Vorlesung 05: Vermischtes

Peter Thiemann

Universität Freiburg, Germany

SS 2010

Inhalt

Arrays

Arrays

- Ein Array (*Feld*, *Reihung*) ist eine spezielle Art Objekt, in dem beliebige Werte (die *Elemente*) vom gleichen Typ (*homogen*) indiziert abgelegt werden können.
- Ein Array besitzt eine *feste Größe*, die der Anzahl der Elemente entspricht. Sie ist im Feld `length` abgelegt.
- Ein Array wird mit den natürlichen Zahlen von 0 bis Größe−1 indiziert.
- Ein Array wird mit einer Variante des **new**-Operators erzeugt.
- Ein Array besitzt keine Methoden, sondern der Zugriff erfolgt über spezielle Syntax.
- Ein Array besitzt einen speziellen Arraytyp, der vom Typ der Elemente abhängt. Der Arraytyp kann für Felder, Parameter und lokale Variable benutzt werden.

Verwendung von Arrays

Vektoren, Matrizen, Messreihen, Puffer, Stringverarbeitung, usw

Deklaration und Erzeugung von Arrays

- Einige Array-Deklarationen:

```
int[] lottoNumbers;
String[] winners;
Employee[] emp;
```

- Der **new**-Operator erzeugt Arrays, unter Angabe des Elementtyps und der Anzahl der Elemente (alle auf 0 initialisiert).

```
lottoNumbers = new int[6];
winners = new String[100];
emp = new Employee[1000];
```

- Abfrage der Länge

```
lottoNumbers.length // == 6
```

Zugriff auf die Array-Elemente

- Lesen der Elemente

```
lottoNumbers[0] // kleinstmöglicher Index
lottoNumbers[5] // größtmöglicher Index: lottoNumbers.length−1
lottoNumbers[1] + lottoNumbers[2]
```

- Zuweisen auf Arrayelemente

```
winners[3] = "Max Frei";
emp[365] = new Employee (...);
```

Verarbeitung von Arrays

Akkumulation

- Typisches Muster: Durchlaufe alle Elemente eines Arrays und führe dabei eine akkumulierende Berechnung durch
- Beispiel: Durchschnitt einer Stichprobe

```
static double average (double[] values) {
  double sum = 0; // Akkumulator
  // Initialisierung
  int i = 0; // Laufvariable
  // Schleifentest
  while (i < values.length) {
    sum = sum + values[i];
    // Inkrementieren der Laufvariable
    i = i + 1;
  }
  return sum / values.length;
}
```

Verarbeitung von Arrays

Akkumulation

- Typisches Muster: Durchlaufe alle Elemente eines Arrays und führe dabei eine akkumulierende Berechnung durch
- Beispiel: Durchschnitt einer Stichprobe

```
static double average (double[] values) {
  double sum = 0; // Akkumulator
  // Initialisierung
  int i = 0; // Laufvariable
  // Schleifentest
  while (i < values.length) {
    sum = sum + values[i];
    // Inkrementieren der Laufvariable
    i = i + 1;
  }
  return sum / values.length;
}
```

- Diese Kombination tritt so häufig auf, dass sie fest eingebaut ist.

Die **for**-Anweisung

- Die **for**-Anweisung

  for(*initialisierung*; *bedingung*; *ausdruck*){*anweisungen*}

  führt zuerst die *initialisierung* durch, testet dann die *bedingung*, falls diese `true` ist, führt sie erst die *anweisungen* und zum Schluss den *ausdruck* aus. Dann wird die *bedingung* erneut getestet, usw.

- Beispiel: die Methode auf der vorangegangenen Folie mit **for**

```
static double average (double[] values) {
  double sum = 0; // Akkumulator
  for (int i = 0; i < values.length; i = i + 1) {
    sum = sum + values[i]; // Kombination
  }
  return sum / values.length;
}
```

Gleiches Muster: Maximumsberechnung

- Aufgabe: Bestimme den Maximalwert einer Messreihe
- Gleiches Muster: Verwende Akkumulator mit dem bisherigen Maximalwert und der max Operation zur Kombination.

```
static double maximum (double[] values) {
  double cand = Double.NEGATIVE_INFINITY; // Akkumulator
  for (int i = 0;
       i < values.length;
       i = i + 1) {
    cand = Math.max(cand, values[i]); // Kombination
  }
  return cand;
}
```

