Programmierung mit Java

Hans Joachim Pflug
IT Center, RWTH Aachen
pflug@itc.rwth-aachen.de

24.6.2024

Inhaltsverzeichnis

Kapitel 1

Lineare Programme

1.1 Hello world

Das traditionell erste Programm, das man in einer neuen Sprache schreibt, heißt „Hello World“. Es macht nichts weiter, als die Textzeile „Hello World“ auszugeben. Die Schwierigkeit besteht darin, das ganze „Drumherum“ so zum Laufen zu kriegen, dass überhaupt ein Programm ausgeführt werden kann. Im ersten Schritt werden wir versuchen, das Hello-World-Programm zu starten, im zweiten Schritt werden wir das Programm etwas näher unter die Lupe nehmen.

1.1.1 Start eines Java-Programms

Ein Java-Programm kann ganz ohne Entwicklungsumgebung mit einem Texteditor erstellt und dann aus einer Eingabeaufforderung (oder einer Linux-Konsole) gestartet werden. Es ist lehrreich, dies in einem ersten Versuch auch zu tun. Später werden wir allerdings die Entwicklungsumgebung *Eclipse* verwenden. Zunächst starten wir einen Texteditor und tippen folgende Zeilen ein:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

Dann speichern wir den Text unter dem Namen `HelloWorld.java`. Es muss genau dieser Name gewählt werden, da er mit dem Namen aus der ersten Zeile des Programms übereinstimmen muss. Anschließend kann das Programm *compiliert* werden. Dazu gibt man in einer Eingabeaufforderung die Zeile

```
javac HelloWorld.java
```

ein. Unter Windows kann möglicherweise das Programm *javac* nicht automatisch gefunden werden. Es befindet sich oft unter einem Pfad, wie

```
C:\Programme\Java\jdk1.5.0\bin\javac.exe
```

javac erzeugt aus der java-Datei (auch *Source-Code* oder *Quellcode* genannt), eine class-Datei, die hier den Namen *HelloWorld.class* hat. Die class-Datei besteht aus Java-Bytecode. Die class-Datei kann nicht direkt ausgeführt werden, sondern benötigt dazu ein weiteres Programm namens *java*[1]. Das entsprechende Kommando heißt:

```
java HelloWorld
```

Das Programm *java.exe* befindet sich im gleichen Verzeichnis, wie *javac.exe*.

1.1.2 Der Programmrahmen

Sehen wir uns noch einmal den Code an. Zur besseren Übersicht erhält jede Zeile eine Zeilennummer:

```
1  public class HelloWorld {
2    public static void main(String[] args) {
3      System.out.println("Hello World");
4    }
5  }
```

Die Zeilen 1, 2, 4 und 5 bilden einen *Rahmen*, der in jedem Java-Programm vorhanden sein muss. Die Form dieses Rahmens ist:

```
public class HelloWorld {
  public static void main(String[] args) {

  }
}
```

Diese Form müssen wir zunächst einmal hinnehmen, ohne nach dem tieferen Sinn zu fragen. Merken müssen wir uns schon jetzt:

- In der ersten Zeile muss immer der Dateiname (ohne die Endung *.java*) eingesetzt werden (in diesem Beispiel also *HelloWorld*). Diesen Namen nennt man auch *Klassenname*, und den Inhalt der Datei nennt man *Klasse* ist. Bis zur Einführung der Objektorientierung reicht es, wenn wir uns der groben Zusammenhang *Datei=Klasse* merken.

- Das Programm startet in der ersten Zeile, die auf

```
  public static void main(String[] args) {
```

folgt und beendet sich, wenn es auf die zugehörige geschweifte Klammer zu stößt

Alle Zeilen dazwischen werden der Reihe nach ausgeführt. Das Programm

[1] Das Programm *java* wird *Laufzeitumgebung* genannt.

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hallo");
    System.out.println("und tschuess");
  }
}
```

gibt den Text

```
Hallo
und tschuess
```

aus.

1.1.3 Formatierung

Java selbst ist die Formatierung des Codes recht gleichgültig. Die Zeilen

```
public class HelloWorld{public static void main(String[] args) {
System.out.println("Hello World");}}
```

funktionieren genauso gut, wie das letzte Beispiel. Um den Quellcode auch für Menschen lesbar zu machen, gibt es zusätzliche Regeln, die *Java Code Conventions*. Die vollständigen Code Conventions sind im Netz unter

```
http://java.sun.com/docs/codeconv/index.html
```

zu finden. Die ersten wichtigen Punkte sind:

- Nur eine Anweisung pro Zeile
- 4 Spalten Einrückung pro sich öffnender geschweifter Klammer, sowie 4 Spalten zurück nach jeder sich schließenden geschweiften Klammer.[2]

1.1.4 Die Anweisung

Die einzige Anweisung ist

```
System.out.println("Hello World");
```

Dazu ist zu sagen:

- Jede Anweisung muss mit einem Semikolon abgeschlossen werden.
- Die Anweisung `System.out.println` gibt alles, was zwischen den nachfolgenden Klammern steht, auf dem Bildschirm aus. Die Anführungszeichen sagen, dass es sich hierbei um ein Stück Text handelt, der so, wie er ist (ohne Anführungszeichen), auszugeben ist.

[2] In diesem Skript wird hiervon leicht abgewichen und nur 2 Spalten eingerückt. 4 Spalten haben den Nachteil, dass der Text sehr schnell nach rechts rückt. Dagegen sehe ich bei 2 Spalten noch keinen Verlust an Übersichtlichkeit.

1.2 Variablen

Eine wichtige Eigenschaft einer Programmiersprache ist die Fähigkeit, **Variablen** zu speichern. Eine Variable kann man sich als einen Platz im Hauptspeicher vorstellen, der einen Namen erhalten hat, gewissermaßen eine Box mit einem Namen und einem Inhalt.

fred	Hello.
hour	11
minute	59

1.2.1 Datentypen

Außer dem Wert und dem Namen hat jede Variable einen **Typ**, der angibt, ob in der Variable eine Zeichenkette, eine ganze Zahl, eine reelle Zahl oder etwas anderes gespeichert werden kann. Java kennt zwei verschieden Arten von Typen

- den *primitiven Datentyp* und
- die *Klasse*.

In diesem Kapitel beschäftigen wir uns, mit einer Ausnahme, nur mit primitiven Datentypen. Die einzige Ausnahme ist der Datentyp *String*, der zwar eine Klasse ist, aber aufgrund einiger Besonderheiten wie ein primitiver Datentyp behandelt werden kann. Java kennt 8 primitive Datentypen:

Typ	Werte	Bit
boolean	*true* oder *false*.	1
char	Ein Zeichen.	16
byte	Ganze Zahl zwischen -128 und 127.	8
short	Ganze Zahl zwischen -32768 und 32767.	16
int[3]	Ganze Zahl zwischen ca. -2 Milliarden und 2 Milliarden.	32
long	Ganze Zahl zwischen ca. -10^{20} und 10^{20} (100 Trilliarden).	64
float	Fließkommazahl mit ca. 8 Stellen zwischen $\pm 10^{38}$ und $\pm 10^{-45}$.	32
double	Fließkommazahl mit ca. 16 Stellen zwischen $\pm 10^{308}$ und $\pm 10^{-324}$.	64

Ein **String** nimmt eine ganze Zeichenkette (bis max. 2 Milliarden Zeichen) auf. Auch wenn wir zunächst Strings wie primitive Datentypen behandeln, müssen wir einen Unterschied immer beachten:

- Primitive Datentypen werden in Java *klein* geschrieben, also: *char, int, float, ...*
- Klassen werden in Java *groß* geschrieben, also: *String*.

[3] **int** ist der Standard-Ganzzahltyp. Rechenoperationen gehen mit int schneller als mit byte, short oder long. byte und short werden nur bei Speicherplatzmangel benutzt, long nur bei sehr großen Zahlen.

Somit hat jede Variable einen *Namen*, einen *Typ* und einen *Wert*. 3 Beispiele:

Name	Typ	Wert
fred	String (Zeichenkette)	Hello.
hour	int (Ganzzahl)	11
minute	int (Ganzzahl)	59

1.2.2 Deklaration von Variablen

Variablen müssen *deklariert* werden. Dabei erhält eine Variable einen bestimmten Namen und einen bestimmten Typ. Weiterhin bekommt die Variable ein Stück des Hauptspeichers als Platz für den Variablenwert. Die Variable erhält aber noch keinen bestimmten Wert. Die Deklaration sieht im Code wie folgt aus:

```
String fred;
int hour;
int minute;
```

Es ist möglich der Variable bei der Deklaration gleich einen Anfangswert zuzuweisen. Dies zeigt das folgende Code-Stück:

```
String fred = "Hallo";
int hour = 10;
int minute = 30;
```

Wird der Variablen bei der Deklaration kein Wert zugewiesen, muss das zu einem späteren Zeitpunkt geschehen. Es ist nicht möglich, mit einer Variablen zu rechnen, die noch keinen Anfangswert hat. Versucht man das, meldet der Compiler einen Fehler. Eine Deklaration darf **überall** im Programm vorkommen, auch nach ausführbaren Anweisungen.

