Kapitel 9: Spezielle Programmierkonzepte

9.1 Benutzungsschnittstellen, Ereignisbehandlung

Viele der bislang betrachteten Programme bekamen ihre Eingabe als Parameter beim Aufruf, und lieferten ihre Ausgabe als Ergebnis bei Terminierung ab. *Interaktive Programme* kommunizieren mit den Benutzern über Ein- und Ausgabegeräte. Bei textbasierten Programmen erfolgt die Ausgabe oft in einem Textfeld („Konsole“); in Java etwa durch die Methode `System.out.println(…)`. Die Komponente „`out`“ bezeichnet dabei den Standard-Ausgabestrom des Systems, vom Typ `PrintStream`. (Ähnlich bezeichnet `err` die Standardausgabe für Fehlermeldungen, und `in` den Standard-Eingabestrom. Über `System.in` lassen sich also vom Benutzer auf der Konsolschnittstelle eingegebene Werte einlesen. Beim Aufruf der Methode `System.in.read` wartet das Programm, bis der Benutzer Text eingibt und die Eingabetaste drückt.
Das folgende Programmfragment liest eine Zahl von der Konsole und gibt sie (um 1 erhöht) wieder aus:

```
int i=0;
int len = 20; byte[] b = new byte[len];
try {
  int l = System.in.read(b, 0, len)-1;
  String s = new String(b,0,l-1);
  i = Integer.parseInt(s);
} catch (IOException e) {}
System.out.println(i+1);
```

`System.in.read` erwartet als Eingabe eine Referenz auf (hinreichend großes) Byte-Array, liest die Bytes von Konsole und gibt die Anzahl der gelesenen Zeichen (inklusive dem „Zeilenwechsel-Zeichen“) als Ergebnis zurück. Die tatsächlich eingegebene Zeichenzahl ist also um eins niedriger. Da Benutzerinteraktionen grundsätzlich Fehler erzeugen können, müssen sie in einen **`try`**–Block eingeschlossen werden, siehe Abschnitt 9.3. Mit der Anweisung `String s = new String(b,0,l-1);` werden die Zeichen 0 bis l-1 dieses Byte-Arrays in eine Zeichenreihe umgewandelt. `Integer.parseInt(s)` schließlich wandelt diese Zeichenreihe (sofern sie nach den Java-Regeln eine korrekte Zahl darstellt) in ein Objekt der Art `Integer` um. Etwas einfacher wird es, wenn wir aus `System.in` einen `BufferedReader` erstellen, da wir mit diesem direkt Strings einlesen können (Methode `readLine()`):

```
BufferedReader konsole = new BufferedReader(
                                 new InputStreamReader (System.in)) ;
try {i = Integer.parseInt(konsole.readLine());}
catch (IOException e){}
```

Für viele Programme ist allerdings eine rein textbasierte Ein-Ausgabe nicht ausreichend. Üblicherweise wird ein Programm heute mit dem Benutzer über eine GUI (graphical user interface) interagieren, d.h. das Programm stellt graphische Elemente wie Knöpfe, Regler, Textfelder usw. auf dem Bildschirm zur Verfügung, über die die Mensch-Maschine-Kommunikation erfolgt. Dazu gibt es inzwischen sehr viele verschiedene Möglichkeiten. Programme, die eine GUI bereitstellen, reagieren auf Ereignisse (events). In der ablaufgesteuerten Programmierung ist ein Programm eine Folge von Anweisungen (d.h. Methodenaufrufen); bei der ereignisgesteuerten Programmierung ist ein Programm eine Sammlung von Behandlungsroutinen (= Methoden) für verschiedene Ereignisse der GUI.

Typische Ereignisse (events) sind:

- Mausklick, Mausziehen, Mausbewegung, Return-Taste, Tastatureingabe, …
- Signale des Zeitgebers, Unterbrechungen, …
- programmerzeugte Ereignisse

In einem ereignisgesteuerten Programm wird für jedes Fenster ein Prozess gestartet, der auf die Ereignisse wartet und sie bearbeitet. Für jedes Ereignis wird eine Bearbeitungsroutine (event handler) registriert, die beim Eintreten des Ereignisses bestimmte Aktionen auslöst.

Zur konkreten Realisierung von GUIs in Java existieren zwei weit verbreitete Bibliotheken: AWT und Swing.

- AWT (Abstract Window Toolkit) ist eine plattformunabhängige schwergewichtige Bibliothek (stützt sich auf Betriebssystem-Routinen ab)
- Swing ist Bestandteil der Java-Runtime (leichtgewichtig), hat ein spezifisches „Look-and-Feel“ sowie gegenüber AWT einige zusätzliche Komponenten und ist generell „moderner“.

Das folgende Programm erzeugt zwei Fenster und zeigt sie auf dem Bildschirm an:

```
import java.awt.*;
class Fenster extends Frame {
  Fenster (String titel) {
    super(titel); //Setzen des Textes in der Titelzeile des Fensters
    setSize(160,120); // Größe in Pixeln einstellen
    setVisible(true); // das Fenster ist anfangs sichtbar
  }
}

public class FensterDemo {
  public static void main(String[] args) {
  Fenster zumHof = new Fenster("zum Hof");
  Fenster zurTür = new Fenster("zur Tür");
  }
}
```

Damit diese Fenster z.B. auf das Schaltfeld „Beenden“ (in Windows: das rote Kreuz rechts oben) reagieren, muss man ihnen einen entsprechenden Fensterbeobachter zuordnen:

```
import java.awt.*;
import java.awt.event.*;
class FensterBeobachter extends WindowAdapter{
  public void windowClosing(WindowEvent e) {// Schließen des Fensters
    System.exit(0); // Aktion beim Schließen: z.B. Programmende
  }
}
class Fenster extends Frame {
  Fenster (String titel) {
    FensterBeobachter fb = new FensterBeobachter();
    addWindowListener(fb); // hier wird der Beobachter registriert
    super(titel); ...); // wie oben
  }
}
public class FensterDemo {
  public static void main(String[] args) {
  Fenster f = new Fenster("PI-1");
  }
}
```

