Einführung in die Programmierung WS 12-13

Lernskript

Dozent:
Dr. Hildebrandt

LaTeX von:
Sven Bamberger

Zuletzt Aktualisiert:
15. Februar 2013

Dieses Skript wurde erstellt, um sich besser auf die Klausur vorzubereiten.

Dieses Dokument garantiert weder Richtigkeit noch Vollständigkeit, da es aus Mitschriften gefertigt wurde und dabei immer Fehler entstehen können. Falls ein Fehler enthalten ist, bitte melden oder selbst korrigieren und neu hoch laden.

Inhaltsverzeichnis

1 Vorlesungen

1.1 Java-Editionen

- Java läuft auf sehr unterschiedlichen Systemen
- Wird in verschiedenen „Packungsgrößen“ angeboten
  - EE (enterprise edition): große Unternehmensserver
  - SE (standard edition): Desktop-Systeme
  - ME (micro edition): Handys, PDAs, Embedded Systems
- Editionen unterscheiden sich in den mitgelieferten Zugaben,nicht in der Programmiersprache

1.2 Java-Entwicklungssystem

- Zum Schreiben neuer Programm ist der Compiler javac nötig, zum Ausführen fertiger Programm nicht
- Java für verschiedene Einsatzzwecke:
  - JRE (java runtime environment): zum Ausführen fertiger Programme
  - JDK (java development kit): JRE + Compiler und weitere Hilfsmittel zum Schreiben neuer Programme

1.3 Namen in Java

- An vielen Stellen frei wählbare Namen = „Bezeichner“, „Identifier“
- Bestandteile: Große und kleine Buchstaben, Ziffern, Underscore (_)
- Erstes Zeichen darf keine Ziffer sein
- Etwa fünfzig reservierte Wörter dürfen nicht als Identifier benutzt werden (beispielsweise class, int, public)
- Beispiele:
  - Counter
  - colorDepth
  - Iso9660
  - XMLProcessor
  - MAX_VALUE
- Nicht erlaubt sind z.B.:
  - 1stTry (erster Buchstabe darf keine Ziffer sein)

  - Herz Dame (Leerzeichen im Namen nicht erlaubt)
  - const (reserviertes Wort)
  - muenchen-erding (Bindestrich im Namen nicht erlaubt)
- Übliche Konventionen für Java-Identifier:
  - Variablen, Methoden, primitive Typen: CamelCode, erster Buchstabe klein: counter, find1stToken, bottomUp
  - Referenztypen: CamelCode, erster Buchstabe groß: Hello, String, ServerSocket
  - Typvariablen (Generics): einzelne große Buchstaben: T, U
  - statische, öffentliche Konstanten: alle Buchstaben groß, Wortteile getrennt mit Underscore: MAX_VALUE, PI, RGB24

1.4 Regel Ebenen

1.4.1 Syntax (Rechtschreibung)

Verteilung von Semikolons, Klammern, Schreibweise von Namen

Syntaxfehler

Compiler meldet einen Fehler

1.4.2 Semantik (Bedeutung)

Zulässige Kombination von Sprachelementen

Semantikfehler

Compiler meldet einen Fehler, Programm verhält sich falsch, stürzt nach dem Start ab, ...

1.4.3 Pragmatik (Gebrauch)

Bewährte und sinnvolle Konstruktionen

Fehler der Pragmatik

Programm ist unleserlich, umständlich, unverständlich

1.5 Polymorphismus

- Der Typ des Ergebnisses, und u.U. der Wert, ist abhängig von den Typen der Operanden:
  $20/8 \rightarrow 2$
  $20.0/8.0 \rightarrow 2.5$
- Zwei Operanden gleichen Typs: Operandentyp = Ergebnistyp
- Gemischte Operandentypen: double-Ergebnis:
  $1 + 2 \rightarrow 3$ (int)
  $1.0 + 2 \rightarrow 3.0$ (double)

$1 + 2.0 \rightarrow 3.0$ (double)

