Einführung in die Programmierung WS 12-13

Lernskript

Dozent:
Dr. Hildebrandt

LaTeX von:
Sven Bamberger

Zuletzt Aktualisiert:
15. Februar 2013

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

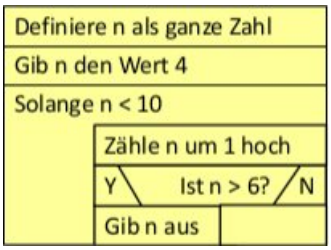

Abbildung 1.1: ein mini Struktogramm

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

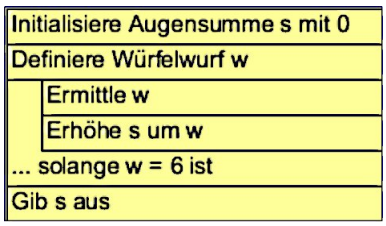

Abbildung 1.2: ein mini Struktogramm

```
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

Anwendung

foreach geeignet für beispielsweise ...

- Ausgabe von Elementen
- Suche nach Element
- Änderungen in Elementen

Nicht brauchbar für...

- Initialisierung
- Kopieren von Arrays
- Vergleich zweier Arrays

1.20.8 Varargs

- Mit Varargs (variable length argument lists) veränderliche Anzahl Argumente möglich
- Methodendefinition mit Vararg-Parameter
- Vararg-Parameter syntaktisch markiert mit Typ und drei Punkten:
  type... name
- Beispiel:
  int sum(int... args) { }
- Varargs ausschließlich in Parameterlisten erlaubt, nirgends sonst
- Vararg-Parameter im Methodenrumpf verwendbar wie ein Array
- Beispiel: int sum(int... args) ist im Rumpf der Methode gleichwertig mit int sum(int[] args)
- Aufrufer liefert beliebig viele Argumente für einen Vararg-Parameter
  sum(1, 2, 3)
- Jedes einzelne Argument muß kompatibel zum Vararg-Parameter sein
- Bei der Parameterübergabe:
  - Allokieren eines neuen Arrays mit Länge = Anzahl Argumente
  - Initialisieren des Arrays mit Argumentwerten
  - Zuweisen des Arrays an den Vararg-Parameter

1.21 Konstruktoren

- Definition eines Konstruktors wie normale Methode, außer ...
  - derselbe Name wie die Klasse
  - kein Vorsatz von void (oder anderem Ergebnistyp)
- Abgesehen davon: Kopf, Parameterliste, Rumpf wie andere Methoden
- Konstruktor mit leerer Parameterliste = Default-Konstruktor (engl. „def-ctor“)
- new ruft automatisch Konstruktor auf

Ziel:

Automatisch sinnvoller Startzustand für neu geschaffene Objekte

Idee:

Aufruf einer normalen Methode, überladen z.B. set für Rational-Objekte

Beispiel:

leere Parameter:

```
Rational r = new Rational();
r.set(0, 1);
```

nicht leere Parameter

```
class Rational
{
    Rational(int n, int d)
    {
        numer = n;
        denom = d;
    }
    ...
}
```

```
Rational r = new Rational(2, 3);    // Aufruf von Rational(int, int)
r.print();   // gibt 2/3 aus
```

- Technisch in Ordnung, aber unzuverlässig
- Besser: Konstruktoren (engl. „constructors“ = „ctors“):
- Spezielle Methoden, werden automatisch bei jedem new aufgerufen

1.21.1 Defaultwerten von Objektvariablen

- Lokale Variablen starten nicht initialisiert (ohne Wert)
- Gegensatz: Objektvariablen automatisch mit Defaultwerten initialisiert

Typ	Defaultwert
int	0
double	0.0
boolean	false
char	\u0000
Referenztypen	null

1.21.2 Copy-Konstruktoren

- Kopier-Konstruktor (engl. copy-ctor) erzeugt eine Kopie eines bereits existierenden Objekts
- Vorlage (= Original-Objekt) wird als Parameter (hier: that) übergeben
- Aufruf mit anderem Objekt als „Kopiervorlage“:

```
Rational original = new Rational(2, 3);
Rational copy= new Rational(original);
copy.print(); // gibt 2/3 aus
```

- Merkmal eines Kopier-Konstruktors: Parameter der gleichen Klasse
- Kopier-Konstruktor ausreichend für einfache Klassen, zu wenig für komplexere Klassen

1.21.3 Verkettete Konstruktoren

- Konstruktoren manchmal aufwändig (Tests und Vorverarbeitung von Parametern, Protokollausgaben, ...)
- Falls mehrere überladene Konstruktoren: Kopien des gleichen Codes in jedem Konstruktor
- Besser: Code nur in einem Konstruktor, von allen anderen mitbenutzen
- Konstruktor-Verkettung (engl. constructor chaining) = Aufruf eines anderen Konstruktors der gleichen Klassen
- Syntax: this als Repräsentant des „eigenen Objektes“
- Einschränkungen verketteter Konstruktoraufrufe:
  - this(...) muss erste Anweisung im Konstruktorrumpf sein
  - nur ein Aufruf von this(...) erlaubt

1.21.4 Ergebnisrückgabe bei Konstruktoren

- Definition von Konstruktoren ohne Ergebnistyp, auch nicht void
- Resultat liegt automatisch fest: neues Objekt
- return in Konstruktoren unzulässig
- Keine Möglichkeit zur Rückkehr mitten aus dem Rumpf

1.22 Klassenvariablen

1.22.1 Definition

- Bisher betrachtete Objektvariablen und Methoden beziehen sich auf ein bestimmtes Objekt (= Zielobjekt)
- Klassenvariable: einer ganzen Klasse zugeordnet, nicht einem einzelnen Objekt
- Definiert wie normale Objektvariable + Modifier static

1.22.2 Zugriff

- Zugriff mit Klassennamen, statt Zielobjekt Rational.count = 0;
- Klassenvariable existiert unabhängig von Objekten der Klasse
- Bereits früher benutzt (Wertebereichsgrenzen, mathematische Konstanten)
- Beispiel: Math.PI = Klassenvariable PI der Klasse Math

1.22.3 Initialisierung

- Initialisierung bei der Definition wie andere Objektvariablen
  . class Rational static int count = 0; ...
- Ohne explizite Initialisierung: Defaultwert abhängig vom Typ
- Beispiel: Rational.count hätte auch ohne explizite Initialisierung den Defaultwert 0
- Lebensdauer einer Klassenvariablen: gesamte Programmlaufzeit, unabhängig von Objekten

1.22.4 Verwendung

- Wichtigste Anwendung von Klassenvariablen: öffentliche Konstanten
- Beispiele: Math.PI, Integer.MAX_VALUE
- Werte öffentlicher Konstanten sollen sich nicht ändern:
  - Oft mit Modifier static und final

```
1 class Integer
2 {
3     static final int MAX_VALUE = 2147483647;
4     static final int MIN_VALUE = -2147483648;
5     ...
6 }
```

- Konvention zur Benennung von Konstanten: ganz in Großbuchstaben, Wortteile mit Unterstrichen (_) getrennt (anders als normale Variablen)
- Öffentliche Konstanten werden bei der Definition initialisiert (Alternative: statische Initializer)

1.23 Statische Methoden

- Statische Methode: richtet sich an ganze Klasse, kein bestimmtes Objekt (Äquivalent zur Klassenvariable)
- Definition mit Modifier static, sonstige Modifier und überladen wie gehabt
- Einsatz von statischen Methoden meist als Hilfsmethoden, dieunabhängig von bestimmten Objekten sind
- Beispiel: ggt-Methode der Klasse Rational