Innerhalb einem Fenster kann man verschiedene Knöpfe definieren. Jedem Knopf wird ein Knopfbeobachter zugewiesen, der auf das Ereignis „Knopf wird gedrückt“ reagiert:

```
class KnopfBeobachter implements ActionListener{
 public void actionPerformed(ActionEvent e) {
    System.out.println("Knopf gedrückt!"); //Aktion beim Klick
  }
}
```

```
class Fenster extends Frame {
  Fenster (String titel) {
  ...
  Button k = new Button("Knopf!"); // neuen Knopf definieren
  KnopfBeobachter kb = new KnopfBeobachter(); // neuer Beobachter kb
  k.addActionListener(kb); // Beobachter kb Knopf k zuordnen
  add(k); // Knopf zum aktuellen Fensterobjekt hinzufügen
  }
}
```

Zur Verwendung aktiver Komponenten (Knöpfe, Schalter, …) in einer GUI sind also immer folgende Schritte nötig

- Erzeugen – `Button b = new Button ()`
- Ereignisbehandlung zuordnen – `b.add.ActionListener(...)`
- Hinzufügen zum Fenster – `add(b)`
- Methode zur Ereignisbehandlung schreiben
  - `actionPerformed, itemStateChanged, adjustmentValueChanged,...`

Jedem Schaltelement können beliebig viele Beobachter zugeordnet werden, die auf die unterschiedlichen Ereignisse reagieren. Beobachter müssen nicht als eigene Variablen benannt werden, sondern können auch anonym existieren:

```
setLayout(new FlowLayout());
Button knopf2 = new Button("Knopf2");
knopf2.addActionListener (new ActionListener (){
  public void actionPerformed(ActionEvent e) {
  System.out.println("Knopf2 gedrückt!"); }});
add(knopf2);
...
```

Für die graphische Gestaltung der Benutzerinteraktion ist eine Vielzahl von automatischen Optionen verfügbar; zu Layout-Fragen konsultiert man am besten die entsprechende Dokumentation. Gängige Layouts sind z.B.

- BorderLayout
- FlowLayout
- GridLayout
- CardLayout
- GridBagLayout

Es ist auch eine Schachtelung von Layouts möglich; darüber hinaus lässt sich natürlich jedes Objekt auch absolut innerhalb des Fensters positionieren.

Als ein etwas größeres Beispiel betrachten wir jetzt eine Stoppuhr, die durch entsprechende Mausklicks gestartet und gestoppt wird, und bei der die Zeitanzeige im Format (Minuten, Sekunden, Hunderstelsekunden) in Textfeldern des Fensters erscheint. Die Klasse enthält die Zeitanzeige mit drei Label (Minuten, Sekunden, Hundertstel) sowie die beiden Knöpfe StartStop und Reset.

```
class Stoppuhr extends Frame {
  Label zeitAnzeigeMin, zeitAnzeigeSec, zeitAnzeigeCsec;
  Button startstop, reset;
  long startTime, elapsedTime, elapsedLast = 0;
  boolean running = false;
```

```
Stoppuhr(){ // Konstruktor erzeugt Layout
  zeitAnzeigeMin = new Label();
  zeitAnzeigeMin.setBounds(50, 50, 40, 25);
  add(zeitAnzeigeMin);
  zeitAnzeigeSec = new Label();
  zeitAnzeigeSec.setBounds(100, 50, 40, 25);
  add(zeitAnzeigeSec);
  zeitAnzeigeCsec = new Label();
  zeitAnzeigeCsec.setBounds(150, 50, 40, 25);
  add(zeitAnzeigeCsec);

  startstop = new Button("Start");
  startstop.setBounds(50,100,60,25);
  startstop.addActionListener(new ButtonListenerStartStop());
  add(startstop);

  reset = new Button("Reset");
  reset.setBounds(120,100,60,25);
  reset.addActionListener(new ButtonListenerReset());
  add(reset);

  addWindowListener(new WindowAdapter (){
    public void windowClosing(WindowEvent e) { System.exit(0); }
  });
  add(new Button()); // sonst Layoutfehler?!?
}

public static void main(String[] args) {
  Stoppuhr uhr = new Stoppuhr(); // erzeuge neues Fenster
  uhr.setBounds(0, 0, 250, 180); //250 Pixel breit, 180 Pixel hoch
  uhr.setVisible(true);
  uhr.new Zeitanzeige ().start(); // starte separaten Thread
}
```

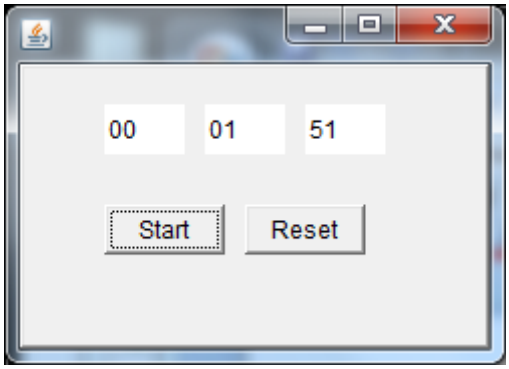
Die Main-Methode startet die Uhranzeige als parallelen Thread, der ständig die Anzeige aktualisiert:

```
class Zeitanzeige extends Thread{
  public void run(){
    while(true){
      if (running){
        elapsedTime = (System.currentTimeMillis() - startTime)
           + elapsedLast; // nach stop, weiter- statt neuzählen
        int csec = (int) (elapsedTime % 1000 / 10);
        int sec = (int) (elapsedTime / 1000) % 60;
        int min = (int) (elapsedTime / 60000);
        zeitAnzeigeCsec.setText(((csec < 10)? "0" : "") + csec);
        zeitAnzeigeSec.setText(((sec < 10)? "0" : "") + sec);
        zeitAnzeigeMin.setText(((min < 10)? "0" : "") + min);
      }
    }
  }
}
```