$1.0 + 2.0 \rightarrow 3.0$ (double)

1.6 Explizite Typkonversionen

- Hohe Priorität, wie andere unäre Operatoren:
  $(\text{int})2.5 * 3 \rightarrow 2 * 3 \rightarrow 6$
  -$(\text{int})2.5 \rightarrow -2$
  Klammern hilft: $(\text{int})(2.5 * 3) \rightarrow (\text{int})(7.5) \rightarrow 7$
- ACHTUNG STOLPERFALLE
  $(\text{int})1e100 \rightarrow 2147483647$
- Typcasts auf ein Minimum beschränken.

1.7 Struktogramme

- Elementarbausteine von Struktogrammen: einfache Anweisungen
- Formulierung einzelner Anweisungen
- Beschreibungs- oder Darstellungsformen für Algorithmen:
  - Umgangssprache
    Problematisch: Mißverständnisse, Interpretationsmöglichkeiten, Sprachkenntnisse
  - Quelltext
    Nur mit Kennnis einer konkreten Programmiersprache lesbar
  - Neutrale, abstrakte Form
    Brauchbarer Kompromiss
- Populär: Struktogramme (=Nassi-Schneiderman-Diagramme)
- Früher auch: Flussdiagramme (flow charts), erlauben wirre Konstruktionen
- Ziel: Reduktion auf die Idee, die wesentlichen Strukturen

1.7.1 Umgangssprachlich

Definiere n als ganze Zahl
Gib n den Wert 4
Zähle n um 1 hoch
Gib n aus

1.7.2 Pseudocode

```
int n
n = 4
n = n + 1
print n
```

1.7.3 Nassi-Schneiderman

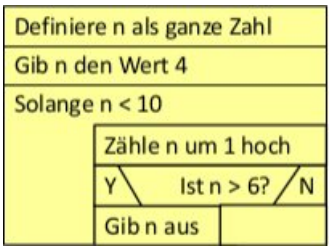

Abbildung 1.1: ein mini Struktogramm

1.8 While-Schleife: Euklidischer Algorithmus

```
1  class EuclidGCD
2  {
3      public static void main(String... args)
4      {
5          int m = Integer.parseInt(args[0]);
6          int n = Integer.parseInt(args[1]);
7          int r = m % n;
8          while (r != 0)
9          {
10             m = n;
11             n = r;
12             r = m % n;
13         }
14         System.out.println(n);
15     }
16 }
```

1.9 While-Schleife: Collatzfolge (3n + 1 – Folge)

$$z_{n+1} = \begin{cases} \frac{1}{2} z_n, & z_n \text{ gerade} \\ 3 \cdot z_n + 1, & z_n \text{ ungerade.} \end{cases}$$

```
class CollatzMax
{
    public static void main(String... args)
    {
        int z = Integer.parseInt(args[0]);
        int n = 0;
        int max = z;
        while (z != 1)
        {
            if (z%2 == 0){
                z = z/2;
            }
            else{
                z = 3*z + 1;
            }
            n++;
            if (z > max){
                max = z;
            }
        }
        System.out.println(n);
        System.out.println(max);
    }
}
```

1.10 Inkrement-/Dekrementoperator

Variable ++; Variable - -;

```
 int a = 1;
int b = a++; // b = 1, a = 2
int c = a--; // c = 2, a = 1
```

++Variable; - -Variable;

```
int a = 1;
int b = ++a; // b = 2, a = 2
int c = --a; // c = 1, a = 1
```

1.11 Der bedingte Operator

- Dreistelliger "bedingter Operator"(engl. "conditional operator")
- Syntax:
  condition? yes-expression: no-expression

- Beziehung zu if:
  variable = condition? yes-expression: no-expression;

- äquivalent zu:
  if (condition)
  . variable = yes-expression;
  else
  . variable = no-expression;