Exkurs: Muster für Akkumulation

Inkrement und Dekrement

- Beim Durchlaufen eines Arrays wird immer eine Laufvariable hochgezählt (oder auch heruntergezählt):

```
i = i + 1; // Inkrementieren
j = j - 1; // Dekrementieren
```

- Hierfür spezielle Ausdrücke, die den gleichen Effekt haben.

```
i++;
j--;
```

Exkurs: Muster für Akkumulation

Akkumulieren

- Beim Akkumulieren eines Wertes während einer Schleife wird oft zu einer Akkumulator-Variable ein neuer Wert mit einer Operation hinzugefügt:

```
sum = sum + values[i];
prod = prod * values[i];
```

- Hierfür spezielle Ausdrücke, die den gleichen Effekt haben.

```
sum += values[i];
prod *= values[i];
```

- Mit den meisten binären Operatoren möglich.

Exkurs: Muster für Akkumulation

Beispiel: `average`

... unter Verwendung der neuen Operatoren:

```
static double average (double[] values) {
  double sum = 0; // Akkumulator
  for (int i = 0; i < values.length; i++) {
    sum += values[i]; // Kombination
  }
  return sum / values.length;
}
```

Verarbeitung von Arrays

Muster: Lineare Suche

- Aufgabe: Durchsuche ein Array nach einer vorgegebenen Zahl
- Implementierung

```
// suche x im Array values, liefere Position oder −1 falls nicht gefunden
static int search (int x, int[] values) {
  for (int i = 0; i < values.length; i = i + 1) {
    if (x == values[i]) {
      return i;
    }
  }
  return −1;
}
```

Verarbeitung von Arrays

Elemente vertauschen

- Aufgabe: Vertausche die Elemente *i* und *j* in einem Array
- Implementierung

```
// vertausche die Einträge i und j in a
// Voraussetzung: 0 <= i,j < a.length
static void swap (int[] a, int i, int j) {
  int t = a[i];
  a[i] = a[j];
  a[j] = t;
  return;
}
```

Verarbeitung von Arrays

Elementreihenfolge umdrehen (spiegeln)

```
// change input to its mirror image
static void mirror (int[] a) {
  int len = a.length;
  for (int i = 0; i < len / 2; i = i + 1) {
    swap (a, i, len − i − 1);
  }
  return;
}
```

$a =$ | 1 | 2 | 3 | 4 | 5 | $\Rightarrow$`mirror` $(a)\Rightarrow$ $a =$ | 5 | 4 | 3 | 2 | 1 |

Mehrdimensionale Arrays

- Die Elemente eines Arrays können selbst Arrays sein.
- Deklaration einer zweidimensionalen Matrix

```
double [][] matrix;
```

- Anlegen einer zweidimensionalen Matrix

```
matrix = new double[10][10];
```

- Zugreifen und Ändern

```
mneu[i][k] = matrix[i][j] * matrix[j][k];
```

Mehrdimensionale Arrays: Fallstrick

- Achtung: eine 2D-Matrix wird in Java als Array von Arrays dargestellt.
- (Nicht sehr effizient . . .)
- Die innereren Arrays müssen nicht notwendigerweise die gleiche Größe haben!
- Beispiel für legalen Code

```
double[][] matrix = new double[2][2];
// matrix[0].length == 2 && matrix[1].length == 2
matrix[0] = new double[1];
// matrix[0].length != matrix[1].length
```

- Vermeiden!

Mehrdimensionale Arrays

Zeilen und Spalten vertauschen

- Zeilen i und j vertauschen

```
double [] tmp = matrix[i];
matrix[i] = matrix[j];
matrix[j] = tmp;
```

- Spalten i und j vertauschen

```
for (int k = 0; k < matrix[i].length; k++) {
  double tmp = matrix[k][i];
  matrix[k][i] = matrix[k][j];
  matrix[k][j] = tmp;
}
```

Verarbeitung von zweidimensionalen Arrays

Durchlaufen beider Dimensionen und Akkumulation

Die Maximumsnorm einer Matrix ist das Maximum ihrer Einträge.

```
static double maxnorm (double [][] matrix) {
  double mx = 0;
  for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
      mx = Math.max (mx, matrix[i][j]);
    }
  }
}
```