Schließlich müssen wir noch die Aktionen definieren, die beim Klicken der Knöpfe ausgeführt werden:

```
    class ButtonListenerStartStop implements ActionListener{
      public void actionPerformed(ActionEvent e){
        startTime = System.currentTimeMillis(); // neue Startzeit
        if (! running){
          running = true;
          startstop.setLabel("Stop");
        } else {
          running = false;
          elapsedLast = elapsedTime; // Speichern des gestoppten Stands
          startstop.setLabel("Start");
        }
      }
    }
    class ButtonListenerReset implements ActionListener{
      public void actionPerformed(ActionEvent e){
        startTime = System.currentTimeMillis();
        elapsedLast = 0;
        zeitAnzeigeMin.setText("00");
        zeitAnzeigeSec.setText("00");
        zeitAnzeigeCsec.setText("00");
        startstop.setLabel("Start");
      }
    }
  }
```

Das entstehende Fenster sieht wie folgt aus:



Graphikprogrammierung

In einem interaktiven Programm möchte man mit der Maus normalerweise noch mehr machen, als nur Knöpfe anzuklicken. Mausevents (Ziehen, Doppelklick, Rechtsklick usw.) werden nach dem selben Schema wie Knopfdrücke behandelt

```
addMouseListener(new MouseAdapter() {
   public void mousePressed(MouseEvent e) {
    x1 = e.getX(); y1 = e.getY(); } ...
```

Auf diese Weise ist es möglich, graphische Elemente (Linien, Kreise, Vielecke, Polygone…) als Objekt der Klasse `Graphics` auszugeben und mit der Maus zu bearbeiten.

- Verwendete Methoden: `drawLine, fillPolygon, drawOval, ...`
- Konstruktor `getGraphics()`

Als Beispiel implementieren wir ein Malprogramm: Nach Klicken auf die Knöpfe „Rechteck“ oder „Oval“ können entsprechende Figuren durch Ziehen mit der Maus gezeichnet werden.

```
import java.awt.*;
import java.awt.event.*;
class GrafikFenster extends Frame{
  private static int x1, y1, x2, y2;
  private static int modus = 0;
  GrafikFenster(String titel, int breite, int höhe){
    super(titel);
    setLayout(new FlowLayout());
    Button rechteckButton = new Button ("Rechteck");
    rechteckButton.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent event) {
         modus = 1; } });
    add(rechteckButton);
    Button ovalButton = new Button ("Oval");
    ovalButton.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent event) {
         modus = 2; } });
    add(ovalButton);
    addMouseListener(new MouseAdapter() {
       public void mousePressed(MouseEvent e) {
         x1 = e.getX(); y1 = e.getY();
       }
       public void mouseReleased(MouseEvent e) {
         x2 = e.getX(); y2 = e.getY();
         Graphics g = getGraphics();
         g.setColor(Color.blue);
         int w=Math.abs(x1-x2), h=Math.abs(y1-y2);
         switch (modus) {
           case 1: g.fillRect(x1,y1,w,h); break;
           case 2: g.fillOval(x1,y1,w,h); break;
           default: break;
         }
       }
    });
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
      dispose(); }
    });
  setBackground(Color.lightGray);
  setSize(breite, höhe);
  setVisible(true);
 }
}
public class GrafikDemo {
  public static void main(String[] args) {
    GrafikFenster f = new GrafikFenster("Künstler",800,600);
  }
}
```

Animationen

Ein Problem des obigen Programms ist es, dass bei Veränderung des Fensters (z.B. Vergrößern) die Zeichnung verloren geht. Das kann mit der Methode `paint (Graphics g)` zur Ausgabe der Zeichnung gelöst werden. `paint` wird bei Fensterveränderungen automatisch aufgerufen und muss so programmiert werden, dass es alle Zeichnungsobjekte neu ausgibt. Eine Möglichkeit dazu besteht darin, alle Zeichnungsobjekte in einer `Collection` (z.B. `Set`) einzutragen und beim Neuzeichnen iterativ alle Elemente auszugeben. Die selbe Technik kann auch für Animationen benutzt werden, indem `paint()` in eigenem Thread (etwa alle 40 ms) aufgerufen wird.

9.2 Abstrakte Klassen, Interfaces, generische Typen

Abstrakte Klassen

Abstrakte Klassen sind solche, die noch nicht „fertig“ sind: sie enthalten Methoden ohne Implementierung. Beispiel:

```
abstract class Figur {
  protected int x, y;
  public void setzeX(int xNeu) {
    x = xNeu;
  }
  public abstract float berechneFlaeche();
}
```

Für abstrakte Klassen ist keine Instanzenbildung erlaubt; allerdings können Unterklassen die abstrakte Klasse erweitern und die abstrakten Methoden konkretisieren.