1.12 do-while

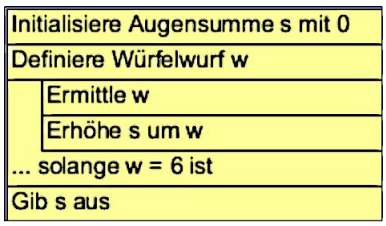

Abbildung 1.2: ein mini Struktogramm

```
int s = 0;
int w;
do{
    w = // wuerfeln ...
    s += w;
} while (w == 6);
System.out.println(s);
```

1.13 Break und Continue

1.13.1 Break

- Anweisung break beendet eine Schleife sofort der Rest des Rumpfes wird übersprungen
- break = einfache Anweisung (wie Definitionen, Wertzuweisungen)
- Zweck: Entscheidung über Fortsetzung einer Schleife fällt mitten im Rumpf

1.13.2 continue

- Anweisung continue startet sofort den nächsten Schleifendurchlauf der Rest des Rumpfes wird übersprungen
- Wie break: Nützlich zur Behandlung von Sonderfällen
- Zweck: Folge von Entscheidungen über Fortsetzung des Schleifendurchlaufes mitten im Rumpf

Achtung: break und continue spalten Kontrollfluss: mit Bedacht verwenden

1.14 Gültigkeitsbereiche

- Idee
  - Blöcke ... gruppieren Anweisungen
  - Innnerhalb eines Blocks alle Anweisungsarten erlaubt, auch Definitionen
  - Gültigkeitsbereich (engl. „scope“) einer Variablen...
    - beginnt mit der Definition und
    - endet mit dem Block, in dem die Definition steht
  - Außerhalb des Blocks: Variable gilt nicht
  - Gültigkeitsbereiche bezogen auf Quelltext, werden vom Compiler überprüft
  - Zur Laufzeit irrelevant
- Namenskollision
  - Gültigkeitsbereich umfasst untergeordnete (geschachtelte) Blöcke
  - Namenskollision: Definition des gleichen Namens, wie in einem umfassenden Block
  - Java: Doppelte Definition unzulässig
  - Aber: Kein Problem in disjunkten Blöcken:

1.15 for-Schleife

Jede For-Schleife kann durch eine While Schleife ersetzt werden.

```
1 for(int i = 0; i < 10; i++)
2     System.out.println(i);
```

1.16 Switch

- Der Wert der expression wird einmal berechnet.
- Das Ergebnis wird nacheinander mit den labels verglichen, bis zum ersten gleichen Wert.
- Die dem label nachfolgenden statements werden ausgeführt, bis zum break;
- Ziel: switch-Anweisungen ersetzen längere, unübersichtliche if-Kaskaden
- Syntax:

```
1  switch (expression)
2  {
3      case label1:
4          statement ...
5          break;
6      case label2:
7          statement ...
8          break;
9      ...
10 }
```

- switch-Rumpf = Gültigkeitsbereich
- Definitionen im switch-Rumpf gelten immer, nicht aber Initialisierungen
- Unübersichtlich: besser keine Definitionen im switch-Rumpf
- switch ist selbst eine Anweisung
  ⇒ kann in einem übergeordneten switch stehen
- Nützlich um unregelmäßige Tabellen zu implementieren
- Typ int als switch-Ausdruck zulässig
- Nicht zulässig:
  - double (Test von exakten Werten problematisch ⇒ Rundungsfehler)
  - boolean (nur zwei Werte)
  - ...
- Allgemein: ganzzahlige Typen und Aufzählungstypen