```
System.out.println("Hallo");
int a;
a=5;
```

Ab Java 10 kann man die Typangabe durch das Schlüsselwort `var` ersetzen, wenn die Variable bei der Deklaration gleich einen Anfangswert enthält. Der Typ der Variablen wird dann aus dem Typ des Anfangswerts ermittelt. Allgemein senken `var`-Anweisungen aber die Lesbarkeit des Codes. Die Programmzeilen

```
var a = 5;
var b = 5.;
```

legen `a` als `int`-Variable und `b` als `double`-Variable an, da `5` ein `int`-Literal und `5.` ein `double`-Literal ist. Diese Information ist aber nur schwer erkennbar. In bestimmten Situationen ist der Einsatz von `var`-Deklarationen trotzdem sinnvoll, wie bei Generics und anonymen inneren Klassen (siehe spätere Kapitel).

1.2.3 Namen von Variablen

Hier gibt es sowohl zwingende Regeln als auch weitergehende Einschränkungen durch die *Java Code Conventions*. Zusammengefasst gilt:

- Variablennamen beginnen mit einem Kleinbuchstaben.
- Die restlichen Zeichen dürfen aus Klein- und Großbuchstaben, Ziffern und dem Unterstrich (_) bestehen. Groß- und Kleinschreibung wird beachtet: name, nAme und nAME sind unterschiedliche Namen.
- Besteht ein Variablenname aus mehreren Worten, werden diese direkt zusammengesetzt. Jedes Teilwort beginnt mit einem Großbuchstaben. Beispiel: Eine Variable, die Worte zählt, könnte `wortZaehler` heißen.[4]
- Allgemein soll natürlich aus der Bezeichnung der Variablen möglichst klar ihre Bedeutung hervorgehen. Sogenannte Einweg- (*throwaway*)-Variablen, die nur in einem sehr kurzen Teilstück verwendet werden, dürfen auch aus einem einzelnen Buchstaben bestehen. Gängig sind i, j, k, m und n für Integer-Einweg-Variablen.

1.2.4 Rechnen mit Variablen

Einfache Ausgabe von Variablen

Bevor wir mit Variablen rechnen, zunächst eine einfache Methode, den Wert einer Variablen auf dem Bildschirm auszugeben. Die Anweisung

```
System.out.println(hour)
```

Gibt den Wert der Variablen *hour* auf dem Bildschirm aus. Beachten Sie, dass im Vergleich zu den vorigen Beispielen die Anführungszeichen in der Klammer fehlen. Dies bedeutet, dass nicht der Text „hour“, sondern der Wert der Variablen *hour* ausgegeben werden soll.

Rechnen mit Variablen

Nun weisen wir den Variablen einen Wert zu. Dazu benötigen wir den sogenannten *Wertzuweisungsoperator* „=“. Beispiel:

```
int minute;
minute = 5;
```

Das „=“ in der zweiten Zeile hat die Bedeutung: *Die Variablen auf der linken Seite erhält das Ergebnis des Ausdrucks auf der rechten Seite.* Es handelt sich dabei ausdrücklich nicht (wie wir gleich sehen werden) um eine mathematische Gleichung. Deshalb ist die Zeile

```
minute = minute + 5;
```

[4] In C würde man statt dessen eher `wort_zaehler` oder nach ungarischer Notation `iWort_zaehler` schreiben.

auch kein mathematischer Unsinn, sondern bedeutet: Nimm den Wert der Variablen *minute*, addiere 5 und weise das Ergebnis wieder der Variable *minute* zu.

Auf der rechten Seite der Zuweisung darf ein Ausdruck stehen. Es sind dabei eine ganze Anzahl von Operatoren möglich, von denen hier außer den 4 Grundrechenarten+, −, ∗, / nur den **Modulo-Operator** % erwähnt sein soll. a % b ergibt den Rest der Ganzzahldivision von a durch b.[5] Mit diesem Wissen können wir schon ein kleines Programm schreiben, das den Kehrwert einer Zahl berechnet:

```
public class Kehrwert {
  public static void main(String args[]) {
    double a;
    a = 7;
    a = 1/a;
    System.out.println(a);
  }
}
```

Diese Programm erzeugt als Ausgabe:

```
0.14285714285714285
```

Auch Operationen innerhalb eines Funktionsaufrufs sind möglich, wie das folgende Beispiel zeigt:

```
System.out.println(2+5);    --> Gibt die Zahl 7 auf dem Bildschirm aus
```

Kurzschreibweisen

Folgende Kurzschreibweisen sind möglich:

```
Lang                        Kurz

a = a + x;                  a += x;
a = a - x;                  a -= x;
a = a * x;                  a *= x;
a = a / x;                  a /= x;
a = a + 1;                  a++;  (nur bei Ganzzahlen)
a = a - 1;                  a--;  (nur bei Ganzzahlen)
```

1.2.5 Besonderheiten bei Rechenoperationen

Rechenoperationen in Java sind zwar prinzipiell einfach, aber oft stolpert man über einige Besonderheiten, die ein unerwünschtes Ergebnis erzeugen.

[5] Vorsicht: Modulo verhält sich anders als erwartet, wenn a oder b kleiner sind als 0.

Literale

Im Ausdruck

```
a = 3 + 4 * 2;
```

stehen drei Zahlen direkt im Code. Solche direkt im Code stehende Werte heißen*Literale*. Literale haben nicht nur einen Wert, sondern auch einen Typ, der sich aus der genauen Schreibweise ergibt. Es gibt int-, long-, double-, float-, boolean-, char- und String-Literale. Die Kennzeichen sind:

Typ	Literal	Beispiele
int	Reine Zahl	12; -74
long	Zahl mit angehängtem „l“	12l; -74l
double	Zahl mit Dezimalpunkt	12.3; -74.
double	Zahl mit angehängtem d	4d; -12d
double	Zahl in Exponentialschreibweise	1e5 (für 10^5), 4e-3 (für $4 \cdot 10^{-3}$)
float	Zahl mit angehängtem f	1f; 1.3e4f
boolean	„true“ oder „false“	true; false
char	Einzelner Buchstabe in Hochkommas	’a’; ’3’
String	Zeichenkette in doppelten Hochkommas	”f”; ”hallo”; ”123”

Eine Variable kann stets nur Werte des eigenen Typs aufnehmen. *int i = 12* ist ok, *int i= ”123”* nicht.

Typkonversionen

Folgende Zeilen sind möglich:

```
double d;
int x = 10;
d = x;

double e = 3;
```

Folgende Zeilen verursachen einen Compiler-Fehler:

```
int i;
double d = 10;
i = d;

int j = 2.5;
```

Was ist der Unterschied? Java konvertiert den Typ automatisch, wenn das „verlustlos“ möglich ist; also in der Richtung:

```
byte -> short -> int -> long -> float -> double
```

In der anderen Richtung könnte der Wert nicht in die neue Variable „passen“. Hier muss man dem Compiler explizit sagen, dass man eine Typkonversion haben will.

```
int i;
double d = 10.5;
i = (int) d;     //Typkonversion auf int. i erhaelt den Wert 10

int j = (int) 2.5;
```

Konvertieren von einer Fließkomma- in eine Festkommazahl rundet immer ab, selbst wenn der Nachkommaanteil 0.999999999 ist.

Typ von Ergebnissen

Wenn die Operanden einer Rechenoperation unterschiedlich sind, wird einer der beiden Operanden vor der Rechenoperation im Typ gewandelt. Der Operand, dessen Typ in der Hierarchie

```
byte -> short -> int -> long -> float -> double
```

weiter links steht, wird in den Typ des anderen Operanden gewandelt. Letztere ist auch der Typ des Ergebnisses. Beispiel: Die Operation $3 + 6.$ hat (wegen der verwendeten Literale) die Operanden `int` und `double`. Vor der Summation wird der Integer-Operand automatisch in double konvertiert. Das Ergebnis ist dann ebenfalls vom Typ double. Wichtig wird diese Vorgehensweise beim nächsten Abschnitt.

Ganzzahl- und Fließkommadivision

Bei einer Division zweier ganzer Zahlen kann eine gebrochene Zahl als Ergebnis herauskommen. Java hält sich aber streng an die im letzten Abschnitt formulierten Regeln und erzeugt als Ergebnis der Division zweier Ganzzahlen wieder eine Ganzzahl. Dabei wird der Nachkommateil des Ergebnisses abgeschnitten. Diese Art der Division nennt man auch Ganzzahldivision. Oft ist dieser Effekt erwünscht, manchmal aber auch unbeabsichtigt. Beispiele:

```
double d1 = 1/2;    //Ganzzahldivision, ergibt 0.0
double d2 = 1/2.;   //Zweiter Operand ist double, deshalb Wandlung
                    //der 1 in double, anschliessend Fliesskommadivision
```

Im ersten Beispiel werden zunächst die beiden Integer-Werte 1 und 2 per Ganzzahldivision geteilt. Das Ergebnis ist der Integer-Wert 0. Dieser Wert wird automatisch in den double-Wert 0 gewandelt und der Variable `d1` zugewiesen. Im zweiten Beispiel hat die Divsion zwei Operanden unterschiedlichen Typs. Da double in der Hierarchie über int steht, wird zunächst die 1 in einen double-Wert gewandelt. Dann wird mit Fließkommadivision geteilt und das double-Ergebnis 0.5 der Variablen `d2` zugewiesen.

Auswertungsreihenfolge

Zwischen den einzelnen Operatoren gibt es eine Hierarchie, die festlegt, welche Operationen zuerst ausgeführt werden. Für die uns bekannten Operatoren heißt

sie schlicht„Punkt- vor Strichrechnung“, wobei der %-Operator zur Punktrechnung zählt. Um die Auswertungsreihenfolge zu ändern, dürfen im Ausdruck Klammern gesetzt werden:

```
int a, b;
a = 3 + 4 * 2;      --> ergibt 11
b = (3 + 4) * 2;    --> ergibt 14
```

Bei mehreren Operatoren werden die Operationen von links nach rechts ausgeführt. Besonders bei der Division ist das wichtig:

```
a = 12 / 2 / 2      --> ergibt 3 = (12/2)/2
a = 12 / 10 / 2     --> ergibt 0 (Ganzzahl-Divisionen)
a = 12 / 10 / 2.    --> ergibt 0.5 (nur erste Divsion Ganzzahl)
a = 12 / 10. / 2    --> ergibt 0.6 (beide Divisionen Fliesskomma)
```

Das Setzen von Klammern ist auch in vielen Fällen sinnvoll, in denen es gar nicht nötig wäre:

- Wenn man die Regeln im einzelnen nicht genau im Kopf hat und nicht nachsehen will.
- Selbst wenn man die Regeln im Kopf hat, ist das bei anderen Leuten, die sich den Code nachher ansehen müssen, noch lange nicht der Fall. Daher sollte man bei nichttrivialen Fällen Klammern zur Verdeutlichung setzen.

1.2.6 Operationen für Character und Strings

Konvertierung zwischen Integern und Charactern

Character-Variablen können in Integer-Variablen (oder andere Ganzzahl-Typen) konvertiert werden und umgedreht. Beispiel:

```
char c = 'A';
int z = c;

int y = 48;
char b = (char) y;
```

Welche Werte enthalten jetzt `z` und `b`? Dies richtet sich nach dem *ASCII-Wert* der Zeichen. Diesen Wert kann man der Tabelle in Anhang C entnehmen. In diesem Beispiel ist also `z=65` und `b='0'`. Ist ein Character eine Ziffer, kann man mit

```
char c = '3';
int z = c-48;    //Ergibt 3
int y = c-'0';   //Ergibt ebenfalls 3
```

diese Ziffer als Integer-Wert erhalten.

Verkettungsoperator für Strings

Für Strings gibt es einen einzigen möglichen Operator: +. Er hängt 2 Strings aneinander. Beispiel:

```
String s = "Hallo "+"Welt";  --> ergibt "Hallo Welt"
```

Konvertieren in Strings

Ist ein Argument eines „+“-Ausdrucks ein String, so werden alle anderen Summanden automatisch in Strings verwandelt. Zum Beispiel:

```
int i = 10;
String s = "i = "+i;    --> ergibt i = 10

System.out.println("Wert fuer i: "+i);
```

Trick: Will man eine String-Darstellung eines primitiven Datentyps, kann man das nach folgendem Muster erhalten:

```
int i = 10;
String s = ""+i;    //s hat jetzt den Wert "10"
```

Leider ist es nicht so einfach, Strings in primitive Datentypen zu wandeln. Wir werden einen Weg im nächsten Kapitel kennenlernen.
Vorsicht Falle: Die Reihenfolge der Auswertung geht auch hier von links nach rechts vor sich. Dies kann auch für erfahrene Programmierer zur Falle werden:

```
System.out.println("a"+1+2);     --> a12
System.out.println(1+2+"a");     --> 3a
System.out.println("a"+1*2);     --> a2 (Punkt- vor Strichrechnung)
System.out.println("a"+1-2);     --> Fehler: Substraktion von Strings
                                     nicht erlaubt.
```

Weitere String-Operationen

Es gibt in Java eine große Bibliothek von String-Operationen. Diese Bibliothek ist objektorientiert aufgebaut und wird darum erst in Kapitel 3 im Zusammenhang mit der Objektorientierung betrachtet. Im Vorgriff darauf gibt es hier schon einen kleinen Einblick, der zahlreiche interessante Übungsaufgaben ermöglicht.
„String-Operationen“ heißen in objektorientierten Sprachen *String-Methoden*. Wir betrachten 2 String-Methoden:

- Die Methode `length()` gibt die Anzahl der Zeichen eines Strings zurück.
- Die Methode `charAt(int pos)` gibt das Zeichen an der Position `pos` als `char` zurück. Das erste Zeichen ist dabei an Position 0.

Um die Methode aufzurufen, wird der Methodenname mit einem Punkt an eine Variable oder ein Literal angehängt. Hier einige Beispiele:

```
int a1 = "Hallo".length();        // -> 5
char c1 = "Hallo".charAt(0);      // -> 'H'
String s = "Test";
char c2 = s.charAt(1);            // -> 'e'
char c3 = s.charAt(s.length()-1); // -> 't' (letztes Zeichen)
```

1.3 Kommentare

Ein Kommentar ist ein Stück deutscher (oder englischer) Text, der gewöhnlich erklärt, was ein Programm an dieser Stelle tut. Java selbst ignoriert die Kommentare beim Compilieren.

- **Zeilenkommentare:** Das Kommentarzeichen für Zeilenkommentare besteht aus zwei Schrägstrichen „//“. Der Text ab dem Kommentarzeichen bis zum Zeilenende zählt als Kommentar.

```
//Zeilenkommentar
System.out.println("Hallo");   //Weiterer Zeilenkommentar
```

- **Blockkommentare:** Blockkommentare beginnen mit „/*‘‘ und enden mit „*/‘‘. Alle Zeichen dazwischen zählen als Kommentar.

```
/*
Das ist
ein grosser
Blockkommentar
*/
```

Auch hierfür gibt es Regeln in den Code Conventions. Blockkommentare, die über mehrere Zeilen gehen, soll eine Leerzeile vorausgehen, um den Kommentar von den vorigen Zeilen abzusetzen. Der Kommentar selbst soll folgendes Aussehen haben:

```
/*
 * Here is a block comment.
 */
```

Eclipse versucht automatisch, einen Blockkommentar auf diese Weise zu formatieren. Ein weiterer, aus C bekannter Stil von Blockkommentaren ist:

```
/*#*****************************************************\
*                                                       *
*                Hervorgehobener Teil                   *
*                                                       *
\*****************************************************/
```

Solche Kommentare sind stark hervorgehoben und gliedern den Quelltext gut. Man beachte das Doppelkreuz (`#`) als drittes Zeichen der ersten Zeile. Hier darf man keinen Stern verwenden, weil Java den Kommentar sonst als Java-Dokumentationskommentar interpretieren würde. Aus mir nicht bekannten Gründen wird diese Form in den Code Conventions ausdrücklich missbilligt:
Comments should not be enclosed in large boxes drawn with asterisks or other characters.

1.3.1 Was soll kommentiert werden?

Auch hier gibt es Regeln aus den Java Code Conventions. Die wichtigsten sind frei übersetzt:

- Kommentare sollten einen Überblick über den Code geben und zusätzliche Information bieten, die nicht aus dem Code selbst ersichtlich ist.
- Erläuterungen nicht offensichtlicher Design-Entscheidungen sollten kommentiert werden, jedoch keine Informationen, die klar aus dem Code hervorgehen. Die Gefahr ist zu groß, dass bei Code-Änderungen die Kommentare nicht aktualisiert werden und in Widerspruch zum Code geraten.
- Die Anzahl der Kommentare gibt manchmal die schlechte Qualität des Codes wieder. Statt einen Kommentar hinzuzufügen, sollte man lieber erwägen, den Code besser und klarer zu schreiben.

Für uns bedeutet das zur Zeit:

1. Zu Beginn des Programms sollte ein Kommentar über den Zweck des Programms stehen.
2. Zwischen größeren Code-Abschnitten sollte ebenfalls ein Kommentar stehen (zum Beispiel: Laden der Daten / Verarbeiten der Daten).

Weitere Kommentare werden wir erst im späteren Verlauf des Kurses benötigen.

1.4 Aufruf von Funktionen

Zu Java gehört die sogenannte *Java-Klassenbibliothek* mit einer fast unüberschaubaren Anzahl von Möglichkeiten. Die meisten können wir erst nutzen, wenn wir die Funktionsweise von Klassen kennen. Daneben müssen wir uns auch mit Interfaces, abstrakten Klassen und Exceptions beschäftigt haben. Einige Möglichkeiten (die sogenannten *Funktionen* oder *statische Methoden*) können wir aber jetzt schon ohne große Vorbereitung nutzen.

1.4.1 Packages

Die Java-Klassenbibliothek ist in eine Anzahl von *packages* aufgeteilt. Um eine spezielle Klasse oder Methode (Funktion) zu nutzen, muss man ganz zu Beginn des Programms das Package *importieren*. Die Anweisung dazu steht noch vor der *class*-Anweisung und heißt:

```
import <Name des Packages>.*;
```

Eine einziges Package wird allerdings schon automatisch importiert: *java.lang*. Die meisten der nachfolgend beschriebenen Methoden befinden sich in java.lang, so dass man sich die Import-Anweisung

```
import java.lang.*;
```

sparen kann (sie schadet aber auch nicht).

1.4.2 Ausgabe von Daten auf den Bildschirm

`System.out.println`

Eine Methode haben wir schon kennengelernt: System.out.println. Das Argument wird auf dem Bildschirm ausgegeben, egal, ob man einen String oder einen primitiven Datentyp übergibt. Tatsächlich funktioniert es sogar bei beliebigen Objekten. Da die Klasse *System* in *java.lang* steht, müssen Sie auch nichts importieren.

`System.out.print`

Wie `System.out.println`, allerdings ohne automatischen Zeilenvorschub am Ende.

`System.out.printf`

Formatierte Ausgabe. Das Format entspricht dem *printf*-Kommando aus C und wird in Anhang A erläutert. Existiert erst ab Java 5.

1.4.3 Einlesen von Daten von der Tastatur

Das Einlesen von Daten von der Tastatur ging in den ersten Java-Versionen noch recht umständlich und wurde dann Schritt für Schritt vereinfacht. Darum finden sich in unterschiedlichen Büchern, je nach Erscheinungsdatum, verschiedene Wege zur Eingabe von Daten. Alle Methoden sollen hier der Vollständigkeit halber erwähnt werden. Nur die letzten beiden beschränken sich auf statische Methoden und sind zum jetzigen Zeitpunkt komplett zu verstehen. Als Beispiel wird jeweils eine Textzeile von der Tastatur in einen String eingelesen.

- Die älteste Methode benutzt einen *BufferedReader*.