```
1 class Rational
2 {
3     static int gcd(int a, int b)
4     {...}
5 }
```

- Kann auch ohne Rational-Objekt benutzt werden
  System.out.println(Rational.gcd(221, 255));
- Einige vordefinierte Klassen (z.B. Math) definieren nur statische Methoden

1.23.1 main

- Von Anfang an benutzt: statische Methode main = Hauptprogramm
- Vor main existiert noch kein Objekt: main muss statisch sein
- Wird beim Start eines Javaprogramms von der JVM automatisch in der Klasse gesucht, die auf der Kommandozeile genannt ist
- Beispiel: Das Kommando $java classname sucht nach der Methode

```
class classname
{
    public static void main(String... args)
    {...}
}
```

- main ansonsten normale Methode: kann zum Beispiel
  - überladen werden
  - vom Programm selbst aufgerufen werden
  - in mehreren verschiedenen Klassen definiert sein
  - mit einem anderen Ergebnistyp als void definiert werden.

1.24 Packages

1.24.1 Motivation

- Pro übersetzter Bytecode-Datei (Extension .class): Bytecode einer Klasse
- Großes Programm: Viele Bytecode-Dateien
- Organisatorische Probleme:
  - Orientierung im Code
  - Bezüge zwischen Klassen
  - Namenskonflikte zwischen gleich benannten Klassen
  - Abgrenzung von Programmteilen
- Vergleichbar mit Dateien auf einer Festplatte

Idee:

- Package = Sammlung zusammengehörender Klassen
- Packages benannt mit Identifiern

Konvetion:

Englische Namen in kleinen Buchstaben und u.U. Ziffern, zum Beispiel utilities, project51, ...

Einschärnkungen:

- keine Großbuchstaben
- keine Interpunktionszeichen
- keine anderen Sonderzeichen

Klassennamen in unterschiedlichen Packages unabhängig, keine Kollisionen

1.24.2 Packagehierarchien

- Packages bilden Hierarchien (Baumstruktur)
- in einem Package untergeordnete Packages (Sub-Packages)
- Jedes Package in höchstens einem übergeordneten Package
- Vergleichbar mit Verzeichnissen in einem Dateisystem
- Packagepfad: List geschachtelter Packagenamen, getrennt mit Punkten
- Beispiele:
  - project51
  - project51.frontend
  - project51.frontend.web
  - project51.frontend.text
- Innerhalb eins Packages eindeutige Namen für alle "Bewohner":
  - Subpackages
  - Klassen
  - Enums (als spezielle Klassen)
  - Interfaces
- Packagehierarchie existiert nur äußerlich: Für Java sind alle Packages gleichrangig und gleichwertig

1.24.3 Dynamisches Laden von Bytecode

- JVM lädt Klassen bei Bedarf (engl. on the fly), nicht beim Programmstart
- Vorteil: Schneller Start, nicht benutzte Klassen werden überhaupt nicht geladen
- Bedingung: Bytecode effizient lokalisierbar
- Maßnahme: Packagepfad einer Klasse gibt Lage des Bytecodes im Filesystem vor Direkte Zuordnung Packagepfad ⇒ Directorypfad im Filesystem (nur eine Möglichkeit unter mehreren)
- Problem:
  - Packagenamen sind Java-Identifier
  - Directorynamen sind Namen des Betriebssystems
- Betriebssysteme schränken Directorynamen ein ⇒ defensive Benennungsregeln

1.24.4 import-Klausel

- Klassen sind mit qualifizierten Namen über Packagegrenzen hinweg ansprechbar
- Aber: Schreibaufwändig, mühsam, fehlerträchtig
- Einfacher mit import-Klausel:
  import packagepath.name;
- name im Rest des Quelltextes ohne Packagepfad nutzbar
- Import aller Klassen eines Packages mit Jokerzeichen (engl. wildcard) *
  import packagepath.*;
- Joker kann nur Klassen eines Packages ansprechen ⇒ nur einmal und nur als letztes Element in einer import-Klausel
- Unzulässig:
  - import *.datastore;
  - import project51.*.*;
- Joker schließt keine Subpackages ein
- Mehrere import-Klauseln zulässig, Reihenfolge ohne Bedeutung
- import-Klauseln stehen am Programmanfang, vor der Klassendefinition
- Bezeichnung Klausel (nicht „Anweisung“): werden nur vom Compiler ausgewertet, generieren keinen ausführbaren Code
- import-Klauseln regeln Zugriff auf andere Packages
- Gegenstück: package-Klausel definiert Packagezugehörigkeit einer Klassendefinition
- Syntax:
  package packagepath;
- Beispiel:

```
1 package project51.datastore;
2 class Rational {...}
```

- package-Klausel als Erstes im Quelltext (abgesehen von Kommentar), vor import-Klauseln und Definitionen
- package-Klausel und Pfad im Filesystem müssen übereinstimmen
- Aufruf von Compiler und JVM mit Packagepfad

1.24.5 Defaultpackage

- Quelltext ohne package-Klausel: Defaultpackage (auch: „anonymes Package“)
- Pfad im Filesystem = CLASSPATH ohne Zusatz
- Kein Import des Defaultpackages in andere Packages möglich (kein Name!)

1.24.6 Archivdateien - Jar-Files

- Bytecode in einer gepackten Archivdatei = alternative Organisationsform für Packages
- Archivformat Jar (java archive), Dateien = „Jar-Files“, Extension .jar
- Technisch: Zip-Files mit Erweiterungen
- Vorteile gegenüber Directories:
  - Leicht zusammen zu halten
  - Kompression für effiziente Übertragung
  - Metainformation bzgl. Inhalt
- Nachteile:
  - Kein direkter Zugriff auf Bestandteile
  - Kompression und Expansion kostet Rechenzeit
- Packagehierarchie in Jar-Files unverändert
- JVM findet Bytecode in Jar-Files
- Jar-Files im CLASSPATH: Gleichrangige Eintrittspunkte für Packagepfade
  CLASSPATH=/somewhere/project51.jar
- Dateien aller Arten in Jar-Files erlaubt, JVM wertet dann nur Bytecode aus
- Kommandozeilenwerkzeug jar zur Bearbeitung von Jar-Files
- Aufgaben:
  - jar -cf jarfile file file ...
    Packt die angegebenen Dateien in ein neues Jarfile (c = create).
    Jokerzeichen sind erlaubt, Directories werden rekursiv eingepackt.
  - jar -tf jarfile
    Liste des Inhalts (t = table of content)
  - jar -xf jarfile, jar -xf jarfile file file ...
    Packt alle bzw. die angegebenen Dateien aus (x = extract). Directories werden rekonstruiert.
- Aufruf mit zusätzlichen Schalter v (= verbose): Protokollausgaben auf dem Bildschirm
- Aufruf ohne Parameter: kurze Bedienungsanleitung
- Metainformation (= Verwaltungsdaten, Information über das Jarfile) in der Datei META-INF/MANIFEST.MF
- Textdatei mit Zeilen der Form
  name:value
- Wird automatisch angelegt mit Standard-Einträgen der Art
  Manifest-Version: 1.0
  Created-By: 1.5.0_04 (Sun Microsystems Inc.)
- Viele Einträge mit fester Bedeutung
- Zusätzliche Einträge erlaubt

1.25 Rekursion

1.25.1 Rekursion = Definition einer Funktion durch sich selbst

- Rekursive Definitionen erlauben oft intuitive und elegante Formulierungen von Problemen und Algorithmen
- Für uns nur sinnvoll: wohldefinierte rekursive Definitionen, rekursiver Abstieg bis zu einer Basisdefinition
- Verwandt mit Induktion

1.25.2 Beispiel: Sum(n) := Summe der Zahlen 1..n

$$Sum(n) = \begin{cases} 1, n = 1 \\ n + Sum(n-1) \quad \text{sonst.} \end{cases}$$

```
class RecursiveSum
{
    static int recursiveSum(intSum(n) = n + Sum(n-1)
    {
        int result = (n==1) ? 1 : n + recursiveSum(n-1);
        return result;
    }
    public static void main(String... args)
    {
        int n = Integer.parseInt(args[0]);
        int sum = recursiveSum(n);
        System.out.printf("Summe von 1...%d ist %d\n", n, sum);
    }
}
```

1.25.3 Beispiel: Fibonacci-Zahlen

$$F(n) := \begin{cases} 0, n = 0 \\ 1, n = 1 \\ Fn - 1 + F(n-2), \text{sonst.} \end{cases}$$