1.16.1 case-Label

- case-Labels müssen eindeutig sein, doppelte Werte unzulässig
- case-Labels müssen konstant (vom Compiler berechenbar) sein
- Das schließt Literale, Numerale, final-Variablen mit compiler-berechenbarem Wert und konstante Ausdrücke ein.
- Wenn kein case-Label passt, geschieht nichts (ganzes switch wirkt wie eine leere Anweisung)
- Mehrere (verschiedene) case-Labels vor einer Anweisungsfolge sind zulässig
- default = spezielles case-Label, passt auf alle übrigen Werte
- default darf nur einmal und nur am Ende genannt werden
- Jeder switch sollte mit einem default enden

```
1  int a = ...;
2  final int b = 3;
3  final double c = b*(b + 1);
4  switch(a)
5  {
6      case 3:
7      case 1 + 2:
8          System.out.println("yes");
9          break;
10
11     case (int)c%(b - 1):
12         System.out.println("maybe");
13         break;
14
15     default:
16         System.out.println("no");
17         break;
18 }
```

1.16.2 Fall through

- break beendet switch (zweite Anwendung von break neben Schleifen)
- Falls break fehlt, wird mit den Anweisungen des nächsten Zweiges fortgefahren (engl. „fall through“)
- Fall through selten sinnvoll, meistens ein Fehler, immer ein Stolperstein
- Wenn möglich eher nicht verwenden, Programm eher umschreiben

1.17 Klassen

Klassen definieren neue Typen

1.17.1 Klassennamen

- Klassen sind mit eindeutigen Identifiern benannt
- In der Regel englische Substantive, erster Buchstabe groß
- Syntax:

```
class Classname
{
    ...
}
```

- Konventionen:
  - Jede Klassendefinition in einer eigenen Quelltextdatei (erzwungen bei öffentlichen Klassen)
  - Dateiname = Klassenname + Extension .java

1.17.2 Objektvariablen

- Objektvariablen sind Variablen, ebenso wie bisher verwendete Variablen
- Zur begrifflichen Abgrenzung: bisher benutzte Variablen = lokale Variablen
- Definitionssyntax von Objektvariablen und lokalen Variablen gleich
- Aber: Ort der Definition unterschiedlich

Objektvariablen	... Elemente von Klassen
lokale Variablen	... Anweisungen in Methoden

- Benennung von Objektvariablen:
  - wie lokale Variablen
  - eindeutig innerhalb einer Klasse

Zugriff

- Jedes Objekt enthält die Objektvariablen, die in der Klassendefinition festgelegt sind
- Objektvariablen eines Objektes können einzeln angesprochen werden: Elementzugriff
- Objekt an das sich ein Elementzugriff richtet: Zielobjekt
- Syntax: Zielobjekt.Objektvariable

Umgang

- Elementzugriff spricht Objektvariablen innerhalb eines Objektes an
- Gleiche Verwendung wie lokale Variablen
- Beispiel: Zähler oder Nenner eines Rational-Objektes ...
  - in einem Ausdruck verwenden: int i = 5 - r.numer*3;
  - mit Operatorzuweisung und Inkrementoperator modifizieren:
    r.numer *= 10;
    r.numer++;
  - vergleichen: if(r.denom != 0) ...
  - usw...
- Nur Zugriffssyntax zeigt Unterschied zwischen Objektvariablen und lokalen Variablen

Objektvariablen unterschiedlicher Objekte

- Jedes Objekt hat eigene Objektvariablen (das ist der ganze Witz...)
- Elementzugriff richtet sich an eine Objektvariable innerhalb eines Objektes (des Zielobjektes), andere Objektvariablen des Zielobjektes und Objektvariablen anderer Objekte unberührt

1.17.3 Referenztypen

- Classname = neuer Typ, gleichberechtigt neben primitiven Typen
- int, double, boolean, char, byte, short, long sowie float sind primitive Typen
- Primitive Typen sind atomar, Bausteine spielen keine Rolle
- Gegensatz: Classname ist ein Referenztyp → enthält separate Bestandteile, diese können einzeln angesprochen und verarbeitet werden
- Alle Klassen definieren Referenztypen
- Auswahl primitiver Typen liegt fest, können nicht neu definiert werden
- Erster Nutzen von Klassen: bündeln ihre Bestandteile

