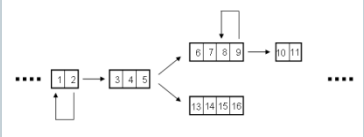
Freie Universität Berlin

Kurs: Programmieren in Java

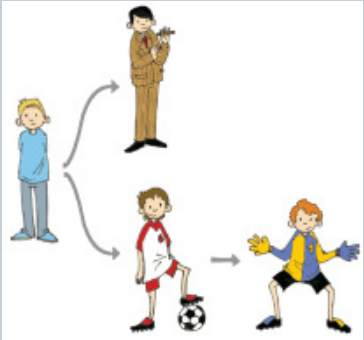
Tag 2



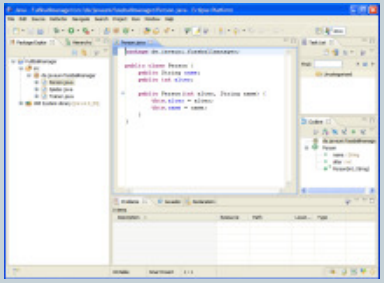
GRUNDLAGEN

GRAFIKKONZEPTE
BILDVERARBEITUNG
MUSTERERKENNUNG

KI UND SPIELE-
PROGRAMMIERUNG



OBJEKTORIENTIERTE
PROGRAMMIERUNG



ENTWICKLUNGS-
UMGEBUNGEN

Daten laden und speichern

Inhalt:

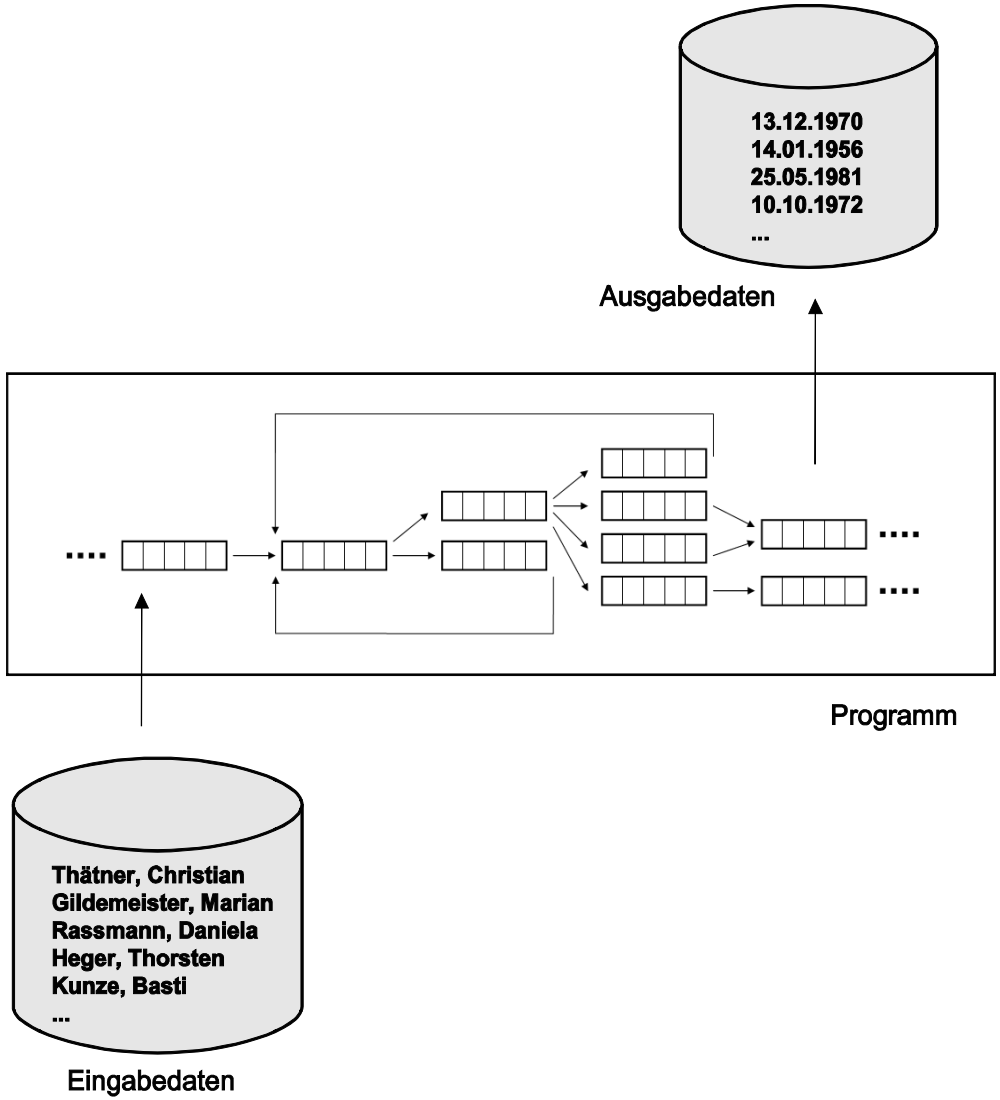
- Externe Programmeingaben
- Daten laden und speichern
- Daten von der Konsole einlesen

Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007

Daten laden und speichern

Freie Universität Berlin

Datenbanken



Daten laden und speichern

Externe Programmeingaben I

Die einfachste Methode, einem Programm ein paar Daten mit auf dem Weg zu geben, ist die Übergabe von Parametern auf der Kommandozeile.

```
// MirIstWarm.java
public class MirIstWarm{
   public static void main(String[] args){
      System.out.println("Mir ist heute zu warm, ich mache nix :).");
   }
}
```

Mit der Ausgabe:

```
C:\>javac MirIstWarm.java

C:\>java MirIstWarm
Mir ist heute zu warm, ich mache nix :).
```