$\Rightarrow$ Zwei Dimensionen, zwei geschachtelte Schleifen

Verarbeitung von zweidimensionalen Arrays

Matrix als Ergebnis

Das Skalarprodukt multipliziert jeden Eintrag einer Matrix mit einem Wert.

```
static double[][] scalarprod (double s, double [][] matrix) {
  double [][] result = new double[matrix.length][1];
  for (i = 0; i < matrix.length; i++) {
    int m = matrix[i].length;
    result[i] = new double[m];
    for (j = 0; j < m; j++) {
      // Eigentliche Operation zur Bestimmung eines Wert in der Matrix
      result[i][j] = s * matrix[i][j];
    }
  }
  return result;
}
```

Alternative

```
static void scalarprod (double s, double[][] matrix) {...}
```

Anwendungsbeispiel: Alternative Implementierung von Entry Listen

Erinnerung

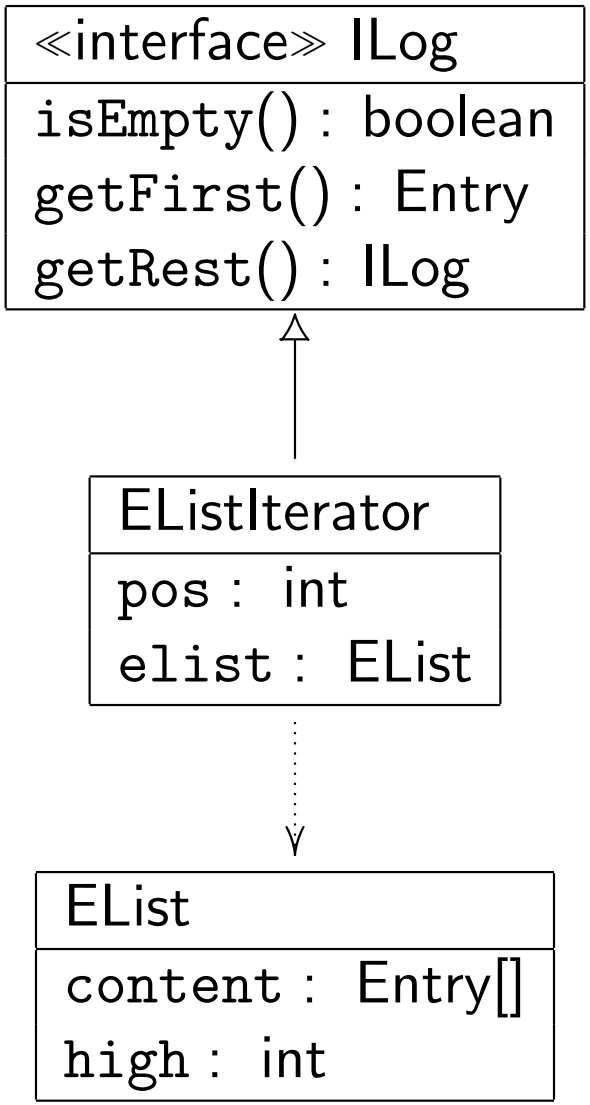
Das Interface ILog enthält das Durchlaufinterface für Listen von Entry:

```
interface ILog {
  ...
  // teste ob diese Liste leer ist
  boolean isEmpty();
  // liefere das erste Element, falls nicht leer
  Entry getFirst();
  // liefere den Rest der Liste, falls nicht leer
  ILog getRest();
  ...
}
```

Aufgabe

Implementiere dieses Durchlaufinterface mit Hilfe von Arrays.