```
String str;
BufferedReader inp = new BufferReader(new InputStreamReader(System.in));
str = inp.readLine();
```

- Ab Version 1.5 kann dies mit einem *Scanner* abgekürzt werden. Der Scanner hat deutlich mächtigere Befehle als der BufferedReader.

```
String str;
Scanner inp = new Scanner(System.in);
str = inp.nextLine();
```

- Ab Version 1.6 gibt es eine noch einfachere Möglichkeit:

```
String str;
str = System.console().readLine();
```

- Alternativ kann ab Version 1.2 ein grafisches Eingabefenster mit einer Eingabeaufforderung geöffnet und ein String aus diesem Fenster eingelesen werden. Hier wird nur eine einzige statische Methode benutzt. Beispiel:

```
String str;
str = JOptionPane.showInputDialog("Bitte Text eingeben.");
```

Die Klasse JOptionPane steht in der Package *javax.swing*. Als erste Zeile ist also

```
import javax.swing.*;
```

einzugeben. Ein vollständiges Programm lautet:

```
import javax.swing.*;

public class Eingabe {
  public static void main(String[] args) {
    String name;
    System.out.println("********");
    name = JOptionPane.showInputDialog("Bitte Name eingeben.");
    System.out.println(name);
    System.out.println("********");
  }
}
```

Wenn Sie sich an einem bislang nicht erklärbaren Parameter *null* nicht stören, können Sie auch Text in einem Fenster ausgeben:

```
JOptionPane.showMessageDialog(null, String text)
```

1.4.4 Umwandlung zwischen Zahlen und Strings

Die Methode

```
Integer.parseInt(String i)
```

verwandelt einen String in einen Integer, falls das geht. Ansonsten wird das Programm mit einer Fehlermeldung beendet. Das nachfolgende Beispiel liest einen String von Tastatur ein, wandelt ihn in eine Integer-Zahl um, erhöht die Zahl um 1 und gibt das Ergebnis auf dem Bildschirm aus.

```
import javax.swing.*;

public class Eingabe {
  public static void main(String[] args) {
```

```
    String name;
    int zahl;
    System.out.println("********");
    name = JOptionPane.showInputDialog("Bitte Zahl eingeben.");
    zahl = Integer.parseInt(name);
    System.out.println(zahl+1);
    System.out.println("********");
  }
}
```

Es gibt auch Long.parseLong(), Double.parseDouble(), Float.parseFloat() usw.. Alle Klassen Integer, Long, Double, Float (mit großen Anfangsbuchstaben) stehen in *java.lang*.
Will man andersherum Zahlen in einen String verwandeln, gibt es dazu die Funktion `String.valueOf`:

```
int a = 5;
String s = String.valueOf(a);
```

Oft wird auch der bereits beschriebene Trick verwendet, dass die Zahl zu einem Leerstring addiert wird:

```
int a = 5;
String s = "" + a;
```

1.4.5 Mathematische Funktionen: Klasse `Math`

In *Math* (*java.lang*) stehen ausschließlich statische Methoden. Zum Beispiel:

```
Math.sin
Math.sqrt
Math.PI     //Konstante PI
```

Beispiel:

```
import javax.swing.*;

public class Wurzel {
  public static void main(String[] args) {
    String name;
    int zahl;
    double wurzel;
    name = JOptionPane.showInputDialog("Bitte Zahl eingeben.");
    zahl = Integer.parseInt(name);
    wurzel = Math.sqrt(zahl);
    System.out.println("Die Wurzel aus "+zahl+" ist "+wurzel);
  }
}
```

1.4.6 Sonstiges

`System.exit`

Der Befehl

```
System.exit(int val);
```

beendet das Programm sofort. Der Wert *val* wird aus dem Java-Programm zurückgegeben und kann in einem Windows-Batch-Programm oder einer UNIX-Shell weiterverwendet werden.

1.4.7 Übergabeparameter

Die Parameter, die wir beim Funktionsaufruf mitgeben, heißen *Übergabeparameter*. Es kann einen Übergabeparameter geben (wie bei `System.out.println`) oder mehrere, wie im folgenden Beispiel:

```
double x = 3;
double y = 2;
double z = Math.pow(x,y);  //Berechnet 3 hoch 2 = 9
```

Es gibt auch Funktionen ganz ohne Übergabeparameter. Eine davon zeigt das nächste Beispiel:

```
//Berechnet eine Zufallszahl im Intervall [0,1[
double z = Math.random();
```

1.5 Schreiben eigener Funktionen

1.5.1 Notation

Aufgabenstellung

Nehmen wir an, wir benötigen eine Funktion, die den Abstand eines Punktes (x/y) im zweidimensionalen Raum zum Koordinatenursprung berechnet. Die mathematische Funktion ist:

$$f(x, y) = \sqrt{x^2 + y^2}$$

Die neue Funktion soll den Namen `getDistance` haben und wie folgt benutzt werden:

```
public static void main(String[] args) {
  double xK=3.2;
  double yK=5.7;

  double d = getDistance(xK, yK);
  System.out.println(d);
}
```

Definition einer eigenen Funktion

Die dazu nötige Funktion hat das folgende Aussehen:

```
public static double getDistance(double x, double y) {
  double res = Math.sqrt(x*x+y*y);
  return res;
}
```

Die erste Zeile ist die sogenannte *Kopfzeile*, *Signatur* oder *Deklaration* der Funktion. Die einzelnen Teile bedeuten:

- `public` bedeutet, dass die Funktion auch aus anderen Klassen aufgerufen werden kann (siehe Ende des Kapitels).
- `static` bedeutet, dass es sich um eine Funktion handelt und nicht um eine sogenannte *Methode*, die im Rahmen der Objektorientierung noch erläutert wird.
- `double` bezeichnet den Typ des Funktionsergebnisses bzw. des *Rückgabewerts* der Funktion.
- `getDistance` ist der *Funktionsname*.
- In Klammern stehen die beiden *Übergabeparameter*. Für jeden Parameter wird der Funktionstyp und der Variablenname angegeben. Die übergebenen Werte werden in diese Variablen kopiert[6] und können in der Funktion unter den entsprechenden Namen angesprochen werden.

Am Ende der Funktion wird der Rückgabewert mit dem Schlüsselwort `return` zurückgegeben.[7]

Fehlende Parameter

Es ist möglich, dass eine Funktion keinen Rückgabewert oder keine Übergabeparameter hat. Als Beispiel nehmen wir eine Funktion `printHello`, die „Hello“ auf dem Bildschirm ausgibt, aber weder Übergabe- noch Rückgabewerte hat.

```
public static void printHello() {
  System.out.println("Hello");
}
```

Die Funktion wird einfach mit `printHello` aufgerufen. Das Schlüsselwort `void` steht für „kein Rückgabewert“. Der Platz zwischen den Klammern für die Übergabeparameter bleibt einfach leer.

[6] Das trifft nicht immer zu. Im jetzigen Stand können wir aber einfach davon ausgehen.

[7] Wir werden sehen, dass es bei Verzweigungen auch mehrere `return`-Anweisungen innerhalb einer Funktion geben kann.

Aufruf aus anderen Klassen

Wird eine Funktion aus einer anderen Klasse (d.h. von einer anderen Datei aus) aufgerufen, muss der Klassenname vorangestellt werden. Nehmen wir an, die Funktion `getDistance` stände in einer Klasse `Geom`, dann wäre der Aufruf aus einer anderen Klasse heraus z.B.