Daten laden und speichern

Externe Programmeingaben II

Die einfachste Methode, einem Programm ein paar Daten mit auf dem Weg zu geben, ist die Übergabe von Parametern auf der Kommandozeile:

```
// MeineEingaben.java
public class MeineEingaben{
   public static void main(String[] args){
      System.out.println("Eingabe 1: >"+args[0]+"< und");
      System.out.println("Eingabe 2: >"+args[1]+"<");
   }
}
```

Mit der Eingabe:

```
C:\>javac MeineEingaben.java
C:\>java MeineEingaben Hallo 27
Eingabe 1: >Hallo< und
Eingabe 2: >27<
```

In diesem Fall war es wichtig zu wissen, wie viele Eingaben wir erhalten haben. Sollten wir auf einen Eintrag in der Stringliste zugreifen, die keinen Wert erhalten hat, dann passiert folgendes:

```
C:\>java MeineEingaben Hallo
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 1
at MeineEingaben.main(MeineEingaben.java:5)
```

Externe Programmeingaben III

Dieser Fehler lässt sich vermeiden, wenn wir zuerst die Anzahl der übergebenen Elemente überprüfen.

```
public class MeineEingaben{
   public static void main(String[] args){
      for (int i=0; i<args.length; i++)
         System.out.println("Eingabe "+i+": >"+args[i]+"<");
   }
}
```

Beispiel:

```
C:\>java MeineEingaben Hallo 27 und noch viel mehr!
Eingabe 0: >Hallo<
Eingabe 1: >27<
Eingabe 2: >und<
Eingabe 3: >noch<
Eingabe 4: >viel<
Eingabe 5: >mehr!<
```

Daten laden und speichern

Aktuelles Lernziel

Ein Lernziel an dieser Stelle wird sein, die im ersten Augenblick unverständlich erscheinenden Programmteile, einfach mal zu verwenden.

Wir müssen am Anfang einen Mittelweg finden zwischen dem absoluten Verständnis für jede Programmzeile und der Verwendung einer gegebenen Teillösung.

Daten laden und speichern

Freie Universität Berlin

Daten aus einer Datei einlesen I

```
import java.io.*;
public class LiesDateiEin{
   public static void main(String[] args){
      // Dateiname wird übergeben
      String filenameIn = args[0];
      try{
         FileInputStream fis    = new FileInputStream(filenameIn);
         InputStreamReader isr  = new InputStreamReader(fis);
         BufferedReader bur     = new BufferedReader(isr);

         // die erste Zeile wird eingelesen
         String sLine = bur.readLine();

         // lies alle Zeilen aus, bis keine mehr vorhanden sind
         // und gib sie nacheinander aus
         // falls von vornherein nichts in der Datei enthalten
         // ist, wird dieser Programmabschnitt übersprungen
         int zaehler = 0;
         while (sLine != null) {
            System.out.println("Zeile "+zaehler+": "+sLine);
            sLine = bur.readLine();
            zaehler++;
         }
         // schließe die Datei
         bur.close();
      } catch (IOException eIO) {
         System.out.println("Folgender Fehler trat auf: "+eIO);
      }
   }
}
```

Daten laden und speichern

Daten aus einer Datei einlesen II

Verwenden könnten wir **LiesDateiEin.java**, z.B. mit der Datei **namen.dat**. Den Inhalt einer Datei kann man sich auf der Konsole mit dem Befehl **type** anschauen:

```
C:\>type namen.dat
Harald Liebchen
Gustav Peterson
Gunnar Heinze
Paul Freundlich

C:\>java LiesDateiEin namen.dat
Zeile 0: Harald Liebchen
Zeile 1: Gustav Peterson
Zeile 2: Gunnar Heinze
Zeile 3: Paul Freundlich
```

Unser Programm kann eine Datei zeilenweise auslesen und gibt das eingelesene gleich auf der Konsole aus.

Daten laden und speichern

Daten in eine Datei schreiben I

Um Daten in eine Datei zu speichern, schauen wir uns mal folgendes Programm an:

```
import java.io.*;
public class SchreibeInDatei{
   public static void main(String[] args){
      // Dateiname wird übergeben
      String filenameOutput = args[0];
      try{
         BufferedWriter myWriter =
            new BufferedWriter(new FileWriter(filenameOutput, false));

         // schreibe zeilenweise in die Datei filenameOutput
         myWriter.write("Hans Mueller\n");
         myWriter.write("Gundel Gaukel\n");
         myWriter.write("Fred Feuermacher\n");

         // schliesse die Datei
         myWriter.close();
      } catch (IOException eIO) {
         System.out.println("Folgender Fehler trat auf: "+eIO);
      }
   }
}
```

Jetzt testen wir unser Programm und verwenden zur Überprüfung die vorher besprochene Klasse **LiesDateiEin.java**.

Daten in eine Datei schreiben II

Ausgabe:

```
C:\>java SchreibeInDatei namen2.dat

C:\>java LiesDateiEin namen2.dat
Zeile 0: Hans Mueller
Zeile 1: Gundel Gaukel
Zeile 2: Fred Feuermacher
```

Wir stellen fest: Es hat funktioniert!

Daten laden und speichern

Daten von der Konsole einlesen

Wir sind nun in der Lage, Daten beim Programmstart mitzugeben und Dateien auszulesen, oft ist es aber wünschenswert eine Interaktion zwischen Benutzer und Programm zu haben. Beispielsweise soll der Benutzer eine Entscheidung treffen oder eine Eingabe machen.