1.17.4 Objekte / Instanzen

- Klassendefinition ≈ Bauplan, Konstruktionsvorschrift, Blaupause
- Objekte der Klasse müssen explizit geschaffen werden, entstehen nicht von alleine
- Objekt = Exemplar, Instanz
- 1 Klassendefinition — beliebig viele Objekte

1.17.5 Operator new

- Erzeugen eines neuen Objektes = instanziieren (auch „konstruieren“, „allokieren“)
- Syntaktisch mit Operator new.
- new produziert aus einer Klassendefinition ein einzelnes, neues Objekt dieser Klasse
- Mehrere Objekte ⇒ mehrere Aufrufe von new

```
1 new Classname();
```

1.17.6 Methoden

- Methoden werden in Klassen definiert, ebenso wie Objektvariablen
- Objektvariablen legen Eigenschaften („Attribute“) von Objekten fest, Methoden legen Operationen fest
- Anders formuliert: Objektvariablen beschreiben den Aufbau von Objekten, Methoden ihr Verhalten
- Methoden haben Namen, wie Objektvariablen (Achtung: eigener Namensraum, Methodennamen und Objektvariablennamen clashen nicht; aber Doppelbelegung beinahe nie sinnvoll)

```
1  class Rational
2  {
3      int numer;
4      int denom;
5
6      void print()
7      {
8      System.out.printf("%d/%d%n", numer, denom);
9      }
10 }
```

- Methodendefinition = Methodenkopf + Methodenrumpf
- Klammern im Rumpf sind Pflicht, auch bei einer (oder keiner) Anweisung
  - nicht außerhalb einer Klassendefinition
  - nicht innerhalb einer anderen Methodendefinition
- Anzahl, Reihenfolge und Anordnung von Methodendefinitionen in einer Klasse beliebig

Aufruf

- Methode wird mit Zielobjekt aufgerufen
- Ohne Zielobjekt kein Aufruf (Ausnahme: statische Methoden)
- Methodenaufruf syntaktisch ähnlich zu Elementzugriff: Zielobjekt.Methodenname();
- Runde Klammern markieren Methodenaufruf, fehlen bei Objektvariablenzugriff
- Ablauf eines Methodenaufrufs in mehreren Einzelschritten: Call-Sequence
- Aufrufendes Programm („Aufrufer“, engl. caller) unterbrechen

  - Werte aller Argumente von links nach rechts berechnen
  - Parameter erzeugen
  - Parameter mit Argumentwerten initialisieren
  - (Aufrufendes Programm („Aufrufer“) unterbrechen)
  - (Methodenrumpf durchlaufen)
  - Parameter zerstören
  - (Aufrufer nach dem Aufruf fortsetzen)

Methodenrumpf

- Methodenrumpf = Block
- Gültigkeitsbereich lokaler Definitionen = Methodenrumpf
- Lebensdauer lokaler Variablen: jeweils ein Aufruf (Gegensatz Objektvariablen: Lebensdauer wie Objekt)
- Zugriff auf Objektvariablen des eigenen Objektes ohne Angabe eines Zielobjekts
- Ebenso: Aufruf von Methoden des eigenen Objektes ohne Angabe eines Zielobjektes
- Methoden erreichen jede Objektvariable der eigene Klasse, unabhängig von der Anordnung der Definitionen

Namenskollision

- Namen von lokalen Variablen und Objektvariablen kollidieren nicht
- Vorteil: Benennung von lokalen Variablen ohne Rücksicht auf Objektvariablen
- Nachteil: Lokale Definition „verdeckt“ Objektvariable (oft unbeabsichtigt).
- In der Praxis unproblematisch: Zugriffe auf Objektvariablen sowieso besser auf einzelne Methoden beschränkt (Stichwort: Datenkapselung)