```
double r = Geom.getDistance(3, 5.8);
```

1.5.2 Vorteile

Vermeidung von doppeltem Code

Es liegt auf der Hand, das ein Programm, welches häufig Distanzen berechnet, durch die Funktion `getDistance` kürzer und besser verständlich wird. Hinzu kommt, dass, falls sich in der Distanzberechnung ein Fehler findet, man nur *eine* Stelle im Code ändern muss.

Abgrenzung von Implementierung und Test

Im momentanen Stadium haben Funktionen für uns einen weiteren Vorteil. Wir können bei der Bearbeitung von Aufgabenstellungen leicht die eigentliche Funktion und die *Funktionsanwendung* bzw. den *Funktionstest* (ein Test simuliert eine Anwendung) trennen. Bei der Programmierung wird im Regelfall jede Funktion einzeln getestet. Das kann so aussehen:

```
public class Geom {

  //Zu testende Funktion
  public static double getDistance(double x, double y) {
    double res = Math.sqrt(x*x+y*y);
    return res;
  }

  public static void main(String[] args) {
    //Funktionstest fuer die Funktion getDistance
    double xK=1.2;
    double yK=3.5;

    double d = getDistance(xK, yK);
    //Erwartetes Ergebnis: 3.7
    System.out.println(d);
}
```

1.5.3 Nachteile

Eine Tatsache müssen wir im Hinterkopf behalten. Mit Funktionen programmieren wir *prozedural*. So programmierte man jahrzehntelang nahezu ausschließlich und benutzte dazu Sprachen, wie C oder Pascal. Zu Java gehört aber die

objektorientierte Programmierung. Daher werden Funktionen bald gegenüber objektorientierten Strukturen deutlich in den Hintergrund treten.

1.6 Lineare Programme

1.6.1 Eingabe - Verarbeitung - Ausgabe (EVA)

Das EVA-Prinzip ist ein wichtiger Grundsatz in der Softwareentwicklung. Er bedeutet, dass ein Programm aus den drei Teilen *Eingabe*, *Verarbeitung* und *Ausgabe* besteht, die in dieser Reihenfolge bearbeitet werden. Obwohl das Prinzip sehr offensichtlich scheint, muss man bei komplexeren Programmen aufpassen, dass man nicht dagegen verstößt (und beispielsweise versucht, Daten zu vearbeiten, ohne dass welche eingegeben wurden). Mit dem bis jetzt erworbenen Kenntnissen können wir kleine Programme schreiben, die das EVA-Prinzip voll umsetzen. Betrachten wir als Beispiel ein kleines Programm, das die Distanzfunktion aus dem letzten Abschnitt benutzt:

```
public class Divisionsrest {

    public static double getDistance(double x, double y) {
      double res = Math.sqrt(x*x+y*y);
      return res;
    }

    public static void main(String[] args) {

          //Eingabe
          String zeile = JOptionPane.showInputDialog("x-Wert: ");
          double x = Double.parseDouble(zeile);
          zeile = JOptionPane.showInputDialog("y-Wert: ");
          double y = Double.parseDouble(zeile);

          //Verarbeitung
          double dist = getDistance(x,y);

          //Ausgabe
          System.out.println("Die Distanz zum Ursprung ist "+dist);
        }
}
```

1.6.2 Struktogramme

Ein Struktogramm ist ein Konzept, ein Programm zu visualisieren. Struktogramme werden nach den Entwicklern Isaac Nassi und Ben Shneiderman auch *Nassi-Shneiderman-Diagramme* genannt (im Englischen wird das dann oft zu „NS diagram“ abgekürzt). Struktogramme bestehen aus Elementen, die nach DIN 66261 genormt sind. Es gibt lediglich 11 solcher Elemente in der DIN. Einige andere sind gängig, aber nicht in der DIN-Norm enthalten.

Nassi-Shneiderman-Diagramme wurden 1972/73 entwickelt, die DIN-Norm wurde 1985 aufgestellt. Die Weiterentwicklungen der letzten 20 Jahre im Bezug auf Programmiersprachen, vor allem die Objektorientierung, können nicht mit Nassi-Shneiderman-Diagrammen wiedergegeben werden.
In der Softwareentwicklung werden Nassi-Shneiderman-Diagramme eher selten eingesetzt, da normaler Programmcode einfacher zu schreiben und zu verändern ist: Korrigiert man einen Fehler oder macht eine Ergänzung, muss man ein Nassi-Shneiderman Diagramm in der Regel komplett neu zeichnen. Daher werden sie in der Regel *nach* Erstellung des Codes von entsprechenden Hilfsprogrammen automatisch erzeugt und dienen der Dokumentation.
In der Lehre werden Struktogramme dagegen häufig verwendet, damit Schüler den Aufbau logischer Abläufe, die für die Programmierung nötig sind, trainieren können. Die Erstellung von Struktogrammen ist immer noch Bestandteil vieler schulischer Abschlussprüfungen. Das Elemente von Struktogrammen sind in Anhang D zu finden.

1.6.3 Flussdiagramm / Programmablaufplan (PAP)

Flussdiagramme (auch Programmablaufpläne genannt) stammen aus den 1960er Jahren und sind damit noch älter als Struktogramme. Objektorientierte Programmkonzepte lassen sich auch hier nicht darstellen. Der Vorteil gegenüber Struktogrammen ist, dass sich kleinere Änderungen leichter einbauen lassen, was besonders für die ersten Konzepte von Vorteil ist. Demgegenüber neigen Flussdiagramme dazu, zum unübersichtlichen „Pfeilsalat“ zu degenerieren. Auch lassen sich Struktogramme schneller und direkter in Programmcode übersetzen. Zum Beispiel lassen sich Schleifen in Flussdiagrammen nicht auf den ersten Blick identifizieren. Flussdiagramme sind nach DIN 66001 genormt. Die Elemente von Flussdiagrammen sind ebenfalls in Anhang D zu finden.

1.6.4 Aktivitätsdiagramm

Eine Weiterentwicklung des Flussdiagramms ist das *Aktivitätsdiagramm*. Das Aktivitätsdiagramm (engl. activity diagram) ist eine der Diagrammarten in der Unified Modeling Language (UML). Mit Aktivitätsdiagrammen lassen sich beispielsweise auch Exceptions oder parallele Programme modellieren. Allerdings gelten die Hauptnachteile der Flussdiagramme auch für Aktivitätsdiagramme. Ausführlich werden Aktivitätsdiagramme in der Vorlesung „Software Engineering“ behandelt. Wir werden uns hier auf die sehr ähnlichen, aber einfacheren Flussdiagramme beschränken.

Kapitel 2

Kontrollstrukturen

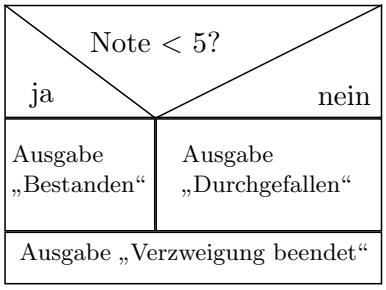
2.1 Auswahl (Selektion)

2.1.1 Entweder-oder-Entscheidungen

Ein Java-Programm kann, abhängig von einem Kriterium, eine Entweder-oder-Entscheidung treffen, d.h. entweder einen bestimmten Programmteil „A" ausführen oder einen anderen bestimmten Programmteil „B". Der nötige Code lässt sich sehr intuitiv verstehen. Sehen wir uns als Beispiel den Code an, der, abhängig vom Wert der Variablen `note`, den Text „Bestanden" oder „Durchgefallen" ausgibt:

```
if (note<5) {
  System.out.println("Bestanden");
} else {
  System.out.println("Durchgefallen");
}
System.out.println("Verzweigung beendet");
```

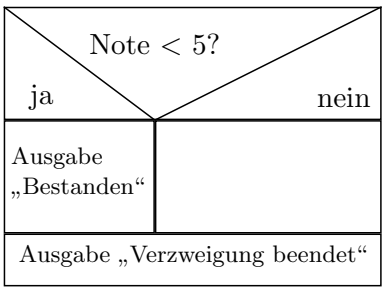
Zuvorderst steht ein neues Schlüsselwort: `if` (wenn). Dann folgt in Klammern das Kriterium: „Wenn `note` kleiner als 5 ist". Wenn das der Fall ist, werden alle Anweisungen in dem folgenden, von geschweiften Klammern eingeschlossenen Block ausgeführt. Dann folgt ein weiteres Schlüsselwort `else` (sonst), gefolgt von einem weiteren Block, der ausgeführt wird, wenn `note` eben nicht kleiner als 5 ist. Danach ist die Verzweigung beendet, das heißt, der folgende Code wird auf jeden Fall ausgeführt und hängt nicht mehr von der if-Bedingung ab. Im Struktogramm sieht das so aus:



Man kann auch den *else*-Teil komplett weglassen. Wenn man zum Beispiel die Meldung „Durchgefallen“ nicht braucht, kann man schreiben:

```
if (note<5) {
  System.out.println("Bestanden");
}
System.out.println("Verzweigung beendet");
```

Das ist die sogenannte *Einseitige Auswahl* im Gegensatz zur *Zweiseitigen Auswahl*, die wir eben kennengelernt haben. Das Struktogramm hat jetzt folgendes Aussehen:



Wenn wir anders herum den „ja-Zweig” (korrekt würde man sagen: *if-Zweig*) weglassen und nur den „nein-Zweig” (korrekt: *else-Zweig*) implementieren wollen, wird es etwas schwieriger. Wir wandeln unser Beispiel so ab, dass nur der Text „Durchgefallen” ausgegeben werden soll. Möglich wäre folgendes:

```
if (note<5) {  //kein guter Stil
} else {
  System.out.println("Durchgefallen");
}
```

Das läuft zwar, ist aber kein guter Stil. Besser ist es, die Bedingung umzudrehen:

```
if (note>=5) {
  System.out.println("Durchgefallen");
}
```

Kommen wir schließlich noch zur Bedingung, die in den Klammern steht. Was darin steht, ist das Ergebnis der Rechenoperation `note<5`. Die Rechenoperation ist eine sogenannte *Vergleichsoperation* und das Ergebnis ist ein *Wahrheitswert*. Ein Wahrheitswert kann nur die beiden Werte *wahr* und *falsch* annehmen.

2.1.2 Boolean-Variablen

In Java gibt es einen eigenen Variablentyp für Wahrheitswerte. Dieser Typ hat den Namen `boolean` (deutsch: Boolesche Variable, benannt nach dem Mathematiker George Boole). Sie kann die Werte `true` (wahr) oder `false` (falsch) annehmen.