Das ist mit der Klasse **BufferedReader** schnell realisiert:

```
import java.io.*;
public class Einlesen{
   public static void main(String[] args){
      System.out.print("Eingabe: ");
      try{
         InputStreamReader isr   = new InputStreamReader(System.in);
         BufferedReader bur      = new BufferedReader(isr);

         // Hier lesen wir einen String ein:
         String str = bur.readLine();

         // und geben ihn gleich wieder aus
         System.out.println(str);
      } catch(IOException e){}
   }
}
```

Beispiel:

```
C:\>java Einlesen
Eingabe: Ich gebe etwas ein 4,5 a 1/2
Ich gebe etwas ein 4,5 a 1/2
```

Einfache Datenstrukturen

Inhalt:

- Arrays
- Matrizen
- Conways „Game of Life“

Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007

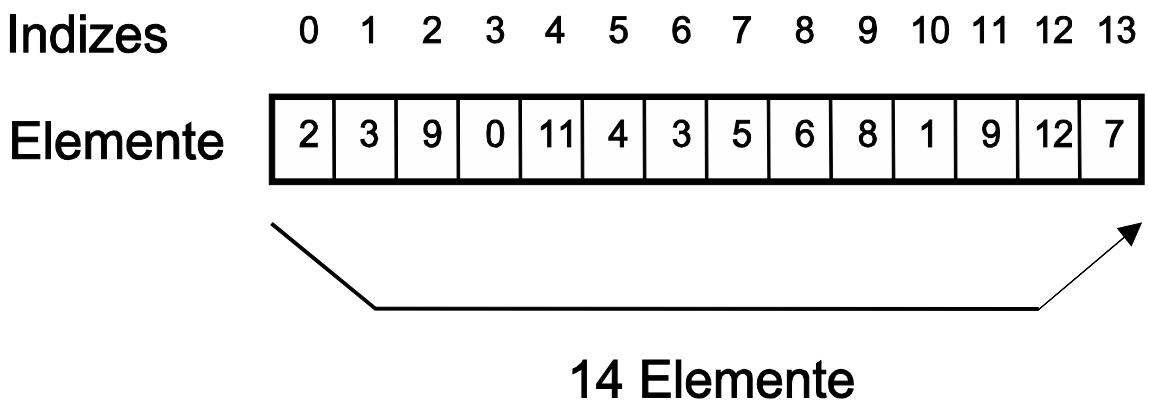
Einfache Datenstrukturen

Listen von Datentypen: Arrays

Nehmen wir an, wir möchten nicht nur einen **int**, sondern viele davon verwalten. Dann könnten wir es, mit dem uns bereits bekannten Wissen, in etwa so bewerkstelligen:

```
int a, b, c, d, e, f;
a=0;
b=1;
c=2;
d=3;
e=4;
f=5;
```

Sehr aufwendig und unschön. Es gibt eine einfachere Möglichkeit mit dem **Array** (Liste).



Einfache Datenstrukturen

Erzeugung von Arrays

Erzeugen eines int-Arrays mit k Elementen:

<Datentyp>[] <name>;
<name> = new <Datentyp>[k];

Oder in einer Zeile:

<Datentyp>[] <name> = new <Datentyp>[k];

Zugriff auf die Elemente und Initialisierung der Variablen:

```
int[] a = new int[2];
a[0]    = 3;
a[1]    = 4;
```

a = [3, 4]

Sollten wir schon bei der Erzeugung des Arrays wissen, welchen Inhalt die Elemente haben sollen, dann können wir das so vornehmen („Literale Erzeugung“):

```
int[] a  = {1,2,3,4,5};
```

a = [1, 2, 3, 4, 5]

Beliebte Fehlerquelle bei Arrays

Daran müssen wir uns gewöhnen und es ist eine beliebte **Fehlerquelle**. Es könnte sonst passieren, dass wir z.B. in einer Schleife alle Elemente durchlaufen möchten, auf das -te Element zugreifen und einen Fehler verursachen:

```
int[] a = new int[10];

for (int i=0; i<=10; i++)
   System.out.println("a["+i+"]="+a[i]);
```

Wir erhalten folgende Fehlermeldung:

```
C:\Java>java Array
a[0]=0
a[1]=0
a[2]=0
a[3]=0
a[4]=0
a[5]=0
a[6]=0
a[7]=0
a[8]=0
a[9]=0
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
at Array.main(Array.java:5)
```

Einfache Datenstrukturen

Freie Universität Berlin

Wir erinnern uns:

Auch beim Einlesen von der Konsole gab es diese Fehlermeldung:

```
// MeineEingaben.java
public class MeineEingaben{
   public static void main(String[] args){
      System.out.println("Eingabe 1: >"+args[0]+"< und");
      System.out.println("Eingabe 2: >"+args[1]+"<");
   }
}
```

Mit der Eingabe:

```
C:\>java MeineEingaben Hallo
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 1
at MeineEingaben.main(MeineEingaben.java:5)
```

Einfache Datenstrukturen

Vereinfachte for-Schleifennotation

Um Fehler zu vermeiden gibt es seit Java 1.5 die folgende vereinfachte for-Schleifennotation:

for (<Typ> <Variablenname> : <Ausdruck>)
<Anweisung>;