Parameterübergabe

- Argumente und Parameter vom Compiler bei jedem Aufruf paarweise abgeglichen
- Ein Argument pro Parameter erforderlich (zu viele oder zu wenige Argumente: wird nicht übersetzt; Ausnahme: Varargs)
- Nötig: Typ jedes Arguments kompatibel zum entsprechenden Parameter
- Beliebig komplizierte Ausdrücke als Argumente zulässig, werden erst ausgerechnet, dann übergeben
- Verwendung der Parameter im Methodenrumpf: vergleichbar mit automatisch initialisierten lokalen Variablen
- Parameter = dritte Art von Variablen, neben lokalen Variablen und Objektvariablen
- Liste von Parametern im Methodenkopf (Komma zwischen je zwei Parametern)

1.18 Methodenüberladung

- Überladen (engl. overloading) = mehrere Methoden mit gleichen Namen, aber unterschiedlichen Parameterlisten
- Sinnvoll für verwandte Methoden mit ähnlichem Zweck
- Überladen mit unterschiedlicher Parameteranzahl oder unterschiedlichen Parametertypen oder beidem
- Namen der Parameter ohne Bedeutung

1.18.1 Aufruf überladener Methoden

- overload resolution = Auswahl einer passenden überladenen Methode zu einer gegebenen Argumentliste — manchmal nicht ganz einfach!
- Zuerst: Alle in Frage kommenden Kandidaten sammeln, einschließlich der Anwendung impliziter Typkonversionen
- Unter den Kandidaten die Methode aufrufen, die am genauesten passt
- Was bedeutet: „passt am genauesten?
- Eine Methode a „passt genauer“ als eine Methode b, wenn jeder Aufruf von a auch von b, akzeptiert werden würde, aber nicht umgekehrt

1.18.2 Mehdeutige Aufrufe: Aufrufbeispiele

- Letzter Fall: Zwei Kandidaten

set(double, int)	nach Konversion des ersten Argumentes
set(int, double)	nach Konversion des zweiten Argumentes

- Jede der beiden Methoden akzeptiert Aufrufe, die die andere nicht akzeptiert. Keine passt genauer als die andere.
- Der Aufruf ist mehrdeutig — Fehler
- Methoden nur mit unterschiedlich vielen Parametern oder mit inkompatiblen Parametertypen überladen

1.19 Ergebnisrückgabe

- Mehrere return-Anweisungen im Rumpf erlaubt
- Methode kehrt zurück, sobald zur Laufzeit das erste return erreicht wird
- Statische Reihenfolge der return-Anweisungen unerheblich, konkreter Ablauf zur Laufzeit entscheidet

1.19.1 Idee

- Parameterübergabe transportiert Information vom Aufrufer zur Methode
- Ergebnisrückgabe liefert Information von der Methode zurück zum Aufrufer
- Eine Methode kann beliebig viele Parameterwerte annehmen, aber nur einen Ergebniswert liefern

1.19.2 Definition

Zwei gekoppelte Maßnahmen zur Ergebnisrückgabe:

- Typ des Ergebniswertes im Methodenkopf
- return-Anweisung im Methodenrumpf

1.19.3 Schema

```
type methodname(...)
{
    ...
    return expression;
}
```

- Typ von expression in der return-Anweisung kompatibel zu type im Methodenkopf
- Ausdruck „Methodentyp“ = Typ des Ergebniswertes der Methode
- Auch kurz: „Typ-Methode“ = Methode die ein Ergebnis des Typs liefert

1.20 Arrays

1.20.1 Motivation

- Arrays (auch „Feld“, „Reihung“) vordefiniert, ohne weitere Maßnahmen verfügbar
- Werden von praktisch allen Programmiersprachen angeboten
- Tief in Java verankert, von der JVM intern genutzt
- Arrays sind Containertypen: Speichern Elemente anderer Typen
- Elementtyp beliebig, aber gleich für alle Elemente
- Werte einzelner Elemente austauschbar
- Anzahl Elemente eines Arrays („Arraylänge“) unveränderlich

1.20.2 Arraytypen

- Arrays = Familie von ähnlichen Typen, kein einzelner Typ

Elementtyp	Arraytyp
int	int[]
boolean	boolean[]
char	char[]
String	String[]
Rational	Rational[]
OpenCounter	OpenCounter[]