```
class Fibonacci
{
    static int fibonacci(int n)
    {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else return fibonacci(n-1) + fibonacci(n-2);
    }
    public static void main(String... args)
    {
    int n = Integer.parseInt(args[0]);
    int fib_n = fibonacci(n);
```

```
15      System.out.printf("Fibonacci(%d) = %d\n", n, fib_n);
16      }
17  }
```

1.25.4 Divide & Conquer

Bisher:

- teile Problem in mehrere kleinere Teile
- löse diese getrennt
- setze sie wieder zusammen

Idee: Wende dieses Verfahren rekursiv auf die Teilprobleme an, bis triviale Instanzen entstehen!
Extrem wichtiges Verfahren im Algorithmenentwurf, erlaubt oft elegante Formulierung und effiziente Lösung!

1.25.5 Effizienz:

- Rekursive Aufrufe können erhebliche Laufzeitkosten bewirken!
- Bei großen Problemen oft sinnvoll: rekursive Formulierung (besseres Verständnis) aber iterative Implementierung (oft höhere Effizienz)
- Bestimmte Klassen rekursiver Funktionen lassen sich automatisch in Schleifen (iterative Funktionen) überführen
- Wird bei Endrekursion oft vom Compiler gemacht

1.25.6 Arten/Stufen der Rekursion

- Rekursive Definitionen können in verschiedene Klassen eingeteilt werden
- Wichtige Klassifizierungen (es gibt noch mehr!):
  - Lineare Rekursion
  - Kaskadenrekursion
  - Endrekursion
  - Primitive Rekursion

Lineare Rekursion

Kommt in der rekursiven Definition nur höchstens ein rekursiver Aufruf vor spricht man von linearer Rekursion
⇒ es ergeben sich Ketten von Aufrufen

Kaskadenrekursion

Kommen mehrere rekursive Aufrufe in der Definition vor spricht man von Kaskadenrekursion
⇒ Baum von Aufrufen, oft exponentielle Laufzeit

Endrekursion (Tail Recursion)

Ist nur die letzte Anweisung einer linear rekursiven Funktion ein rekursiver Aufruf spricht man von Endrekursion
⇒ Kann unmittelbar durch while-Schleife ersetzt werden und umgekehrt, wird oft vom Compiler umgesetzt

Primitive Rekursion

Nimmt bei einer linear rekursiven Funktion der natürlichen Zahlen der Parameter mit jedem Aufruf um 1 ab oder zu spricht man von primitiver Rekursion
⇒ kann unmittelbar durch for-Schleife ersetzt werden

1.26 Unveränderliche Klassen

1.26.1 Motivation

- Bereits besprochenes Problem: Methode kann unerkannt fremdes Objekt manipulieren
- Ursache: Freier Zugriff auf Objektvariablen
- Lösungsidee: Unveränderliche Klassen (engl. „immutable classes“)
- lassen keine Änderungen an Objektvariablen zu
- Folge: Lesen von Informationen aus Objekten ok, Ändern nicht möglich

1.26.2 final-Objektvariablen

- Modifier final für Objektvariablen blockiert Änderungen (wie bei lokalen Variablen)
- final-Objektvariable muss einmal mit einem Wert versorgt werden
- Zuweisung des Wertes wahlweise ...
  - Im Konstruktor
  - Bei Initialisierung
- Nicht ausreichend:
  - Defaultwert
  - anderer Methodenaufruf
- Nachfolgende Änderung unzulässig ⇒ erster (und einziger) Wert bleibt für die gesamte Lebensdauer des Objektes konstant

1.26.3 Rückgabe neuer Objekte

- Problem: Änderungen von this nicht mehr möglich
- Ausweg: Ergebnis in neuem Objekt zurückliefern, statt this zu modifizieren

1.27 Wertesemantik

1.27.1 Definition

- Wertesemantik (engl. „value semantics“) einer Klasse: Methoden lassen this und Parameter unberührt
- Verhalten entspricht dem von primitiven Typen: Operationen lassen Operanden unberührt
- Gegensatz: Referenzsemantik (engl. „reference semantics“):
  Operationen können this oder Parameter-Objekte oder beide verändern
- Zusammenfassung:

Primitive Typen	Wertesemantik
Unveränderliche Klassen	Wertesemantik
Veränderliche Klassen	Referenzsemantik

1.27.2 Warum Wertesemantik?

- Wertesemantik ist oft vorteilhaft
- Beispiel: Suche nach der Ursache für fehlerhaftes Objekt
- Referenzsemantik: Jeder Methodenaufruf kommt in Betracht, muss überprüft werden
- Beispiel: a.mult(b);
  könnte a oder b oder keines oder beide verändern
- Wertesemantik: Nur Konstruktoraufrufe zu prüfen
- Methodenaufrufe können ein korrekt erzeugtes Objekt nicht mehr stören
- Definieren Sie Klassen wenn möglich mit Wertesemantik (= unveränderlich)

1.28 Kopieren und vergleichen

1.28.1 Flache Kopie

- Problematisch: Klasse mit Objekten als Objektvariablen
- Simpler Kopier-Konstruktor mit Wertzuweisungen: dupliziert Referenzen, nicht Objekte (Aliasing)
- Der einfache Kopier-Konstruktor erzeugt eine flache Kopie (engl. „shallow copy“): Original und Kopie referenzieren dieselben Objektvariablen