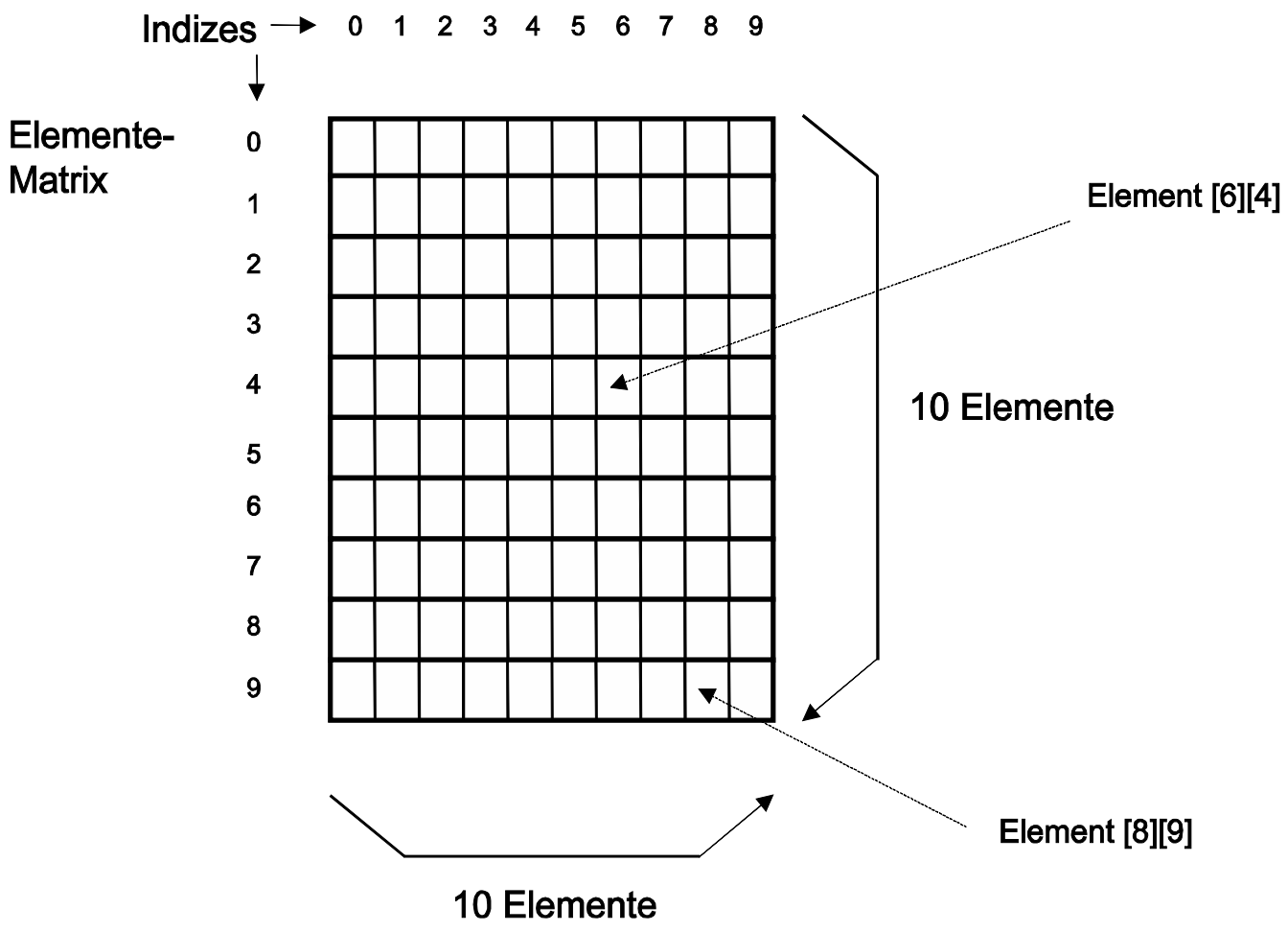
Lies: „Für jedes **x** aus der Liste **werte**“:

```
int[] werte = {1,2,3,4,5,6};    // Literale Erzeugung

// Berechnung der Summe
int summe = 0;
for (int x : werte)
   summe += x;
```

Hilft „IndexOutOfBoundsExceptions“ zu vermeiden.

Einfache Datenstrukturen

Matrizen und multidimensionale Arrays



Einfache Datenstrukturen

Matrizen und multidimensionale Arrays

Wir erzeugen Matrizen, indem wir zum Array eine Dimension dazu nehmen:

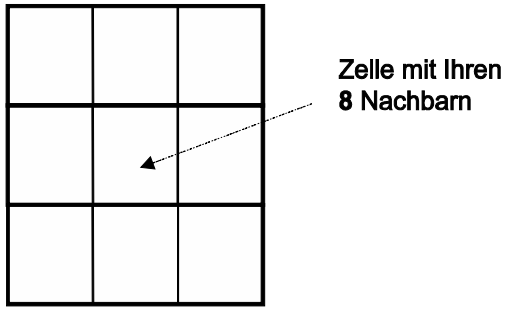
```
int[][] a = new int[n][m];
a[4][1]   = 27;
```

Auf diese Weise können wir sogar noch mehr Dimensionen erzeugen:

```
int[][][][] a = new int[k][l][m][n];
```

Conways Game of Life I

Man stelle sich vor, die Welt bestünde nur aus einer 2-dimensionalen Matrix. Jeder Eintrag, wir nennen ihn jetzt mal Zelle oder Zellulärer Automat, kann zwei Zustände annehmen, er ist entweder lebendig oder tot. Jede Zelle interagiert mit ihren 8 Nachbarn.