Version 1: "Listen" mit fester Maximallänge



Implementierung EListIterator

```
// Zustand eines Durchlaufs einer EList
class EListIterator implements ILog {
    private int pos;
    private EList elist;
    // private Standardkonstruktor weggelassen
    EListIterator (EList elist) {
        this.pos = 0;
        this.elist = elist;
    }
    public boolean isEmpty () {
        return this.elist.isEmptyAt (this.pos);
    }
    public Entry getFirst() {
        return this.elist.elementAt (this.pos);
    }
    public ILog getRest () {
        return new EListIterator (this.pos + 1, this.elist);
    }
}
```

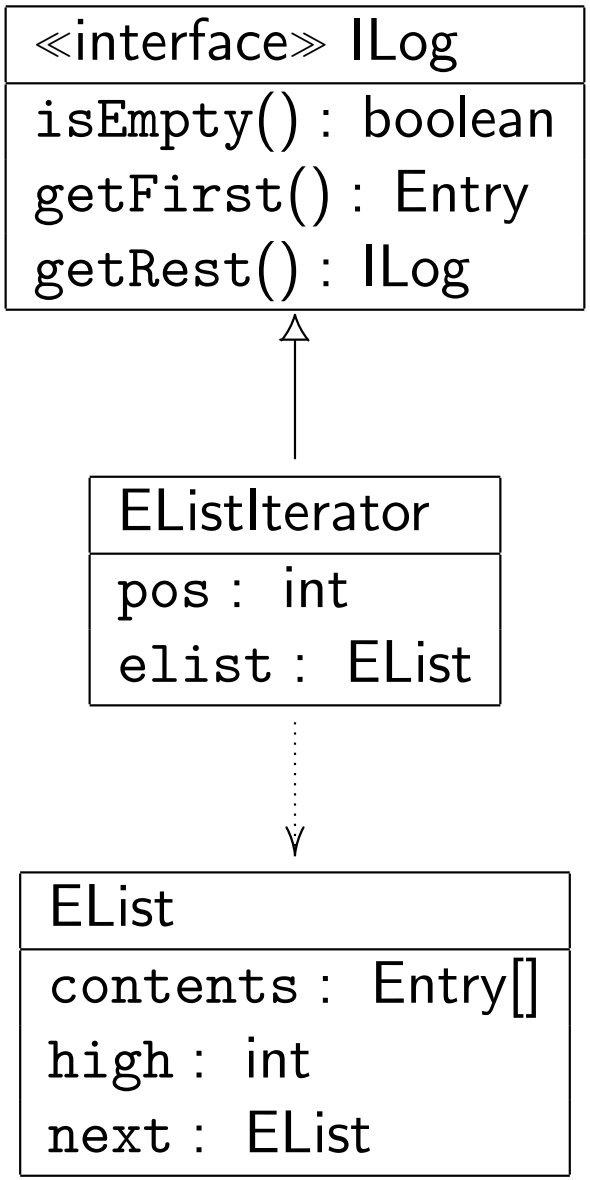
Implementierung EList

```
// Liste von Entry mit Längenbeschränkung
class EList {
    private Entry[] contents;
    private int high; // high <= contents.length

    // teste ob an pos noch Einträge vorhanden sind
    public boolean isEmptyAt (int pos) {
        return pos >= this.high;
    }

    // liefere Element an Position
    public Entry elementAt (int pos) {
        return this.contents[pos];
    }
}
```

Version 2: “Listen” ohne Längenbeschränkung



Version 2: Implementierung

```
// Liste von Entry ohne Längenbeschränkung
class EList {
    private Entry[] contents;
    private int high; // high <= contents.length
    private EList next;

    // teste ob an pos noch Einträge vorhanden sind
    public boolean isEmptyAt (int pos) {
        return (pos >= this.high)
            && ((this.next == null) || this.next.isEmptyAt (pos - this.high));
    }

    // liefere Element an Position
    public Entry elementAt (int pos) {
        if (pos < this.high) {
            return this.contents[pos];
        } else {
            return this.next.elementAt (pos - this.high);
        }
    }
}
```

Main

Statische Felder und Methoden

- Neben den normalen Feldern und Methoden kann eine Klasse *statische Felder* und *statische Methoden* besitzen.
  (In anderen Sprachen: *Klassenfelder* bzw. *Klassenmethoden*)
- Beide sind *unabhängig* von Objekten und können gelesen, geschrieben und aufgerufen werden, ohne dass ein Objekt der Klasse beteiligt ist.
- Der Zugriff erfolgt mit

```
Klasse.feldname // statisches Feld von Klasse
Klasse.methode (arg...) // statische Methode von Klasse
```

- Beispiel: Die Javabibliothek definiert eine Klasse `Math`, die spezielle Konstanten (e und π) als statische Felder zur Verfügung stellt und trigonometrische und andere Funktionen als statische Methoden bereithält.

```
Math.E, Math.PI
Math.min(4, 5), Math.max(−1, 1), Math.sin(Math.PI / 4)
```

Statische Felder und Methoden

Beispiel

Statische Felder können für Buchhaltungsaufgaben über alle Objekte einer Klasse verwendet werden.