```
1 class RatRange
2 {
3     RatRange(RatRange that)
4     {
5         lower = that.lower;
6         upper = that.upper;
7     }
8     ...
9 }
```

1.28.2 Tiefe Kopie

- Objektvariablen mit Kopier-Konstruktor statt Wertzuweisung kopieren
- Äußerlich kein Unterschied
- Erzeugt eine tiefe Kopie (engl. „deep copy“): Original und Kopie enthalten logisch gleiche, isolierte Objektvariablen:

```
1 class RatRange
2 {
3     RatRange(RatRange that)
4     {
5         lower = new Rational(that.lower);
6         upper = new Rational(that.upper);
7     }
8     ...
9 }
```

1.28.3 Kopieren von Objektvariablen

- Regeln zum Kopieren von Objektvariablen im Kopier-Konstruktor abhängig vom Typ:

primitiver Typ	Wertzuweisung (Beispiel Rational)
Referenztyp	Kopier-Konstruktor
Unveränderliche Klassen	Wertzuweisung (wie primitive Objektvariablen)

- Aber: Kopier-Konstruktor zu schwach bei Vererbung (siehe später)
- Dort: mächtigerer Mechanismus nötig

1.28.4 Vergleich von Objekten

- Methode equals zum Vergleich von Objekten
- equals erwartet anderes Objekt x als Parameter, liefert ...
  - true wenn dieses Objekt und x logisch gleichwertig sind
  - false wenn dieses Objekt und x logisch verschieden sind
- Was heißt „logisch gleich“?
  - Benutzer kann Objekte nicht unterscheiden
  - ... so lange keines von beiden verändert wird
- Gleichheit ist schwächer als Identität:
  - zwei identische Objekte sind auch gleich,
  - zwei gleiche Objekte müssen nicht identisch sein

1.29 Datenkapselung

1.29.1 Idee:

- Modul (engl. module) = Programmteil, in Java i.d.R. eine Klasse
- Modularisierung = Aufteilung in Module
- Modularisierung ist Ziel bei der Konstruktion von (größeren) Programmen
- Reduktion der Abhängigkeiten zwischen Modulen: Module leichter einzeln
  - entwerfen
  - implementieren
  - austauschen
  - erweitern
  - testen
  - korrigieren
  - ...
- Datenkapselung (engl. „data hiding“, „data encapsulation“): Maßnahme zur Reduktion der Abhängigkeiten

1.29.2 Daten und Operationen

- Datenkapselung = Verbergen von Daten hinter Operationen
- Zugriff auf Daten nur über Operationen
- Java (und andere objektorientierte Sprachen):

Daten	↔ Objektvariablen
Operationen	↔ Methoden

- Anwender einer Klasse darf Methoden aufrufen, aber keine Objektvariablen ansprechen

1.29.3 Zugriffsschutz

- Modifier private begrenzt Sichtbarkeit einer Definition auf die eigene Klasse
- Von außen kein Zugriff
- Innerhalb der eigenen Klasse keine Einschränkung
- private gilt für den Compiler beim Übersetzen, nicht für die JVM zur Laufzeit
- private schottet Klassen gegeneinander ab, nicht Objekte!
  ⇒ Unveränderliche Klassen zusammen mit Datenkapselung einsetzen

1.29.4 private-Methoden

- private für alle Definitionen möglich, auch für Methoden
- private-Methoden von außen nicht aufrufbar, nur von Methoden der eigenen Klasse
- Sinnvoll für Hilfsmethoden, die der Anwender nicht benutzen soll

1.29.5 Getter

- private-Objektvariable von außen nicht erreichbar
- Wert dennoch zugänglich machen: Auskunftsmethode (auch „Inspector-Methode“, „Accessor-Methode“, engl. „getter“) anbieten
- Definition eines Getter zu einer Objektvariablen:
  private type name;
- nach dem Muster:

```
type getName()
{
    return name;
}
```

1.29.6 Setter

- private-Objektvariablen in veränderlichen Klassen: von außen kein Zugriff, keine Veränderung möglich
- Um dennoch Modifikation zulassen: Änderungsmethoden (auch „Modifier-Methoden“, engl. „setter“) anbieten
- Definition eines Setter zu einer Objektvariablen
  private type name;
- nach dem Muster (hier this sinnvoll anwendbar):

```
void setName(type name)
{
    this.name = name;
}
```

1.29.7 Vorteile Setter/Getter

Private Objektvariablen, Setter und Getter auf den ersten Blick unnötig kompliziert gegenüber ungeschützten, öffentlichen Objektvariablen. Einsatz von Settern und Gettern dennoch sehr sinnvoll:

- Kontrolle der Werte
  - Oft nur eine Teilmenge aller Werte des Typs einer Objektvariablen zulässig
  - Beispiel Rational: denom darf nicht null sein
  - Setter können unerwünschte Werte abweisen; bei freiem Zugriff keine Kontrolle
- Fehlersuche
  - Setter können alle Zugriffe auf Objektvariablen protokollieren; bei freiem Zugriff keine Überwachung möglich
- Abweichende Implementierung
  - Tatsächliche Objektvariable hinter Setter und Getter nicht sichtbar, muss überhaupt nicht existieren

  - Entsprechende Werte können pro Aufruf neu berechnet werden; für den Anwender nicht unterscheidbar

- Verborgene Optimierungen
  - Mit Gettern können Maßnahmen bis zum tatsächlichen Zugriff verzögert werden
  - Beispiel Rational: Mit ungekürztem Bruch rechnen, erst beim Aufruf eines Getters wirklich kürzen

1.30 Vererbung

1.30.1 Interfaces

- Schnittstelle (engl. interface) einer Klasse:
  - Signaturen der öffentlichen (= nicht-privaten) Methoden
  - + öffentliche (= nicht-private) Objektvariablen
- Implementierung: alles andere
- Intuitiv:
  - Das Interface beschreibt, was eine Klasse bietet
  - Die Implementierung legt fest, wie sie das bewerkstelligt
- Anwender muss nur das Interface einer Klasse kennen, um sie zu verwenden
- Beziehung zwischen Klassendefinition und Anwendung ähnelt Geschäftsbeziehung:

Java	**Geschäftsleben**
Klassendefinition	Anbieter, Lieferant
Anwendung einer Klasse	Kunde
Interface	Vertrag, vereinbarte Leistungen
Implementierung	Betriebsmittel, interne Maßnahmen, Geschäftsgeheimnisse

Idee:

- Interface = isolierte Schnittstelle, ohne Implementierung (siehe Datenkapselung)
- Ähnlich wie Klassendefinition, aber mit reserviertem Wort interface statt class
- Im Interface fehlen Rümpfe der public-Methoden (statt dessen nur „;“) andere als public-Methoden, Konstruktoren, Objektvariablen (außer öffentlichen Konstanten)