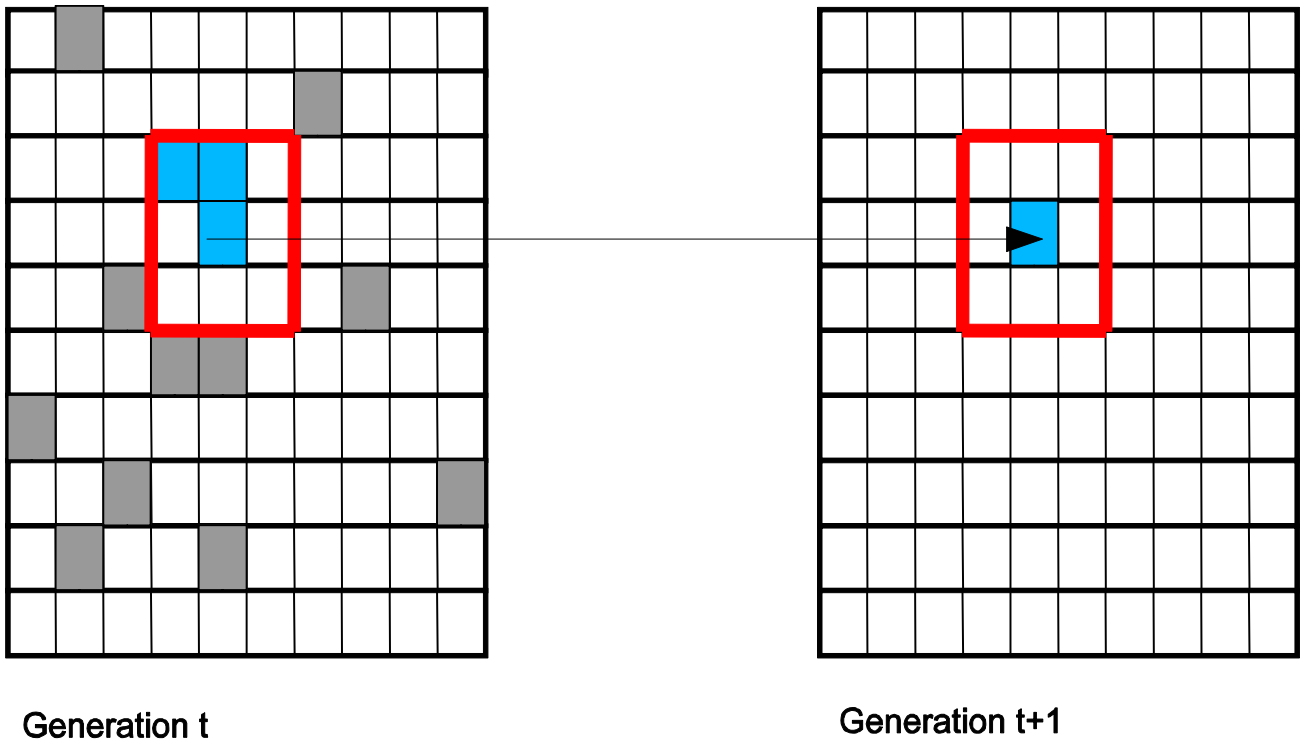
Diese Interaktion unterliegt den folgenden vier Regeln:

1. jede lebendige Zelle, die weniger als zwei lebendige Nachbarn hat, stirbt an Einsamkeit
2. jede lebendige Zelle mit mehr als drei lebendigen Nachbarn stirbt an Überbevölkerung
3. jede lebendige Zelle mit mit zwei oder drei Nachbarn fühlt sich wohl und lebt weiter
4. jede tote Zelle mit genau drei lebendigen Nachbarn wird wieder zum Leben erweckt.

Conways Game of Life II

Die zweite Generation wird durch die Anwendung der vier Regeln auf jede der Zellen erzeugt. Es wird geprüft, ob Zellen lebendig bleiben, sterben oder neu entstehen.

Einfache Implementierung:



Conways Game of Life III

Das Programm **GameOfLife.java** beinhaltet neben der main-Funktion zwei weitere Methoden.

Zum einen eine kleine Ausgabefunktion:

```
import java.util.Random; // erläutern wir später
public class GameOfLife{

   public static void gebeAus(boolean[][] m){
      // Ein "X" symbolisiert eine lebendige Zelle
      for (int i=0; i<10; i++){
         for (int j=0; j<10; j++){
            if (m[i][j])
               System.out.print("X ");
            else
               System.out.print("  ");
         }
         System.out.println();
      }
   }
```

Conways Game of Life IV

und zum anderen eine Funktion, die die Elemente der Nachbarschaften zählt. Dazu verwenden wir einen kleinen Trick:

```
// Wir nutzen hier die Tatsache aus, dass Java einen Fehler erzeugt,
// wenn wir auf ein Element außerhalb der Matrix zugreifen
public static int zaehleUmgebung(boolean[][] m, int x, int y){
   int ret = 0;
   for (int i=(x-1);i<(x+2);++i){
      for (int j=(y-1);j<(y+2);++j){
         try{
            if (m[i][j])
               ret += 1;
         }
         catch (IndexOutOfBoundsException e){}
      }
   }
   // einen zuviel mitgezaehlt?
   if (m[x][y])
      ret -= 1;

   return ret;
}
```

Conways Game of Life V

In der main-Methode werden zwei Matrizen für zwei aufeinander folgende Generationen bereitgestellt. Exemplarisch werden die Zellkonstellationen einer Generation, gemäß den zuvor definierten Regeln, berechnet und ausgegeben.

```
public static void main(String[] args){
      // unsere Welt soll aus 10x10 Elemente bestehen
      boolean[][] welt     = new boolean[10][10];
      boolean[][] welt_neu = new boolean[10][10];

      // ************************************************************
      // Erzeugt eine zufällige Konstellation von Einsen und Nullen
      // in der Matrix welt. Die Chancen liegen bei 50%, dass eine
      // Zelle lebendig ist.
      Random generator = new Random();
      double zufallswert;
      for (int i=0; i<10; i++){
         for (int j=0; j<10; j++){
            zufallswert = generator.nextDouble();
            if (zufallswert>=0.5)
               welt[i][j] = true;
         }
      }
      // ************************************************************
```

Einfache Datenstrukturen

Freie Universität Berlin

Conways Game of Life VI

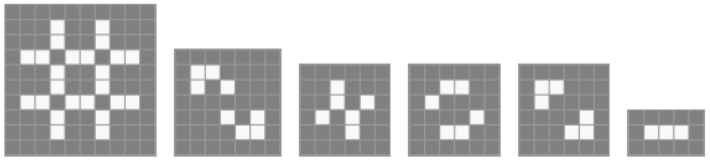
```
      // ***********************************************************
      // Ausgabe der ersten Generation
      System.out.println("Generation 1");
      gebeAus(welt);

      int nachbarn;
      for (int i=0; i<10; i++){
         for (int j=0; j<10; j++){
            // Zaehle die Nachbarn
            nachbarn = zaehleUmgebung(welt, i, j);

           if (welt[i][j]){
               // Regel 1, 2:
               if ((nachbarn<2) || (nachbarn>3))
                  welt_neu[i][j] = false;

               // Regel 3:
               if ((nachbarn==2) || (nachbarn==3))
                  welt_neu[i][j] = true;
            }
            else {
               // Regel 4:
               if (nachbarn==3)
                  welt_neu[i][j] = true;
            }
         }
      }
      // Ausgabe der zweiten Generation
      System.out.println("Generation 2");
      gebeAus(welt_neu);
   }
}
```

Methode zur Ermittlung der Anzahl der Nachbarn wird aufgerufen.