Eine Klasse soll mitzählen, wie oft ihr Konstruktor aufgerufen worden ist.

```
class CountedStuff {
  private static int count = 0;
  private static int inc() {
    return count++;
  }
  private int serial;
  public CountedStuff () {
    this.serial = CountedStuff.inc ();
  }
  public int getSerial () {
    return this.serial;
  }
}
```

Statische Felder und Methoden

Beispiel (Fortsetzung)

```
> CountedStuff x1 = new CountedStuff();
> x1.getSerial()
0
> CountedStuff x2 = new CountedStuff();
> x2.getSerial()
1
> CountedStuff x3 = new CountedStuff();
> x3.getSerial()
2
```

Das Hauptprogramm

- Das Hauptprogramm kann in einer beliebigen Klasse definiert werden.
- Die Klasse ist gekennzeichnet durch eine statische Methode `main` mit folgender Deklaration

```
public static void main (String [] arg) {
  ...
}
```

- Das String-Arrays enthält dabei die Parameter des Programmaufrufs (z.B. auf der Kommandozeile). Der Aufruf

  `java MyClass eins zwei drei`

  bewirkt, dass im Rumpf von `main`

```
arg.length == 3
arg[0].equals ("eins")
arg[1].equals ("zwei")
arg[2].equals ("drei")
```

Einfache Ausgabe

- Das Objekt `System.out` stellt Methoden zur Ausgabe auf die Konsole bereit.
- Es besitzt (überladene) Methoden `print` für alle primitiven Typen, die jeweils ihr Argument ausdrucken.

```
void print(boolean b)
void print(double d)
void print(int i)
void print(String s)
```

- Die gleichermaßen überladenen Methoden `println` drucken ihr Argument gefolgt von einem Zeilenvorschub.

```
void println() // nur Zeilenvorschub
void println(boolean x)
void println(double x)
void println(int x)
void println(String x)
```

Beispiel: main mit Ausgabe

```
class HelloWorld {
  public static void main (String[] args) {
    System.out.print ("Hello world,");
    for (int i = 0; i < args.length; i++) {
      System.out.print (" " + args[i]);
    }
    System.out.println ();
  }
}
```

Druckt `Hello world,` gefolgt von allen Kommandozeilenargumenten und abgeschlossen mit einem Zeilenvorschub.

Dateien

Dateien

- Bisher gingen alle Ausgaben auf die Konsole.
- Alternative: Ausgabe auf Datei, wo die Information dauerhaft erhalten bleibt (**Persistenz**).
- Grundlegende Eigenschaften von Dateien:

  Dateiname: Eine Zeichkette.
  Inhalt (Daten): Beliebige Informationen (Zeichenfolge)

Grundlegende Operationen auf Dateien

- Erzeugen einer Datei
- In eine Datei schreiben.
- Aus einer Datei lesen.
- Eine Datei löschen.
- Den Dateinamen ändern.

Die Klasse `File`

- Java definiert in der Package `java.io` eine Klasse `File`.
- Einer der Konstruktoren für `File` nimmt als Argument einen Dateinamen. Beispiel:

```
File f1 = new File("/etc/passwd");
File f2 = new File("/home/joe/.mail");
```

- Hinweis: Ein `File`-Objekt repräsentiert nur einen Pfad (Dateinamen), der nicht unbedingt mit einer Datei assoziiert sein muss.

```
boolean joeHasMail = f2.exists();
// true, falls die Datei /home/joe/.mail existiert
```

Weitere Infomethoden siehe `java.io.File`.

Dateien manipulieren

Einige Methoden von `File`

```
// Datei löschen
void delete ();
// Datei umbenennen
void renameTo (File newname);
// Datei lesbar?
boolean canRead();
// Datei schreibbar?
boolean canWrite ();
```

uva.