```
boolean b;
b = true;
b = false;
System.out.println(b);   //Ergibt false
```

2.1.3 Vergleichsoperatoren

Vergleichsoperationen liefern als Ergebnis einen Wahrheitswert:

```
int a = 5;
boolean b = a > 4;
```

Der Ausdruck `a > 4` hat als Ergebnis *true*, falls $a > 4$ und *false*, falls $a \leq 4$. In diesem Beispiel hätte `b` also den Wert *true*. Es gibt 6 Vergleichsoperatoren:

```
a > b;
a >= b;
a == b;  //Gleichheitsoperator
a != b;  //Ungleichheitsoperator
a <= b;
a < b;
```

Die Bezeichnungen sind selbsterklärend. Nur zum Gleichheitsoperator == ist eine Erklärung angebracht. Zunächst ein Beispiel:

```
int a = 5;
boolean b = a == 4;
boolean c = a == a;
```

`b` ist *false*, denn $5 \neq 4$. `c` ist immer *true*, gleichgültig, welchen Wert `a` hat. Der Gleichheitsoperator besteht aus zwei Ist-Gleich-Zeichen, da das einzelne Ist-Gleich-Zeichen schon vom Zuweisungsoperator belegt ist.
Ein beliebter Fehler ist, statt dem Gleichheitsoperator „==" den Zuweisungsoperator „=" zu benutzen. In Java führt das in den meisten Fällen glücklicherweise gleich zu einem Compilerfehler, so dass man die Verwechslung schnell bemerkt. Sollten sie dagegen C oder C++ Programme erstellen, kann die Verwechslung Ihnen schwer auffindbare Fehler bescheren.

2.1.4 Boolean-Werte als Bedingung für Verzweigungen

Verzweigungen erfolgen immer aufgrund eines boolean-Wertes. Dies kann der Inhalt einer Variablen, das Ergebnis einer Vergleichsoperation oder einer Logikoperation (siehe nächster Abschnitt) sein.

```
boolean x = (b!=0);
if (x) {  ...  }
```

oder

```
if (b!=0) { ... }
```

2.1.5 Logikoperatoren

Logikoperatoren verknüpfen ein bis zwei boolean-Werte zu einem neuen boolean-Wert. Der einfachste Logik-Operator ist das Ausrufezeichen, das die Bezeichnung „nicht" trägt. Dieser Operator negiert den Wert einer boolean Variablen.

```
boolean b = false;
boolean c = ! b;  //ergibt true
```

Weiterhin gibt es die Operatoren „und" (`&&`), „oder" (||) und „exklusiv-oder" (xor) (`^`). Dabei gilt:

- `a && b` ist `true`, falls `a` *und* `b` `true` sind.
- `a || b` ist `true`, falls `a` *oder* `b` `true` ist.
- `a ^ b` ist `true`, falls `a` und `b` nicht die gleichen Werte haben (xor-Operation).

Beispiel:

```
boolean a = true;
boolean b = false;
boolean c = a && b;  //Ist false, da b = false
boolean d = a || b;  //Ist true
boolean e = a ^ b;   //Ist true, da a ungleich b
```

Sollten sie aus Versehen einmal `&` statt `&&` oder | statt || schreiben, wird das in aller Regel auch funktionieren. Der Unterschied zwischen `&` und `&&` bzw. zwischen | und || soll hier nur angedeutet werden. Bei den verdoppelten Operatoren wird der zweite Operand nicht mehr ausgewertet, wenn das Ergebnis nach dem ersten Operanden schon feststeht. Bei

```
int a = 1;
int b = 0;
boolean c = (b!=0) && (a/b == 1);
boolean d = (b!=0) & (a/b == 1);
```

ergibt sich `c=false`, die Bestimmung von `d` bricht wegen einer Division durch 0 ab. Wir werden immer `&&` und || verwenden.

2.1.6 Operatorhierarchie

Wir kennen jetzt insgesamt die Operatoren

```
+ - * / % < > <= >= == != ! && || ^
```

Um die Abarbeitungs-Reihenfolge zwischen allen diesen Operatoren festzulegen, wird die Regel „Punkt- vor Strichrechnung" zu einer Operatorhierarchie erweitert. Weiter oben stehende Operatoren haben Vorrang vor weiter unten stehenden.

```
*  /  %
+  -
<  >  <= >=
== !=
^
&&
||
```

Der Ausdruck

```
boolean b = a % 5 > 2 && a > 0
```

prüft, ob der Rest von $\frac{a}{5}$ größer als 2 und gleichzeitig $a > 0$ ist, ergibt also *true* bei $a = 3, 4, 8, 9, 13, 14, \ldots$.

2.1.7 Weglassen der Klammern

Die öffnende und die schließende geschweifte Klammer darf in der if-Anweisung weggelassen werden. **Dies ist jedoch immer schlechter Stil** und kann leicht zu schwer auffindbaren Fehlern führen. Werden die Klammern weggelassen, besteht der if-Block aus der Zeile, die der if-Anweisung folgt und der else-Block besteht aus der Zeile, die der else-Anweisung folgt. Beispiel:

```
if (wert%2==0)
    System.out.println("Die Zahl ist gerade");
```

Die Gefahr darin zeigt sich in folgendem Codeausschnitt:

```
if (wert%2==0)
    System.out.println("Die Zahl ist gerade");
    System.out.println("Die Zahl ist durch 2 teilbar");
```

Entgegen dem Anschein wird die zweite println-Zeile auch bei ungeraden Zahlen ausgeführt, denn Java verwendet wegen der fehlenden geschweiften Klammern nur die erste println-Zeile für den if-Block. Dies ist ein nachträglich schwer zu findender Fehler, der von vornherein vermieden werden kann, wenn man konsequent Klammern für den if und den else-Block setzt.
Ein weiteres schwer auffindbarer Fehler ist:

```
if (wert%2==0);
    System.out.println("Die Zahl ist gerade");
```

Diesen Code muss man folgendermaßen interpretieren: Falls die Bedingung wahr ist, werden die Anweisungen bis zum nächsten Semikolon ausgeführt, also bis zum Semikolon am Ende der if-Zeile. Anschließend ist die if-Verweigung zu Ende. Die println-Anweisung wird also immer ausgeführt, gleichgültig ob `wert` gerade oder ungerade ist.

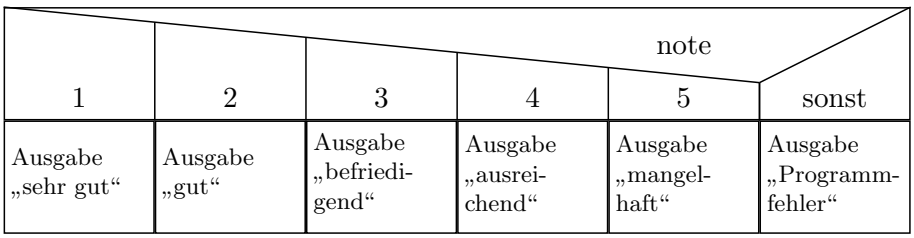
2.1.8 if-else-Kaskaden

In einem Sonderfall lässt man teilweise die geschweiften Klammern aber doch weg. Manchmal gibt es mehr als zwei Fälle, die unterschiedlich behandelt werden müssen. In diesem Fall schachtelt man mehrere if-Anweisungen ineinander und erhält die sogenannte *if-else-Kaskade*. Im nachfolgenden Beispiel bauen wir die Ausgabe einer Schulnote so weit aus, dass für jede Note ein individueller Text ausgegeben wird.

```
if (note==1) {
  System.out.println("sehr gut");
} else if (note==2) {
  System.out.println("gut");
} else if (note==3) {
  System.out.println("befriedigend");
} else if (note==4) {
  System.out.println("ausreichend");
} else if (note==5) {
  System.out.println("mangelhaft");
} else {
  System.out.println("Fehler im Programm");
}
```

2.1.9 Mehrseitige Auswahl

In allen neueren Sprachen, gibt es eine Verzweigung, die, abhängig von einer Integer-Variablen, einen von mehreren Programmblöcken anspringt. Das Notenprogramm, das im vorigen Kapitel mit einer if-else-Kaskade gelöst wurde, ist ein gutes Beispiel dafür. Das Struktogramm dazu ist:



switch-Anweisung

Die entsprechende Anweisung heißt in Java *switch-Anweisung*. In anderen Sprachen ist sie als case- oder select-Anweisung bekannt. Sie beginnt in Java mit einer Zeile `switch`, gefolgt von der Variablen, deren Wert für die Verzweigung herangezogen wird:

```
switch (note) {
```

Dann folgt für jede Note ein sogenannter *case-Block*. Die erste Zeile eines case-Blocks wird eingeleitet durch das Schlüsselwort `case`, gefolgt von dem Wert, für den der Block ausgeführt werden soll und einem Doppelpunkt:

```
case 1:
```

Dann kommen die Anweisungen für den Block. Es gibt keine geschweiften Klammern. Ein Block wird mit dem Befehl `break` abgeschlossen.