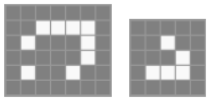
Auswahl besonderer Muster

Für den interessierten Leser ist hier eine kleine Sammlung besonderer Zellkonstellationen zusammengestellt:

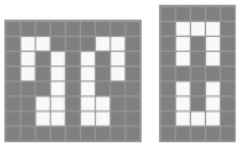
Zyklische Muster



Gleiter



Interessante Startkonstellationen



Debuggen und Fehlerbehandlungen

Inhalt:

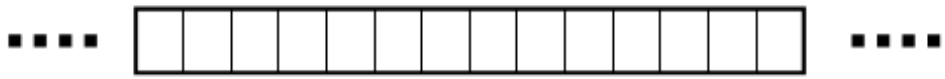
- Das richtige Konzept
- Exceptions
- Zeilenweises Debuggen und Breakpoints

Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007

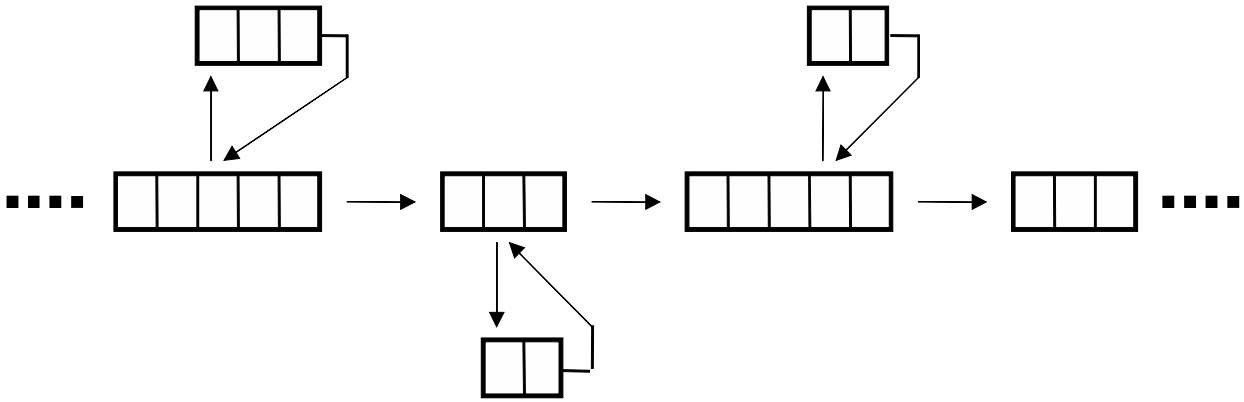
Debuggen und Fehlerbehandlungen

Das richtige Konzept

Es sollte bei der Entwicklung darauf geachtet werden, dass nicht allzu viele Programmzeilen in eine Funktion gehören.

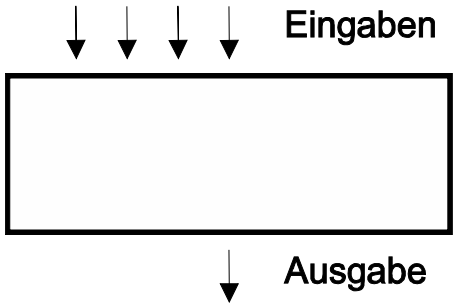


Besser ist es, das Programm zu gliedern und in Programmabschnitte zu unterteilen. Zum einen wird damit die Übersicht gefördert und zum anderen verspricht die Modularisierung den Vorteil, Fehler in kleineren Programmabschnitten besser aufzuspüren und vermeiden zu können.



Constraints I

Ein wichtiges Werkzeug der Modularisierung stellen sogenannte Constraints (Einschränkungen) dar. Beim Eintritt in einen Programmabschnitt (z.B. eine Funktion) müssen die Eingabeparameter überprüft und damit klargestellt werden, dass der Programmteil mit den gewünschten Daten arbeiten kann. Das gleiche gilt für die Ausgabe.



Constraints II

Kommentare sind hier unverzichtbar und fördern die eigene Vorstellung der Funktionsweise eines Programmabschnitts. Als Beispiel schauen wir uns mal die Fakultätsfunktion an:

```
public class Fakultaet{
   /*
     Fakultätsfunktion liefert für i=1 .. 19 die entsprechenden
     Funktionswerte i! = i*(i-1)*(i-2)*...*1

     Der Rückgabewert liegt im Bereich 1 .. 121645100408832000

     Sollte eine falsche Eingabe vorliegen, so liefert das Programm
     als Ergebnis -1.
   */
   public static long fakultaet(long i){
      // Ist der Wert ausserhalb des erlaubten Bereichs?
      if ((i<=0)||(i>19))
         return -1;

      // Rekursive Berechnung der Fakultaet
      if (i==1)
         return 1;
      else
         return i*fakultaet(i-1);
   }

   public static void main(String[] args){
      for (int i=0; i<15; i++)
         System.out.println("Fakultaet von "+i+" liefert "+fakultaet(i));
   }
}
```

Debuggen und Fehlerbehandlungen

Exceptions in Java I

Wenn ein Fehler während der Ausführung eines Programms auftritt, wird ein Objekt einer Fehlerklasse Exception) erzeugt. Da der Begriff Objekt erst später erläutert wird, stellen wir uns einfach vor, dass ein Programm gestartet wird, welches den Fehler analysiert und wenn der Fehler identifizierbar ist, können wir dieses Programm nach dem Fehler fragen und erhalten einen Hinweis, der Aufschluss über die Fehlerquelle gibt.

