nicht benutzen! Terminiert den Prozess u. U. in einem inkonsistenten Zustand

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 (**`JComponent`**-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).

Prozessklasse der Ampelsimulation

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

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

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

   public void run ()
   {  while (running)                          // drei Ampelphasen wiederholen
      {  for (light = 0; light < 3; light++)
         {  Graphics g = area.getGraphics();   // Kontext für Zeichenfläche
            drawLights (g);                    // neuen Zustand zeichnen
            g.dispose();                       // Systemressourcen freigeben
            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 (Graphics g) {…}     // veränderliche Bildteile zeichnen
}
```


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 (Graphics g)
   {                                  // veränderliche Teile des Bildes zeichnen:

      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);
   }
}
```

Applet-Klasse der Ampelsimulation

```
public class TrafficLight extends JApplet
{  private JComponent 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 JComponent-Objekt zum Zeichnen der Ampeln
       new JComponent()                  // anonyme Unterklasse mit 1 Objekt:
        { public void paint (Graphics g) // Zeichnen an Ampel-Objekt deligieren
          { lights.draw(g);                          // zeichne statischen Teil
            lights.drawLights(g);               // 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(); }
}
```


12. 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.

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
{  public synchronized void abbuchen (...)   {...}
   public synchronized 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.

Beispiel: gegenseitiger Ausschluss durch Monitor

```
class Bank
{  public synchronized 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 n, Bank b) { name = n; meineBank = b; }
   public void run ()
   {  for (int i = 1; i <= 2; i++) meineBank.abbuchen(name); }
   private Bank meineBank; private String name;
}

class BankProgramm
{  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();
}  }
```

13. 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 (beliebigen) 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) {}
```


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
   }
}
```


Beispiel: Prozesse und Hauptprogramm zu GP-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 = rand.nextInt(max) + 1;
         mon.getElem (m, ident);                  // m Elemente anfordern
         try { sleep (rand.nextInt(1000) + 1);}
            catch (InterruptedException e) {}
         mon.putElem (m, ident);                // m Elemente zurückgeben
         rounds -= 1;
      }
   }
}
```

```
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();
}  }
```


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 GP-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

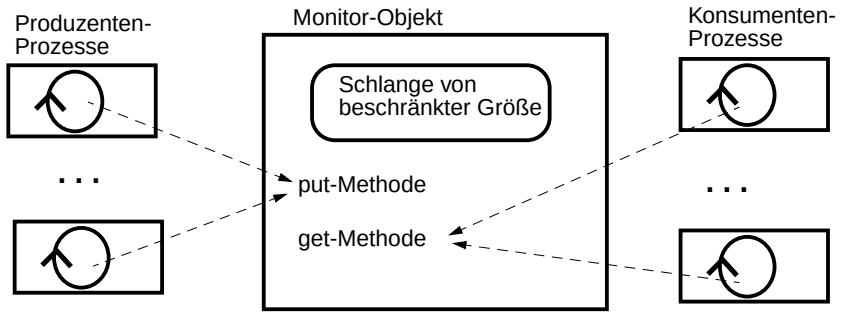
Schema: Beschränkter Puffer

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 **entnehmen, nur wenn** er **nicht leer** ist.



Monitor-Klasse für “Beschränkter Puffer”

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

   public synchronized 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
   }
   public synchronized Object get ()
   {             // ein Konsumenten-Prozess versucht, ein Element zu entnehmen
      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;
   }
}
```


14. 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, ebenfalls wartenden Prozess erfüllt werden.

Verklemmung bei Ressourcenvergabe kann eintreten,
wenn folgende Bedingungen gelten**:**

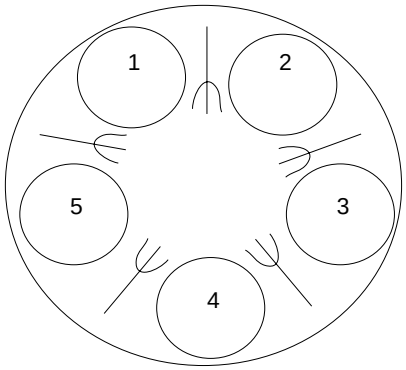
1. Jeder Prozess **fordert mehrmals nacheinander Ressourcen an**.
2. Jeder Prozess benötigt bestimmte Ressourcen **exklusiv** für sich allein.
3. Die **Relation** “Prozess *p* benötigt eine Ressource, die Prozess *q* hat” ist **zyklisch**.

Beispiel: Dining Philosophers (GP-152)

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,…,5} gilt:
Prozess *i* hat seine rechte Gabel
und wartet auf seine linke Gabel.

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.

Dining Philosophers, Lösung (2)

Lösungsprinzip: Die Aussage die den Verklemmungszustand charakterisiert negieren.

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

Negation liefert Lösungsidee:

Es gibt einen Prozess i aus {1,…,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 GP-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.

Zusammenfassung von GP II

allgemeine Begriffe und Konzepte	**Java Programmierung**
Graphische Bedienungsoberflächen	
Komponenten-Bibliothek	Swing (AWT)
hierarchische Objektstrukturen	**`LayoutManager`**
Eingabekomponenten	**`JButton`**, **`JTextField`**, …
Observer-Muster	Model/View-Paradigma
Reaktion auf Ereignisse, Folgen	Applets
Parallele Prozesse	
Grundbegriffe	Java-Threads, Prozess-Objekte
parallel, verzahnt, nebenläufig	Interface **`Runnable`**, Oberklasse **`Thread`**
	`Thread`-Methoden, Threads in Applets
Monitore	Monitor-Objekte in Java
gegenseitiger Ausschluss	**`synchronized`** Methoden
Bedingungssynchronisation im Monitor	Methoden **`wait`**, **`notifyAll`**
Schema: Vergabe gleichartiger Ressourcen	
Schema: Beschränkter Puffer	
Schema: Ressourcenvergabe	
Verklemmung	