```
switch(note) {
  case 1:   System.out.println("sehr gut");
            break;
```

```
    case 2:   System.out.println("gut");
              break;
    case 3:   System.out.println("befriedigend");
              break;
    case 4:   System.out.println("ausreichend");
              break;
    case 5:   System.out.println("mangelhaft");
              break;
    default:  System.out.println("Fehler.");
  }//switch
```

Am Ende der switch-Anweisung darf man noch einen sogenannten *default-Block* unterbringen, der immer dann ausgeführt wird, wenn keiner der vorigen case-Blöcke zutreffend war. Man kann ihn auch weglassen. Dann wird statt dessen der switch-Block komplett übersprungen.
Die switch-Anweisung ist in Java für byte, short, char und int erlaubt. Strings funktionieren erst ab Java 7. long oder andere Datentypen sind in Java nicht möglich (anders als in C# oder Skriptsprachen).
Die switch-Anweisung hat ihre Tücken. Vor allem darf man das `break` am Ende nicht vergessen. Man kann das so verstehen: Die case- Zeilen sind *Ansprungstellen*. Das heißt, wenn jetzt zum Beispiel die Note gleich 2 ist, wird die Zeile mit `case 2` angesprungen. Dann läuft das Programm Zeile für Zeile weiter. Wird ein `break` erreicht, springt das Programm aus dem switch-Block heraus.
Haben wir jetzt beispielsweise das `break` nach `case 2` vergessen, dann läuft das Programm einfach Zeile für Zeile weiter, bis der switch-Block zu Ende ist oder ein `break` erreicht wurde. In unserem Beispiel würde dann

```
gut
befriedigend
```

ausgegeben.
Diesen Effekt kann man durch geschickte Programmierung auch ausnutzen und Zweige für ganze Bereiche definieren. Im folgenden Beispiel wird bei den Noten 1-4 der Text „bestanden” ausgegeben. Für diese Fälle weicht man besser auf die neue Java `switch`-Syntax aus, die ab Java 14 möglich ist und im nachfolgenden Abschnitt kurz erklärt wird.

```
  switch(note) {
    case 1:
    case 2:
    case 3:
    case 4:   System.out.println("bestanden");
              break;
    case 5:   System.out.println("mangelhaft");
              break;
    default:  System.out.println("Fehler.");
  }//switch
```

Eine andere Möglichkeit, Bereiche bei switch-Anweisungen anzugeben, gibt es in Java (anders als z.B. in Pascal) leider nicht. Wenn man das folgende Beispiel mit einer switch-Verzweigung umsetzen wollte, bräuchte man mehrere hundert case-Anweisungen. In diesem Fall greift man besser wieder auf if-else-Kaskade zurück.

```
if (windgeschwindigkeit<=2) {
  System.out.println("Windstille");
} else if (windgeschwindigkeit<=45) {
  System.out.println("schwacher Wind");
} else if (windgeschwindigkeit<=75) {
  System.out.println("starker Wind");
} else if (windgeschwindigkeit<=120) {
  System.out.println("Sturm");
} else if (windgeschwindigkeit<=200) {
  System.out.println("Orkan");
} else {
  System.out.println("Messgeraet kaputt, weil Wind zu stark");
}
```

Neue switch-Anweisung ab Java 14

Die switch-Anweisung aus dem vorigen Abschnitt hat Java aus C übernommen. Sie ist auch in anderen Sprachen weit verbreitet. Ab Java 14 gibt es parallel dazu noch eine andere Syntax, die mehrere Varianten hat. Das Noten-Beispiel aus dem vorigen Abschnitt kann man mit der neuen Syntax wie unten gezeigt optimieren. Für größere Wertebereiche der `case`-Abschnitte ist aber auch die neue Syntax nicht geeignet.

```
switch(note) {
  case 1, 2, 3, 4 -> System.out.println("bestanden");
  case 5          -> System.out.println("mangelhaft");
  default         -> System.out.println("Fehler.");
}//switch
```

2.2 Schleifen(Iteration)

Beispiel zur Einführung

Es soll ein Programm geschrieben werden, das einen Countdown simuliert. Es soll also nacheinander die Werte von 10 bis 0 auf dem Bildschirm ausgeben. Mit den bisherigen Mitteln würden wir dies so lösen:

```
System.out.println(10);
System.out.println(9);
System.out.println(8);
...
System.out.println(1);
System.out.println(0);
```

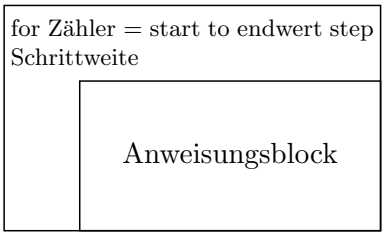
Es werden 10 fast gleiche Zeilen ausgeführt. Da muss es doch eine kürzere Lösung geben. Wir bräuchten eine Anweisung wie:

```
Fuehre die Zeile System.out.println(i)
nacheinander mit i=10,9,8,...,1,0 aus.
```

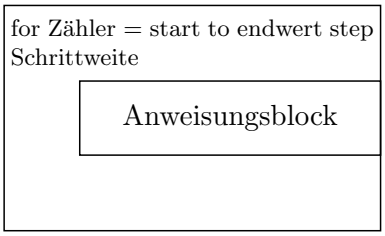
Dies führt zur häufigsten Schleifenvariante: der Zählschleife.

2.2.1 Die Zählschleife

Die Zählschleife ist eine Schleifenart, bei der von Anfang an feststeht, wieviele Wiederholungen der Schleife ausgeführt werden. Es gibt einen Zähler (Laufvariable) der von einem Anfangswert bis zu einem Endwert läuft und sich bei jedem Durchlauf um einen festen Betrag ändert. Das Struktogramm der Zählschleife ist:



Oft wird auch folgende Variante benutzt:



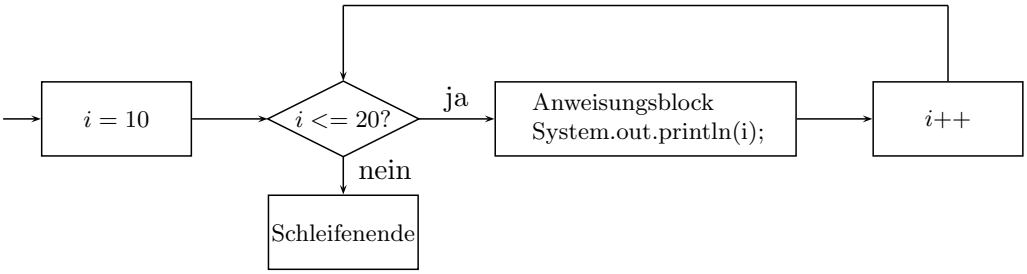
Zählschleifen werden in Java mit dem Schlüsselwort `for` eingeleitet.

```
for (i=10; i<=20; i++) {
  System.out.println(i);
}
```

Dem Schlüsselwort folgt eine Parameterliste, die in einer runden Klammer zusammengefasst ist und aus 3 Teilen besteht, die jeweils durch ein Semikolon getrennt sind. Die 3 Teile sind:

1. Initialisierung der Laufvariablen.
2. Abbruchbedingung (kein Abbruch, solange die Bedingung erfüllt ist).
3. Veränderung der Laufvariablen.

Die Reihenfolge, in der die Teile abgearbeitet werden, veranschaulicht das folgende Flussdiagramm:



Das Countdown-Programm würde damit wie folgt aussehen:

```
int i;
for (i=10; i>=0; i--) {
  System.out.println(i);
}
```

2.2.2 Besonderheiten bei for-Schleifen

Änderung der Laufvariablen im Schleifenkörper

Es ist möglich, die Laufvariable im Schleifenkörper, also zwischen den geschweiften Klammern, zu ändern. Ein Beispiel dafür ist:

```
for (i=10; i>=0; i++) {
  i=i-2;
  System.out.println(i);
}
```

Damit ist die Schleife aber keine reine Zählschleife mehr. In anderen Sprachen (z.B. Pascal) ist das auch verboten. In Java ist es möglich, aber schlechter Programmierstil. Natürlich ist es beim obigen Beispiel besonders unsinnig, aber man sollte es generell unterlassen und lieber auf eine while-Schleife (folgt in Kürze) ausweichen.

Deklaration der Laufvariablen in der Schleife

Es ist möglich, die Laufvariable im Schleifenkopf zu deklarieren:

```
for (int i=10; i>=0; i--)
```

Dann ist die Laufvariable nur in der Schleife gültig und kann nach Beendigung der Schleife nicht mehr angesprochen werden. Diese Form ist die üblichste Form einer for-Schleife.

Schleife ohne geschweifte Klammern

Die Anweisung

```
for (int i=10; i>=0; i--);
```

ist eine leere Schleife. Die Variable i wird von 10 auf 0 heruntergezählt.

Endlosschleife

Die Anweisung

```
for(;;)
```

ist eine Endlosschleife. Der Anweisungsblock wird endlos wiederholt. Da innerhalb des Anweisungsblocks mit `break` die Schleife verlassen werden kann (siehe dazu die folgenden Kapitel), kann diese Anweisung sinnvoll eingesetzt werden. Dagegen ist

```
for(;;);
```

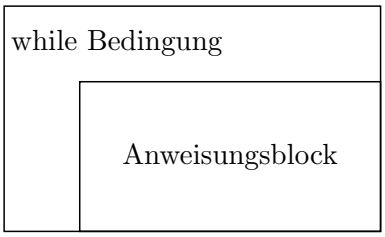
eine leere Endlosschleife, die fortgesetzt durchlaufen wird. Aus Benutzersicht rührt sich dann gar nichts mehr, bis der Benutzer das Programm abbricht. Man sagt, das Programm hat sich *aufgehängt*.

2.2.3 Schleife mit Anfangsabfrage (Kopfgesteuerte Schleife)

Bei manchen Schleifen steht zu Anfang die Anzahl der Durchläufe noch nicht fest. Es kann sein, dass es mehrere Abbruchbedingungen gibt oder dass die Laufvariable ihre Werte unvorhersehbar verändern kann. Hier benutzt man entweder die kopfgesteuerte oder die fußgesteuerte Schleife. Die kopfgesteuerte Schleife hat das Aussehen