Interfaces

Ein Interface ist das Java-Äquivalent zur Signatur eines abstrakten Datentyps. Es besteht nur aus den Köpfen der Methoden, enthält keine Datenfelder oder Algorithmen (aber auch keine algebraischen Gesetze). Eingeleitet werden Interfaces mit dem Schlüsselwort interface: Beispiel:

```
public interface StackInterface {
    public boolean isempty();
    public void push(char x);
    public char top();
    public void pop();
}
```

Die implementierende Klasse referenziert dann das Interface:

```
class Stack implements StackInterface { ... }
```

Interfaces sind gut geeignet, um Benutzungsschnittstellen von Klassen noch vor der eigentlichen Implementierung festzulegen. Von einem Interface können keine Instanzen gebildet werden (da ja keine Methodenrümpfe und Datenfelder vorliegen). Allerdings kann der Compiler die Konsistenz einer Benutzung der Klasse sowie die Konsistenz einer Implementierung des Interfaces mit der Definition überprüfen. Eine Variable, die als Typ das Interface hat, darf als Wert Objekte aller implementierenden Klassen haben. Ein wesentlicher Unterschied zwischen abstrakten Klassen und Interfaces besteht darin, dass eine Klasse mehrere Interfaces implementieren kann (sogenannte Mehrfachvererbung).

Generische Typen

Generische Typen realisieren da Konzept der parametrisierten abstrakten Datentypen. Es ist möglich, einer Klasse einen Typ als Parameter mitzugeben:

```
class Tupel<T> {
  private T first;
  private T second;

  public Tupel(T fst, T scd) {
    first = fst;
    second = scd;
  }

  public T getFirst() {
    return first;
  }

   public T getSecond() {
     return second;
   }
}
```

Der Zugriff erfolgt dann wie folgt:

```
pi = new Pair<Integer> (17, 24);
ps = new Pair<String> ("Hallo", "World");
```

Allerdings ist das Konzept in Java nur eingeschränkt nutzbar, da z.B. keine Felder über generischen Typen gebildet werden können.

9.3 Fehler, Ausnahmen, Zusicherungen

Ausnahmen

Eine typische Situation beim Programmieren ist der Entwurf von Klassenbibliotheken für (spätere) Anwendungen. Hier muss man beständig mit unzulässige Eingaben rechnen (z.B. top(empty()), Division durch Null, usw.). Es erhebt sich die Frage, wie man mit solchen unzulässigen Eingabewerten umgehen sollte. Eine Möglichkeit besteht darin, die unzulässige Eingabe zu ignorieren und einen Standardwert zurückgeben. (z.B. x/0 = 0). Dies hat mehrere Nachteile: Der Definitionsbereich der Funktionen wird unzulässig erweitert, der Anwender der Klasse bemerkt seinen FDehler nicht und der Fehler wird unkalkulierbar verschleppt. Besser ist es, das Programm mit einer Fehlermeldung abzubrechen („lieber ein Ende mit Schrecken als ein Schrecken ohne Ende“). Die dritte Möglichkeit besteht darin, zu jeder Methode einen zusätzlichen boole’schen Ergebniswert („ok“) einzuführen, der auf true gesetzt wird wenn der Methodenaufruf fehlerfrei war. Diese Methode erfordert gegebenenfalls erheblichen zusätzlichen Schreibaufwand, wird aber z.B. in der Programmiersprache C praktiziert. In Java besteht die Lösung im „Aufwerfen“ (throw) einer Ausnahmesituation. Das „Abfangen“ (catch) der Ausnahmesituation liegt in der Verantwortung des Aufrufers; wenn er sich nicht darum kümmert, wird die Ausnahmesituation „nach oben weitergereicht“. Wenn sich niemand um den Fehler kümmert, bricht das Programm ab. Dadurch ist es möglich, gewisse Fehler auf der entsprechenden Programmebene kontrolliert zu behandeln, Debugging-Informationen auszugeben, oder das Programm sogar unter Umständen fortzusetzen.

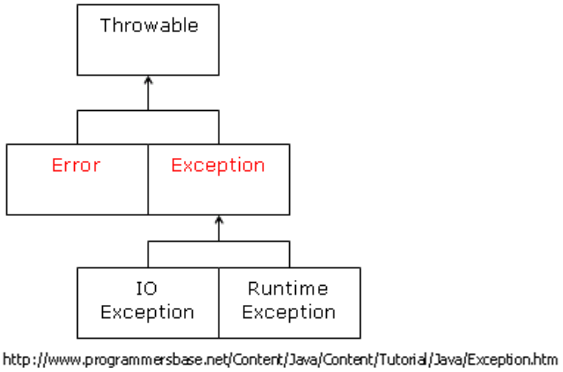
```
public class Ausnahmebehandlung {
  static void f() throws Exception {
    Exception e = new Exception("Fehler1");
      if (1==1) throw e;
  }
  public static void main(String[] args) {
    // …
    System.out.println("vor Aufruf");
    try {
      f();
    } catch (Exception x) {
      System.out.println (x + " ist aufgetreten");
    }
    // …
  }
}
```

Achtung: Ausnahmesituationen gehören zur Signatur der Methode! Dies bedeutet, dass Ausnahmesituationen in Java „Bürger erster Klasse“ sind. Eine Ausnahmesituation ist ein Objekt der Klasse Exception. Das bedeutet, es gibt Konstruktoren ohne und mit Argument (String). Das Werfen einer Ausnahmesituation (throw) bewirkt die Beendigung der Methode und die Rückkehr zur Aufrufstelle. Aufruf einer Methode, die eine Ausnahmesituation werfen kann, ist nur in einem try-Block möglich; das Auftreten der Ausnahmesituation wird von nachfolgenden Exception Handlern überwacht. Der erste Exception Handler, dessen Parameter-Typ mit dem Typ der geworfenen Exception übereinstimmt, wird ausgeführt, und

nach Beendigung wird des Exception Handlers wird das Programm normal fortgesetzt. Auf diese Weise ist es möglich, für jede mögliche Ausnahmesituation eine dezidierte Fehlerbehandlung anzustoßen. Jede Ausnahme muss behandelt oder weitergereicht werden (Ausnahme folgt), d.h., der Compiler zwingt den Programmierer, sich über die notwendigen Maßnahmen beim Eintreten der Ausnahmesituation Gedanken zu machen.

```
f();//geht nicht!
static void g() throws Exception {
  f();
}
```