```
1  interface Complex
2  {
3      public double getReal();
4      public double getImag();
5      public double getDistance();
6      public double getPhase();
7      public Complex add(Complex that);
8      public Complex mult(Complex that);
9      ...
10 }
```

Interface vs. Klasse

- Interface ≈ Vertrag, aber Interface ≠ Klasse
- Legt nur Anforderungen fest, liefert keine Implementierung
- Keine Objekte von Interfaces möglich!
- Wie bei Klassen: Eine Interfacedefinition pro Quelltextdatei,
  Dateiname = Interfacename + .class
- Beispiel: Interface Complex in der Quelltextdatei Complex.java
- Bytecode: ohne ausführbare Anweisungen
- Elemente eines Interfaces implizit public, alles andere unzulässig

Implementieren von Interfaces: Klasse zum Interface

- Isoliertes Interface nutzlos
- Konkrete Klassen implementieren das Interface = definieren die Methoden des Interface
- implements koppelt Klasse mit Interface.
- Schematisch:
  class Name implements Interface

  . ...

- Klassendefinition ansonsten normal
- Implementierung von Interfaces in einer Klasse
  class Polar implements Complex ...
- Interfaces kennen keine konkreten Objekte, keine Objektvariablen ⇒ können keine Konstruktoren besitzen

Interfaces als Typen

- Interfaces sind Typen, ebenso wie Klassen
- Zulässig für Variablendefinitionen, Parameterlisten, Methodenergebnis, Arrayelemente, ...
- Beispiel: Variable c vom Typ Complex
  Complex c;
- Objekte des Interface gibt es nicht. Was könnte c überhaupt zugewiesen werden?
- Alle implementierenden Klassen kompatibel zum Interface
- Beispiel: An c kann ein Cartesian-Objekt zugewiesen werden:
  Complex c = new Cartesian(1, 2);

Methodenaufrufe

- Methodenaufruf mit Variable von Interfacetyp uneindeutig
- Im allgemeinen Fall: Der Compiler kann keine getReal-Methode auswählen!

Dynamisches Binden

- Auswahl einer konkreten Methode fällt erst zur Laufzeit, der Compiler trifft keine Entscheidung
- JVM sucht pro Aufruf eine Methode des momentan zugewiesenen Objekts
- Bezeichnung: Dynamisches Binden = Zuordnung Aufruf ↔ Rumpf zur Laufzeit (weitere Form des Polymorphismus)

Statischer vs. Dynamischer Typ

- Dynamischer Typ = Typ des tatsächlich an eine Variable zugewiesenen Objekts
  Complex c = new Cartesian(1, 2);
- c hat den ...

statischen Typ	Complex (gemäß Variablendefinition)
dynamischen Typ	Cartesian (das tatsächlich zugewiesene Objekt)

- Der ...
  - statische Typ bleibt immer gleich
  - dynamische Typ kann sich ändern
- Beim Übersetzen ist der statische Typ entscheidend (Compiler)
- Zur Laufzeit ist der dynamische Typ entscheidend (JVM)

Einsatz von Interfaces

- Entwicklung, Modifikation von Interface-Implementierungen ohne Rücksicht auf andere Klassen zum gleichen Interface
- Beispiel: neue, dritte Implementierung komplexe Zahlen namens Perplex:
- Existenz von Cartesian und Polar kann vollkommen ignoriert werden
  - Anwenderprogramme können sofort mit Perplex arbeiten
  - Reibungsloses Zusammenspiel automatisch sichergestellt