Schauen wir uns ein Beispiel an:

Die Zahl in einer Zeichenkette wird in einen Wert umgewandelt.

```
public class ExceptionTest{
   public static void main(String[] args){
      int d = Integer.parseInt(args[0]);
      int k = 10/d;
      System.out.println("Ergebnis ist "+k);
   }
}
```

Auf den ersten Blick ist kein Fehler erkennbar, aber ein Test zeigt schon die Fehleranfälligkeit des Programms.

Exceptions in Java II

Auf den ersten Blick ist kein Fehler erkennbar, aber ein Test zeigt schon die Fehleranfälligkeit des Programms:

```
C:\>java ExceptionTest 2
Ergebnis ist 5

C:\>java ExceptionTest 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionTest.main(ExceptionTest.java:5)

C:\>java ExceptionTest d
Exception in thread "main" java.lang.NumberFormatException: For input
string: "d"
       at java.lang.NumberFormatException.forInputString(Unknown Source)
       at java.lang.Integer.parseInt(Unknown Source)
       at java.lang.Integer.parseInt(Unknown Source)
       at ExceptionTest.main(ExceptionTest.java:4)
```

Zwei Fehlerquellen sind hier erkennbar, die Eingabe eines falschen Typs und die Eingabe einer 0, die bei der Division einen Fehler verursacht. Beides sind für Java wohlbekannte Fehler, daher gibt es auch in beiden Fällen entsprechende Fehlerbezeichnungen **NumberFormatException** und **ArithmeticException**.

Exceptions in Java III

Um nun Fehler dieser Art zu vermeiden, müssen wir zunächst auf die Fehlersuche gehen und den Abschnitt identifizieren, der den Fehler verursacht. Wir müssen uns dazu die Frage stellen: In welchen Situationen (unter welchen Bedingungen) stürzt das Programm ab? Damit können wir den Fehler einengen und den Abschnitt besser lokalisieren. Dann müssen wir Abhängigkeiten überprüfen und die Module identifizieren, die für die Eingabe in diesen Abschnitt zuständig sind.

Angenommen, wir haben einen Bereich lokalisiert und wollen diesen nun beobachten, dazu verwenden wir die **try-catch-Klausel**.

```
try {
   <Anweisung>;
   ...
   <Anweisung>;
} catch (Exception e) {
   <Anweisung>;
}
```

Die try-catch Behandlung lässt sich als *„Versuche dies, wenn ein Fehler dabei auftritt, mache das.“* lesen.

Exceptions in Java IV

Um unser Programm **ExceptionTest .java** vor einem Absturz zu bewahren, wenden wir diese Klausel an und testen das Programm:

```
public class ExceptionTest2{
   public static void main(String[] args){
      try{
         int d = Integer.parseInt(args[0]);
         int k = 10/d;
         System.out.println("Ergebnis ist "+k);
      } catch(Exception e){
         System.out.println("Fehler ist aufgetreten...");
      }
   }
}
```

Wie testen nun die gleichen Eingaben:

```
C:\>java ExceptionTest 2
Ergebnis ist 5

C:\>java ExceptionTest 0
Fehler ist aufgetreten...

C:\>java ExceptionTest d
Fehler ist aufgetreten...
```

Einen Teilerfolg haben wir nun schon zu verbuchen, da das Programm nicht mehr abstürzt. Der Bereich in den geschweiften Klammern nach dem Schlüsselwort **try** wird gesondert beobachtet.
Sollte ein Fehler auftreten, so wird die weitere Abarbeitung innerhalb dieser Klammern abgebrochen und der Block nach dem Schlüsselwort **catch** ausgeführt.

Exceptions in Java IV

Dummerweise können wir nun die Fehler nicht mehr eindeutig identifizieren, da sie die gleiche Fehlermeldung produzieren. Um die Fehler aber nun eindeutig abzufangen, lassen sich einfach mehrere catch-Blöcke mit verschiedenen Fehlertypen angeben:.

```
try {
      <Anweisung>;
      ...
      <Anweisung>;
    } catch(Exceptiontyp1 e1){
      <Anweisung>;
    } catch(Exceptiontyp2 e2){
      <Anweisung>;
    } catch(Exceptiontyp3 e3){
      <Anweisung>;
}
```

Exceptions in Java V

Wenden wir die neue Erkenntnis auf unser Programm an:

```
public class ExceptionTest3{
   public static void main(String[] args){
      try{
         int d = Integer.parseInt(args[0]);
         int k = 10/d;
         System.out.println("Ergebnis ist "+k);
      } catch(NumberFormatException nfe){
         System.out.println("Falscher Typ! Gib eine Zahl ein ...");
      } catch(ArithmeticException ae){
         System.out.println("Division durch 0! ...");
      } catch(Exception e){
         System.out.println("Unbekannter Fehler aufgetreten ...");
      }
   }
}
```

Bei den schon bekannten Eingaben liefert das Programm nun folgende Ausgaben:

```
C:\>java ExceptionTest3 2
Ergebnis ist 5

C:\>java ExceptionTest3 0
Division durch 0! ...

C:\>java ExceptionTest3 d
Falscher Typ! Gib eine Zahl ein ...
```

Fehlerhafte Berechnungen aufspüren I

Beim Programmieren wird leider ein nicht unwesentlicher Teil der Zeit mit dem Aufsuchen von Fehlern verbracht. Das ist selbst bei sehr erfahrenen Programmierern so und gerade, wenn die Projekte größer und unübersichtlicher werden, sind effiziente Programmiertechniken, die Fehler vermeiden, unabdingbar. Sollte sich aber doch ein Fehler eingeschlichen haben, gibt es einige Vorgehensweisen, die die zum Auffinden benötige Zeit auf ein Mindestmaß reduzieren.