Exception-Hierarchie



- Basisklasse aller Ausnahmen ist Throwable
- Davon abgeleitet sind
  Exception und Error
  - Konvention: Exception für behebbare, Error für nicht vom Anwender behebbare Ursachen
  - Von Exception werden u.a. IOException und RuntimeException abgeleitet
  - RuntimeException und davon abgeleitete Klassen müssen nicht behandelt werden

Syntax

```
try {
  // Anweisungen die Ausnahmen werfen könnten
} catch(Typ1 Objekt1) {
  // Anweisungen für Ausnahme1
} catch(Typ2 Objekt2) {
  // Anweisungen für Ausnahme2
} finally {
  // Anweisungen für Endbehandlung
}
```

Es ist natürlich möglich, eigene Ausnahmen zu definieren:

```
static class MyException1 extends Exception{}
static class MyException2 extends Exception{}
public class Ausnahmebehandlung {
 static void f() throws MyException1, MyException2 {
   if (1==1) throw new MyException1();
  else throw new MyException2();
 }
 public static void main(String[] args) {
  System.out.println("vor Aufruf");
  try {
   f();
  } catch (MyException1 x) {
   System.out.println (x+" - Handler1");
  } catch (MyException2 x) {
   System.out.println (" Handler2: " + x);
  }
  finally {System.out.println("Finally!");}
 };
}
```

Hier noch eine allgemeine Anmerkung zu Ausnahmen: Ausnahmen sollten NICHT zur Ablaufkontrolle missbraucht werden! Ausnahmen von dieser Regel sind möglich und zulässig (z.B. bei Benutzereingaben). Als Beispiel verfeinern wir unser Programm zum Einlesen einer natürlichen Zahl (größer Null).

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class ZahlEingabe {
  public static void main(String[] args) {
    int i = 0;
    System.out.println ("Bitte Zahl eingeben!");
    while (i <= 0){
      BufferedReader inputReader =
        new BufferedReader(new InputStreamReader(System.in));
      String eingabeZeile = "";
      try {
        eingabeZeile = inputReader.readLine();
        i = Integer.parseInt (eingabeZeile);
        if (i<=0) {
          System.out.println ("Bitte Zahl größer Null eingeben!");}
      }
      catch (NumberFormatException e){
          System.out.println ("Not a Number");}
      catch (IOException e){
          System.out.println ("I/O Exception!");}
    }
    System.out.println ("Eingegeben: " + i);
  }
}
```

Zusicherungen

Zusicherungen (Assertions) sind boole‘sche Ausdrücke, die dazu dienen, die Sicherheit beim Programmieren zu erhöhen. Sie können bei der Entwicklung von Programmen helfen, Irrtümer und Missverständnisse zu vermeiden. Eingeleitet werden Zusicherungen durch das Schlüsselwort

```
assert expr;
```

oder auch

```
assert expr : expr;
```

Beispiel für eine Zusicherung:

```
assert (i<liste.length()): "Index " + i + " falsch";
```

Semantik: Während des Ablaufs des Programms wird geprüft, ob die Zusicherung eingehalten ist. Falls nicht, wird eine Ausnahme geworfen (bzw. der Ausdruck an die Ausnahme übergeben). Verwendung finden Zusicherungen unter anderem als Vor- und Nachbedingungen von Methoden (z.B. beim Aufruf von pop(x) wird zugesichert, daß !isEmpty(x)). Zusicherungen sind, wenn sie richtig eingesetzt werden, eine effiziente Methode zur Entwicklung und zum Debuggen von Programmen.
Bei der Ausführung verlangsamt die Auswertung einer Zusicherung natürlich das Programm. Daher können Zusicherungen in der Java-Laufzeitumgebung ein- und ausgeschaltet werden:

Argument: -enableassertions oder –ea

Wenn die Zusicherungsauswertung ausgeschaltet ist, können Zusicherungen als eine Art Kommentar betrachtet werden.

9.4 Parallelität

Im objektorientierten Paradigma bedeutet Berechnung die Interaktion von Objekten. Das bedeutet, dass mehrere Objekte gleichzeitig existieren. In der sequentiellen Programmierung ist jedoch immer nur eines dieser Objekte aktiv. Anfangs gibt es nur das statische Main-Objekt. Dieses erzeugt andere Objekte und ruft diese auf. Jeder Aufruf ist blockierend, d.h. der Aufrufer muss warten, bis das aufgerufene Objekt fertig ist und die Kontrolle zurückgibt. Folglich ist zu einem gegebenen Zeitpunkt immer nur ein Objekt rechnend. Parallelität bedeutet, dass mehrere Handlungsstränge zur selben Zeit ablaufen. Dies ist

1. dem objektorientierten Paradigma angemessener
2. für die Ausführung auf Mehrkernprozessoren besser geeignet.

Es gibt in Java mehrere Arten der Parallelität:

- schwergewichtige Parallelität: mehrere Prozesse (tasks) gleichzeitig, werden vom Betriebssystem verwaltet, Kommunikation über spezielle BS-Mechanismen (pipes, sockets)
- leichtgewichtige Parallelität: mehrere Handlungsfäden (threads) innerhalb einer Task, Verwaltung vom Laufzeitsystem, Kommunikation über gemeinsame Variable

Threads in Java

In Java gibt es zwei Arten der Definition

1. Erweiterung der Klasse Thread mit Überlagerung der Methode run
2. Implementierung der Schnittstelle Runnable mit neuer Methode run
    - Definition eines objektlokalen Datenfelds t vom Typ Thread, Erzeugung eines zugehörigen Thread-Objektes
    - Aufruf von t.start() führt zur Ausführung von run als separater Handlungsfaden

Beispiel (1): Wir erzeugen eine Erweiterungsklasse von Thread mit Methode run. Diese wird dann vom aufrufenden Programm aus gestartet.