- Arraytypen sind Referenztypen
- Ein Arraytyp legt keine Länge fest
- Ein konkretes Exemplar eines Arrays hat eine feste, unveränderliche Länge

1.20.3 Allokieren

Erzeugen eines neuen Arrays mit einer bestimmten Anzahl Elemente eines beliebigen typs.
Beispiele: Arrays mit 4 bzw. 36 Elementen:

```
1 int [] a = new int[4]
2 Rational[] b = new Rational[3*8+12]
```

a und b referenzieren Arrays welche mit null initialisiert werden.

- Elementanzahl wird zur Laufzeit beim new-Aufruf festgelegt, kann nachher nicht mehr verändert werden
- Elemente eines Arrays beim Allokieren automatisch mit Defaultwerten vorbesetzt (ebenso wie Objekt- und Klassenvariablen)
- Bildhafte Vorstellung: Array = Liste namenloser Variablen, werden gemeinsam definiert, bleiben für die Lebensdauer des Arrays beisammen
- Erzeugt nur das Array, keine Objekte, ruft keinen Element-Konstruktor auf

1.20.4 Arrayliterale

- Arrayliteral = Konstante eines Arraytyps
- Allokiert neues Array aus einer Liste vorgegebener Werte
- Schema:
  . new type[] expression, expression, ..., expression
- Länge der Arrays = Anzahl Elemente
- Beispiel:
  new int[] 71, -4, 7220, 0, 238
- Listenelemente = beliebige Ausdrücke, kompatibel zum Elementtyp des Arrays
- Sinnvoll, wenn Anzahl und Werte von Elementen im Quelltext bekannt

1.20.5 Elementzugriff

- Elemente eines Arrays folgen linear aufeinander
- Jedes Element hat ganzzahligen Index
- Index des ersten Elementes = 0, dann fortlaufend weiter
- Index des letzten Elementes = (Arraylänge − 1)
- Zugriff auf alle Element (ungefähr) gleich schnell = random access
- Schema für „Array-Elementzugriff“:
  . array[expression]
- Index zur Laufzeit berechnet aus int-Ausdruck expression
- Zugriff auf ein Element berührt die anderen Elemente des Arrays nicht
- Arrayelement benutzbar wie gewöhnliche Variable des Elementtyps

- Unzulässige Indexwerte werfen ArrayIndexOutOfBoundsException
- Negativer Index immer unzulässig
- JVM prüft zur Laufzeit jeden Array-Elementzugriff
- Anzahl Elemente eines Arrays (konzeptionell) beliebig
- Erlaubt Arrays mit einem und keinem Element
- Anzahl Elemente als öffentlich lesbare final-Objektvariable length
- Zugriff wie Objektvariablen in Objekten:
  array.length

1.20.6 Syntax

- Eckige Klammern syntaktisch in verschiedenen Kontexten
- Typangaben:
  Typ + leere eckige Klammern
  int[] a;
- Allokieren eines neuen Arrays:
  new + Typ + Anzahl Elemente in eckigen Klammern
  a = new int[5];
- Array-Literal:
  new + Typ + leere eckige Klammern + Liste von Elementen
  a = new int[] 1, 2, 3;
- Elementzugriff:
  Arrayausdruck + Index in eckigen Klammern
  a[1] = 23;

1.20.7 forech-Schleife

Kurzform einer for-Schleife für bestimmten Zweck
for (type variable: array)
statement

```
1  for (int e: a){
2      ...
3  }
```

Äquivalent zu:

```
1  for (int i = 0; i < a.length; i++){
2      int e = a[i];
3      ...
4  }
```

- Nur Lesen, kein Schreiben des Arrays
- Start immer mit erstem Element
- Sequentieller Durchlauf, keine Sprünge

- Nur ein Array, nicht mehrere parallel
- Durchlauf abbrechen nur mit break

Abbildungsverzeichnis