Ausgabe in Dateien

- Java verwendet *Ströme* um Dateien zu lesen und zu schreiben.
- Zur Ausgabe dient die Klasse `FileOutputStream`,
- Der Konstruktor von FileOutputStream akzeptiert als Argument ein `File`-Objekt.
- Die Datei mit dem durch das Argument gegebenen Namen wird geöffnet.
- Ist die Datei nicht vorhanden, so wird sie erzeugt.
- Ist die Datei vorhanden, wird ihr Inhalt gelöscht.
- Beispiel:

```
File path = new File ("mysecrets");
FileOutputStream fs = new FileOutputStream (path);
```

FileOutputStream

- Ein `FileOutputStream` ist sehr primitiv. Er stellt nur Operationen zum Schreiben von einzelnen Bytes (oder Arrays von Bytes) zur Verfügung.
- Gewünscht ist aber eine `print` oder `println`-ähnliche Schnittstelle!
- Diese sind Methoden der Klasse `PrintStream`, die unter Rückgriff auf einen Ausgabestrom (wie `FileOutputStream`) implementiert ist.
- Daher akzeptiert der Konstruktor von `PrintStream` einen `FileOutputStream`.

Schreiben in eine Datei

```
import java.io.*;
class WriteMySecrects {
  public static void main (String[] args) throws IOException {
    File path = new File ("mysecrets");
    FileOutputStream fs = new FileOutputStream (path);
    PrintStream output = new PrintStream (fs);
    output.println("you win!");
    output.close();
  }
}
```

IOException

- `throws IOException` deutet an, dass der Rumpf von `main` eine *Exception* werfen kann.
- Dies muss in Java explizit angegeben oder behandelt werden (später).
- Eine Exception zeigt das Vorliegen eines Fehlers an.
- Der Konstruktor von `FileOutputStream` signalisiert durch `FileNotFoundException`, falls die Datei nicht zum Schreiben geöffnet werden kann.
- Die `close` Methode kann einen allgemeinen Ein-/Ausgabefehler signalisieren.

Packages

Packages

- Java organisiert Klassen in *Packages*.
- Eine Package enthält eine Menge von Java-Klassen und Interfaces.
- Der Name einer Package besteht aus durch Punkt getrennten Bezeichnern. Dieser sollte möglichst weltweit einzigartig sein.
- Beispiele für Packagenamen: `java.lang`, `java.util`, `java.io`
- Der **volle Namen** einer Klasse (Interface) besteht aus dem Packagenamen gefolgt vom Klassennamen.
- Beispiele: `java.lang.String`, `java.util.Date`, `java.io.InputStream`

Packages verwenden

- Wenn eine Package bekannt ist, so kann jederzeit mit dem vollen Namen auf die darin befindlichen Klassen zugegriffen werden
- Beispiel:

```
java.lang.String s = "ein string";
de.unifr.informatik.Myclass x = new de.unifr.informatik.Myclass ();
```

Packages importieren

- Zur Abkürzung kann einfach der Klassennamen verwendet werden, wenn die Klasse zu Beginn der Datei *importiert* worden ist. Alle Klassen in der Package `java.lang` werden automatisch importiert.
- Beispiel

```
import de.unifr.informatik.Myclass;
class Application {
  Myclass x = new Myclass ();
  ...
}
```

- Ein $*$ anstelle eines Klassennamens importiert alle Klassen und Interfaces einer Package:

```
import java.util.*;
...
Date d = new Date(); // voller Name: java.util.Date
Random r = new Random (); // voller Name: java.util.Random
```

Packages definieren

- Wenn eine oder mehrere Klassen von mehreren Anwendungen verwendet werden sollen, so müssen sie in Packages zusammengefasst werden.
- Jede Klasse, die zu einer Package gehört beginnt mit einer Package-Deklaration (vor allen Importen und vor allen Klassendefinitionen).
- Beispiel:

```
package de.unifr.informatik;
```

- Die Klassen müssen dann vom Java-Compiler in Bytecode übersetzt werden und in einer dem Package-Namen entsprechenden Verzeichnisstruktur abgelegt werden.
- Wenn die Wurzel dieser Verzeichnisstruktur im CLASSPATH erwähnt wird, so können andere Klassen via `import` auf die Klassen der Package zugreifen.