```
class Toe extends Thread {
  public void run() {
    System.out.println("Prozess Toe gestartet");
  }
  public static void main(String[] args) {
    Toe toe = new Toe(); toe.start();
    System.out.println("Prozess Toe beendet");
  }
}
```

Achtung: Das Ergebnis ist (in dieser Reihenfolge!):

```
Prozess Toe beendet
Prozess Toe gestartet
```

Beispiel (2): Wir erzeugen zwei Instanzen einer Implementierungsklasse zum Interface Runnable. Diese enthalten jeweils einen Thread, der vom Aufrufer gestartet werden kann.

```
class TicTacToe {
  static class TicTac implements Runnable{
    Thread faden;
    private int wer;
    public TicTac(int w) {
      faden = new Thread(this);
      wer=w;}
```

```
    public void run() {
      System.out.println("Prozess T"+((wer==1)?"i":"a")+"c
                         gestartet");
    }
  }

  public static void main(String[] args) {
    TicTac tic = new TicTac(1);
    TicTac tac = new TicTac(2);
    tic.faden.start(); tac.faden.start();
    try {tic.faden.join(); tac.faden.join();
    } catch (Exception e) {}
    System.out.println("alles beendet");
  }
}
```

Wie das Beispiel zeigt, kann der Aufrufer auch auf die Beendigung eines Threads warten. Beispiel (3) zeigt, dass man bei der parallelen Programmierung vorsichtig sein muss, weil sich leicht Fehler einschleichen. Wir inkrementieren und dekrementieren eine gemeinsame Variable in zwei verschiedenen Threads gleich oft.

```
class TicTac implements Runnable{
  static int summe = 0;
  Thread faden;
  private int wer;
  public TicTac(int w) {
    faden = new Thread(this);
    wer=w;
  }
  public void run() {
    for(int i=1; i<10000; i++) {
      if(wer==1) summe = summe + 1;
      else summe = summe - 1;
    }
  }

  public static void main(String[] args) {
    TicTac tic = new TicTac(1);
    TicTac tac = new TicTac(2);
    tic.faden.start(); tac.faden.start();
    try {tic.faden.join(); tac.faden.join();
    } catch (Exception e) {}
    System.out.println("Summe=" + summe);
  }
}
```

Der Thread `tic` zählt die Variable summe hoch, der Thread `tac` zählt sie runter. Das Ergebnis ist aber nicht in jedem Fall 0, sondern unvorhersagbar. (Für „kleine“ Schleifen ergibt sich jedoch immer 0).

Interpretation der Ergebnisse

- Prinzipiell werden die Aktionen in den Threads parallel und unabhängig voneinander ausgeführt
- Falls mehr Prozesse als Prozessoren existieren, muss die Rechenzeit aufgeteilt werden
- Zeitscheibenzuteilung: Jeder Thread erhält eine bestimmte Zeitspanne, bevor er unterbrochen wird. Dadurch kann es passieren, dass ein bereits Thread fertig ist, bevor der zweite überhaupt anfängt
- Interleaving: Durch parallelen Zugriff auf gemeinsame Variable können Werte verfälscht werden.

Interleaving

Durch verzahnt Ausführung (wie bei einem Reißverschluss) können mehrere Prozesse auf nur einem Prozessor ausgeführt werden. Jeder Prozess erhält dabei der Reihe nach eine Zeitscheibe einer bestimmten Dauer zugewiesen (die Dauer ist nicht notwendigerweise immer gleich (Bild „Einfädeln“ von Autos bei einer Baustelle). Beim Zugriff auf gemeinsame Variable kann es dabei zu unerwarteten Ergebnissen kommen:

Der Befehl {s++} wird zur Befehlsfolge {load s; add 1; store s} übersetzt. Analog bedeutet {s--} in Maschinensprache {load s; sub 1; store s}. Verschiedene Prozesse benutzen dabei verschiedene Register für die Rechnung. Bei zwei Prozessen, die beide auf dieselbe Variable zugreifen ergeben sich jedoch bei der quasiparallelen Ausführung von s++ || s-- folgende Szenarien:

s=5 (z.B.)

Befehl P1	Befehl P2	Werte (AC1, AC2, s)
load s		(5, -, 5)
add 1		(6, -, 5)
store s		(6, -, 6)
	load s	(6, 6, 6)
	sub 1	(6, 5, 6)
	store s	(6, 5, **5**)

aber auch:

Befehl P1	Befehl P2	Werte (AC1, AC2, s)
load s		(5, -, 5)
	load s	(5, 5, 5)
add 1		(6, 5, 5)
	sub 1	(6, 4, 5)
store s		(6, 4, 6)
	store s	(6, 4, **4**)

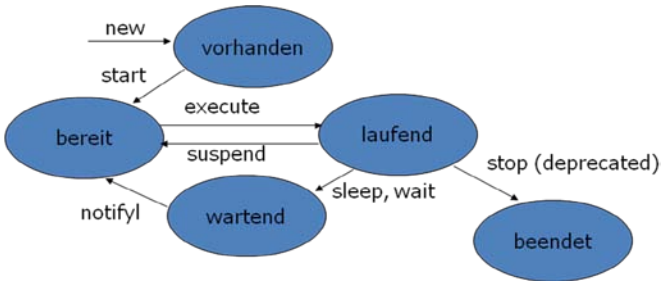
oder

Befehl P1	Befehl P2	Werte (AC1, AC2, s)
load s		(5, -, 5)
	load s	(5, 5, 5)
	sub 1	(5, 4, 5)
	store s	(5, 4, 4)
add 1		(6, 4, 4)
store s		(6, 4, **6**)

Das bedeutet, wenn man s simultan inkrementiert und dekrementiert, ist das Ergebnis zufällig s oder s+1 oder s-1! Wenn wir diesen Effekt oft wiederholen (for (int i = 0; i<1000000; i++)) verstärkt sich der Fehler.

Prozess-Zustände

Prozesse können dem Laufzeit- oder Betriebssystem selbst ankündigen, ob sie nach Ablauf ihrer Zeitscheibe weiter ausgeführt werden sollen oder nicht. Das nennt man präemptives Multitasking.



Zur Synchronisation von Prozessen gibt es in Java folgende Möglichkeiten:

- start() – startet den Thread
- sleep(int i) – suspendiert den Prozess i ms
- wait() – Wartet auf den Eintritt eines Ereignisses
- notify() – benachrichtigt einen wartenden Thread
- notifyAll() – benachrichtigt alle wartenden Threads
- join() – wartet auf die Beendigung des Threads
- interrupt() – unterbricht die Ausführung eines Threads
- isInterrupted() – Test ob unterbrochen
- synchronized – exklusiver Ressourcenzugriff-Monitor

Ein Monitor ist eine Klasse, die die Unteilbarkeit auf die von ihr verwalteten Ressourcen garantiert. Dies geschieht mit dem Schlüsselwort synchronized vor einer Methode.