Beispiel: Berechnung der Zahl PI

Gottfried Wilhelm Leibniz gab 1682 für die Näherung von PI/4 eine Berechnungsvorschrift an, die als **Leibniz-Reihe** bekannt ist. Am Ende der Berechnung müssen wir das Ergebnis also noch mit 4 multiplizieren, um eine Näherung für PI zu erhalten.

Die Vorschrift besagt:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots$$

Fehlerhafte Berechnungen aufspüren II

Der Nenner wird also immer um 2 erhöht, während das Vorzeichen jeden Schritt wechselt. Eine Schleife bietet sich zur Berechnung an:

```
public class BerechnePI{
   static int max_iterationen = 100;
   static public double pi(){
      double PI = 0;
      int vorzeichen = 1;
      for (int i=1; i<=max_iterationen*2; i+=2){
         PI += vorzeichen*(1/i);
         vorzeichen *= -1;
      }
      return 4*PI;
   }

   public static void main(String[] args){
      double PI = pi();
      System.out.println("Eine Naeherung fuer PI ist "+PI);
   }
}
```

Nach 100 Iterationen stellen wir fest:

```
C:\JavaCode>java BerechnePI
Eine Naeherung fuer PI ist 4.0
```

Es scheint sich ein Fehler eingeschlichen zu haben...

Fehlerhafte Berechnungen aufspüren III

Um den Fehler aufzuspüren, versuchen wir die Berechnungen schrittweise nachzuvollziehen. Überprüfen wir zunächst, ob das Vorzeichen und der Nenner für die Berechnung stimmen. Dazu fügen wir in die Schleife folgende Ausgabe ein:

```
...
for (int i=1; i<max_iterationen; i+=2){
   System.out.println("i:"+i+" vorzeichen:"+vorzeichen);
   PI += vorzeichen*(1/i);
   vorzeichen *= -1;
}
...
```

Als Ausgabe erhalten wir:

```
C:\JavaCode>java BerechnePI
i:1 vorzeichen:1
i:3 vorzeichen:-1
i:5 vorzeichen:1
i:7 vorzeichen:-1
...
```

Das Vorzeichen alterniert, ändert sich also in jeder Iteration. Das ist korrekt.

Fehlerhafte Berechnungen aufspüren IV

Die Ausgabe erweitern wir, um zu sehen, wie sich PI im Laufe der Berechnungen verändert:

```
...
System.out.println("i:"+i+" vorzeichen:"+vorzeichen+" PI:"+PI);
...
```

Wir erhalten:

```
C:\JavaCode>java BerechnePI
i:1 vorzeichen:1 PI:0.0
i:3 vorzeichen:-1 PI:1.0
i:5 vorzeichen:1 PI:1.0
i:7 vorzeichen:-1 PI:1.0
...
```

Vor dem ersten Schritt hat PI den Initialwert 0. Nach dem ersten Schritt ist PI=1. Soweit so gut. Allerdings ändert sich PI in den weiteren Iterationen nicht mehr. Hier tritt der Fehler zum ersten Mal auf.

Fehlerhafte Berechnungen aufspüren V

Werfen wir einen Blick auf die Zwischenergebnisse:

```
...
double zwischenergebnis = vorzeichen*(1/i);
System.out.println("i:"+i+" Zwischenergebnis:"+zwischenergebnis);
...
```

Jetzt sehen wir, an welcher Stelle etwas schief gelaufen ist:

```
C:\JavaCode>java BerechnePI
i:1 Zwischenergebnis:1.0
i:3 Zwischenergebnis:0.0
i:5 Zwischenergebnis:0.0
i:7 Zwischenergebnis:0.0
...
```

Der Fehler ist zum Greifen nah! Die Berechnung **vorzeichen*(1/i)** liefert in jedem Schritt, außer dem ersten, den Wert 0.0 zurück.

Das Problem kennen wir bereits!!!

Die Berechnung **1/i** liefert immer das Ergebnis **0**, da sowohl **1** als auch **i** vom Typ **int** sind. Der Compiler verwendet die ganzahlige Division, was bedeutet, dass alle Stellen nach dem Komma abgerundet werden. Dieses Problem lässt sich leicht lösen, indem wir die **1** in **1.d** ändern und damit aus dem implizit angenommenen **int** ein **double** machen. Eine andere Möglichkeit wäre das Casten von **i** zu einem **double**.

Fehlerhafte Berechnungen aufspüren VI

Ändern wir das Programm entsprechend:

```
public class BerechnePI{
   static int max_iterationen = 100;
   static public double pi(){
      double PI = 0;
      int vorzeichen = 1;
      for (int i=1; i<=max_iterationen*2; i+=2){
         PI += vorzeichen*(1/(double)i);
         vorzeichen *= -1;
      }
      return 4*PI;
   }

   public static void main(String[] args){
      double PI = pi();
      System.out.println("Eine Naeherung fuer PI ist "+PI);
   }
}
```

In beiden Fällen liefert das korrigierte Programm eine gute Näherung:

```
C:\JavaCode>java BerechnePI
Eine Naeherung fuer PI ist 3.1315929035585537
```

Ein etwas geübter Programmierer erahnt einen solchen Fehler bereits beim Lesen des Codes. Das strategisch günstige Ausgeben von Zwischenergebnissen und Variablenwerten ist jedoch auch ein probates Mittel zum Auffinden von schwierigeren Fehlern. Wird das Programm jedoch größer, ist ein Code von anderen Programmieren beteiligt und sind die Berechnungen unübersichtlich, dann helfen die von den meisten Programmierumgebungen bereitgestellten Debugger enorm.

Debuggen und Fehlerbehandlungen

Vielen Dank fürs Zuhören!

Das genügt erstmal für heute ...

Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*", Springer-Verlag 2007