```
class Monitor {
    private int i = 0;
    synchronized void inc() { i++;}
    synchronized void dec() { i--;}
    int val() { return i; }

class TicTacMon implements Runnable{
 static Monitor summe = new Monitor();
 Thread faden;
 private int wer;
 public TicTacMon(int w) {
  faden = new Thread(this);
  wer=w;}
 public void run() {
  System.out.println("Prozess " + wer + " gestartet");
  for(int i=1;i<10000000;i++) {
    if(wer==1) summe.inc();
    else summe.dec();
  }
  System.out.println("Prozess "+ wer + " beendet");
 }
public static void main(String[] args) {
  TicTacMon tic = new TicTacMon(1);
  TicTacMon tac = new TicTacMon(2);
  tic.faden.start(); tac.faden.start();
  try {tic.faden.join(); tac.faden.join();} catch (Exception e) {}
  System.out.println("Summe=" + summe.val());
 }
}
}
```

synchronisierte Methoden

Für jedes Objekt gibt es ein *intrinsisches Schloss*, welches den Zugang einschränkt. Während des Ablaufs einer synchronisierten Methode wird das Schloss verschlossen, daher kann keine andere synchronisierte Methode dieses Objekts gleichzeitig ablaufen. Diese muss ggf. warten (Gefahr der *Verklemmung (deadlock)!*)

Mit einer Synchronisationsanweisung lässt sich der gleiche Effekt wie mit einem Monitor erzielen:

```
class Mutex {};
class TicTac implements Runnable{
 static Mutex m = new Mutex();
 ...
 public void run() {
  for(int i=1;i<10000000;i++) {
   synchronized(m) {
    if(wer==1) summe++;
    else summe--;
   }
 ...
```

Dinierende Philosophen

Dieses Standardbeispiel von E.W.Dijkstra verdeutlicht die Probleme, die bei der Koordinierung paralleler Abläufe durch die Konkurrenz um gemeinsame Betriebsmittel auftreten können. 5 Philosophen sitzen um einen runden Tisch, in der Mitte steht eine Schüssel Spaghetti. Zwischen je zwei Philosophen ist eine Gabel (bzw. ein Ess-Stäbchen). Jeder Philosoph betätigt sich zyklisch nur mit den Tätigkeiten denken – essen – denken – essen – denken – essen – denken –... Zum Essen benötigt ein Philosoph allerdings zwei Gabeln / Ess-Stäbchen, nämlich die zu seiner linken und zu seiner rechten. Ein einfacher Algorithmus für jeden Philosophen wäre also:

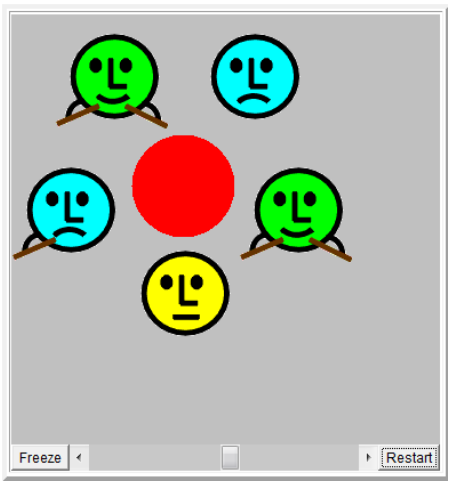
wiederhole immer wieder:
denke
warte, bis linke Gabel frei, dann nimm sie
warte, bis rechte Gabel frei, dann nimm sie
iss
gib beide Gabeln wieder frei

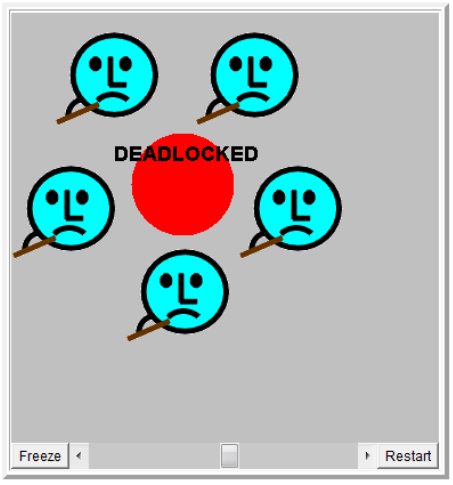
In Java lässt sich das etwa wie folgt formulieren:

```
public class Gabel {
  private boolean benutzt = false;
  private int welche;
  Gabel (int i){welche = i;}

  synchronized void aufnehmen()
    throws InterruptedException {
      while (benutzt) wait();
      System.out.println("Gabel " + welche + " aufgenommen");
      benutzt = true;
  }

  synchronized void hinlegen(){
    benutzt = false;
    System.out.println("Gabel " + welche + " hingelegt");
  }
}
```

```
public class Philosoph extends Thread {
  private int wer;
  private static int n = 5;
  private static Gabel [] gabel = new Gabel [n];
  private static Philosoph [] phil = new Philosoph [n];
  Philosoph (int i){wer = i;}
  public void run (){
      System.out.println("Philosoph " + wer + " gestartet");
      try{
      while (true){
        System.out.println("Philosoph " + wer + " denkt");
        sleep((long)(10000*Math.random())); // Denken
        System.out.println("Philosoph " + wer + " hungrig");
        gabel[(wer==0)?n-1:wer-1].aufnehmen();
        gabel[wer].aufnehmen();
        System.out.println("Philosoph " + wer + " isst");
        sleep((long)(10000*Math.random())); // Essen
        gabel[(wer==0)?n-1:wer-1].hinlegen();
        gabel[wer].hinlegen();
      }
    } catch(InterruptedException e){}
  }
  public static void main(String[] args) {
    for (int i=0; i<n; i++){
      gabel[i] = new Gabel (i);
      phil[i] = new Philosoph(i);
    }
    for (int i=0; i<n; i++){
      phil[i].start();
    }
  }
}
```

Diese Lösung ist allerdings verklemmungsbedroht! Es kann passieren, dass jeder Philosoph individuell seine linke gabel nimmt (damit liegt keine Gabel mehr auf dem Tisch) und dann wartet, bis er die rechte nehmen kann (was aber nie passieren wird). Allgemein ist eine *Verklemmung* (deadlock) ein zyklischer Wartezustand: A wartet auf B, B wartet auf C, …, Y wartet auf Z und Z wartet auf A. Ein Applet, um diesen Effekt auszuprobieren, findet sich z.B. unter http://www.doc.ic.ac.uk/~jnm/concurrency/classes/Diners/Diners.html





Ein etwas besserer Algorithmus vermeidet das Verklemmungsproblem. Jeder Philosoph macht folgendes:

```
wiederhole immer wieder
    denke
    wiederhole solange bis beide Gabeln genommen wurden:
        warte, bis linke Gabel frei, dann nimm sie
        falls rechte Gabel nicht da, gib linke wieder frei
        warte, bis rechte Gabel frei, dann nimm sie
        falls linke Gabel nicht da, gib rechte wieder frei
    iss
    gib beide Gabeln wieder frei
```

Obwohl diese Lösung nachweislich verklemmungsfrei ist, hat sie einen anderen gravierenden Nachteil: Es ist möglich, dass sich einzelne oder alle Philosophen endlos in einer sinnlosen Beschäftigung verlieren, in dem sie abwechselnd die linke und rechte Gabel aufnehmen und wieder ablegen. Solch eine Situation nennt man manchmal Endlosschleife (livelock); formal ist das eine Situation, in der intern immer dieselben Handlungen ausgeführt werden, ohne dass nach außen irgend ein Fortschritt erkennbar wäre.

Ein noch besserer Algorithmus vermeidet dieses Problem, indem durch einen geeigneten Synchronisationsalgorithmus das Aufnehmen beider Gabeln simultan erfolgt. Jeder Philosoph macht also folgendes:

```
wiederhole immer wieder
    denke
    warte, bis beide Gabeln frei, dann nimm sie (beide gleichzeitig)
    iss
    gib beide Gabeln wieder frei (und teile dies ggf. den Nachbarn mit)
```

Das simultane Aufnehmen der beiden Gabeln kann in Java dadurch programmiert werden, dass ein Monitor den Zugriff auf die Gabeln regelt.
Die Lösung hat jedoch immer noch eine Schwäche: Es kann sein, dass zwei Philosophen sich zusammentun, um den zwischen ihnen sitzenden „auszuhungern“: Philosoph 2 ist hungrig, aber erst isst Philosoph 1 (und 2 wartet auf die linke Gabel), und dann Philosoph 3 (und 2 wartet auf die rechte Gabel). Allgemein ist ein System *aushungerungsfrei* (starvation free), wenn garantiert ist, dass jeder kontinuierlich fortsetzungswillige Prozess auch irgendwann fortgesetzt wird. Aushungerungsfreiheit ist ein spezieller Fall der so genannten *Fairness*, die garantiert, dass kein Prozess immer wieder benachteiligt wird. Allgemeine Fairness-Eigenschaften lassen sich programmtechnisch nur schwer garantieren; pragmatische Lösungsmöglichkeiten bestehen darin

- vor dem Aufnehmen der Gabel(n) eine zufällig bestimmte Zeitdauer zu warten (keine garantierte Aushungerungsfreiheit, wird aber in Kommunikationsprotokollen so gelöst)
- einen unabhängigen Aushungerungs-Erkennungsalgorithmus einzusetzen („Wachhund“, watchdog)
- die Symmetrie zwischen den Philosophen zu brechen und z.B. eine feste Reihenfolge der Zuteilung vorzugeben, oder
- ein Wartenummernverfahren (ähnlich wie auf Behörden-Wartezimmern) einzuführen.