```
while(Bedingung) {
    //Anweisungs-Block
}
```

Das bedeutet, dass der Anweisungsblock ausgeführt wird, solange die Bedingung in der while-Zeile den Wert `true` ergibt. Das entsprechende Struktogramm hat das Aussehen:



Sinnvolle Beispiele sind bereits etwas komplizierter. Wir nehmen ein Countdown-Programm, dass von 100 rückwärts bis 10 zählt, aber alle durch 7 teilbaren Zahler auslässt:

```
int i=100;
while (i>=10) {
  System.out.println(i);
  i--;
  if (i%7==0) {
```

```
    i--;  //Durch 7 teilbare Zahlen überspringen
  }
}
```

for- und while-Schleifen

Im vorigen Kapitel wurde erwähnt, dass for-Schleifen in C bzw. Java über reine Zählschleifen hinaus einsetzbar sind. Tatsächlich sind sie nur eine Kurzform für eine while-Schleife. Die Schleifen:

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

und

```
int i=0;
while (i<10) {
  System.out.println(i);
  i++;
}
```

enstsprechen sich exakt. for- und while-Schleifen lassen sich immer gegenseitig umwandeln. Es gibt daher eine Grundregel, die sagt:

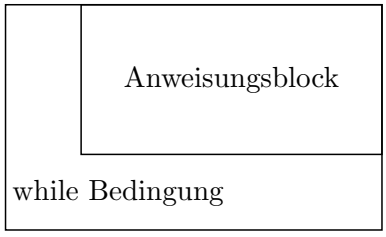
for-Schleifen werden nur bei wirklichen Zählschleifen eingesetzt, ansonsten werden kopf- oder fußgesteuerte Schleifen benutzt.

2.2.4 Schleife mit Endabfrage (Fußgesteuerte Schleife)

Die Schleife mit Endabfrage ähnelt der Schleife mit Anfangsabfrage. Allerdings wird die fußgesteuerte Schleife *mindestens einmal durchlaufen*, während demgegenüber die kopfgesteuerte Schleife gar nicht durchlaufen wird, wenn die Anfangsbedingung vor dem 1. Durchlauf nicht erfüllt ist. Die Schleife mit Endabfrage hat folgendes Aussehen:

```
do {

  //Anweisungsblock

} while (Bedingung)
```

Das Struktogramm ist:



Das Countdown-Beispiel hat hier folgendes Aussehen:

```
int i=10;
do {
  System.out.println(i);
  i--;
} while (i>=0);
```

In C und Java wird diese Schleife auch *do-while-Schleife* genannt, in Unterscheidung zur kopfgesteuerten *while-Schleife*. In Pascal spricht man stattdessen von einer *repeat-until-Schleife*. Da die fußgesteuerte Schleife sich immer mit einer kopfgesteuerten nachbilden lässt, besitzen manche Sprachen (z.B. Fortran, Python) keine fußgesteuerte Schleife.

2.2.5 break und continue

continue

Innerhalb einer Schleife kann mit dem Befehl *continue* der aktuelle Schleifendurchlauf abgebrochen werden, d.h. das Programm wird mit dem Beginn des nächsten Schleifendurchlaufs fortgesetzt. Die *continue*-Anweisung kann in allen Schleifenvarianten benutzt werden. Das folgende Countdown-Programm ist mit *continue* so abgewandelt, dass die Zahl 3 ausgelassen wird.

```
for (int i=10; i>=0; i--) {
  if (i==3) {
    continue;
  }
  System.out.println(i);
}
```

break

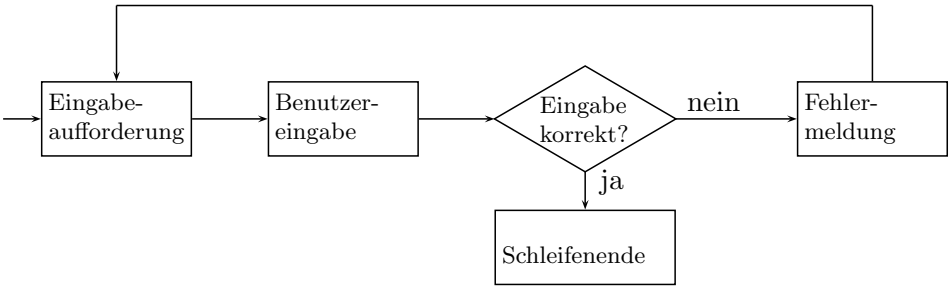
Der *break*-Befehl bewirkt, dass eine Schleife komplett abgebrochen wird. Das Programm

```
for (int i=10; i>=0; i--) {
  if (i==3) {
    break;
  }
  System.out.println(i);
}
```

zählt nur bis zur Zahl 4 herunter.

„Mittengesteuerte" Schleifen

In manchen Fällen möchte man gerne die Abbruchbedingung in der Mitte der Schleife testen. Mit Hilfe eines Flags (einer Boolean-Variablen) kann man dies mit einer fußgesteuerten Schleife nachbilden. Ein typisches Beispiel ist die Behandlung einer Benutzereingabe, die im folgenden Flussdiagramm erläutert ist.



Mit einem Flag und einer fußgesteuerten Schleife ergibt das:

```
boolean korrekt = false;
String dreiB;

do {
  dreiB = JOptionPane.showInputDialog("Bitte 3 Buchstaben eingeben: ");
  if (dreiB.length() == 3) {
    korrekt = true;
  } else {
    System.out.println("Fehler bei der Eingabe.");
  }
} while (!korrekt);
```

Kürzer (aber für manche Puristen unsauber) geht es mit einer Endlosschleife und der break-Anweisung. Als Syntax für Endlosschleifen haben wir schon

```
for (;;) {
  //Anweisungsblock
}
```

kennengelernt. Eine Alternative ist:

```
while(true) {
  //Anweisungsblock
}
```

Die Eingaberoutine sieht jetzt wie folgt aus:

```
String dreiB;

while(true) {
  dreiB = JOptionPane.showInputDialog("Bitte 3 Buchstaben eingeben: ");
  if (dreiB.length() == 3) {
    break;
  }
  System.out.println("Fehler bei der Eingabe.");
}
```

2.2.6 Mehrere verschachtelte Schleifen

Mehrere verschachtelte Schleifen sind möglich, wie am nachfolgenden Beispiel mehrerer verschachtelter for-Schleifen zu sehen ist. Es gibt ein Dreieck aus Sternen aus:

```
*
**
***
****
*****
```

Der Code ist:

```
public class Dreieck {
  public static void main(String[] args) {
    int max=5;
    for (int i=0; i<max; i++) {
      for (int j=0; j<=i; j++) {
        System.out.print("*");
      }
      System.out.println();
    } //for i
  } //main
} //class
```

break- und *continue*-Befehle in verschachtelten Schleifen wirken sich nur auf die jeweils innerste Schleife aus. Soll sich ein break oder continue direkt auf eine äußere Schleife auswirken, muss diese Schleife mit einem Label versehen werden. Das folgende Beispiel gibt das kleine Einmaleins aus, bis eine Zahl > 50 auftaucht (also bis `9*6=54`).

```
outer:
for (int i=1; i<=10; i++) {
  inner:
  for (int j=1; j<=10; j++) {
    System.out.printf("%d*%d=%2d\n",i,j,i*j);
    if (i*j>50) {
      break outer;
    }
  }
}
```

2.2.7 Aufzählungsschleife

Der letzte Schleifentyp, die Aufzählungs- oder foreach-Schleife, müssen wir hinter das nächste Kapitel verschieben, da wir dazu Felder benötigen.

2.2.8 Geltungsbereich von Variablen

Variablen können an jeder Stelle im Code deklariert werden, also auch innerhalb von Verzweigungen und Schleifen. Solche Variablen sind allerdings auch nur innerhalb der jeweiligen Schleife oder Verzweigung gültig. Außerhalb sind sie nicht mehr ansprechbar.

```
while(true) {
  int i=0;
  break;
}
System.out.println(i);     //Compiler-Fehler
```

```
int i=0;
while(true) {
  i=1;
  break;
}
System.out.println(i);     //i=1
```

```
{
  int i=1;
}
System.out.println(i);     //Compiler-Fehler
```

Die Laufvariable von for-Schleifen kann innerhalb der for-Anweisung deklariert werden. Diese Variable ist nur innerhalb der for-Schleife gültig.

```
Java, C++, C#

for (int i=1; i<=10; i++) {
   System.out.println(i);
}
System.out.println(i);     //Compiler-Fehler

---------------------------

int i;
for (i=1; i<=10; i++) {
   System.out.println(i);
}
System.out.println(i);     //ergibt 11
```

Eine typische Pascal-Falle tritt in dieser Art in Java nicht auf:

```
Pascal-Falle:
FOR i:=1 TO 10 DO BEGIN
  writeln(i);
END;
writeln(i);   (* i ist unbestimmt, nicht unbedingt=11 *)
```

2.2.9 Tabellen

Die Ausgabe von Tabellen wird sehr häufig gebraucht, daher hier ein einfaches Beispiel. Wir wollen eine Logarithmus-Tabelle mit den Logarithmen von 1-20 in Schritten von 0,1 ausgeben.

```
public class Logarithmus {
  public static void main(String[] args) {
    for (double i=1; i<=20; i+=0.1) {
      System.out.printf("%4.1f      %7.5f\n",i,Math.log(i));
    }
  }
}
```

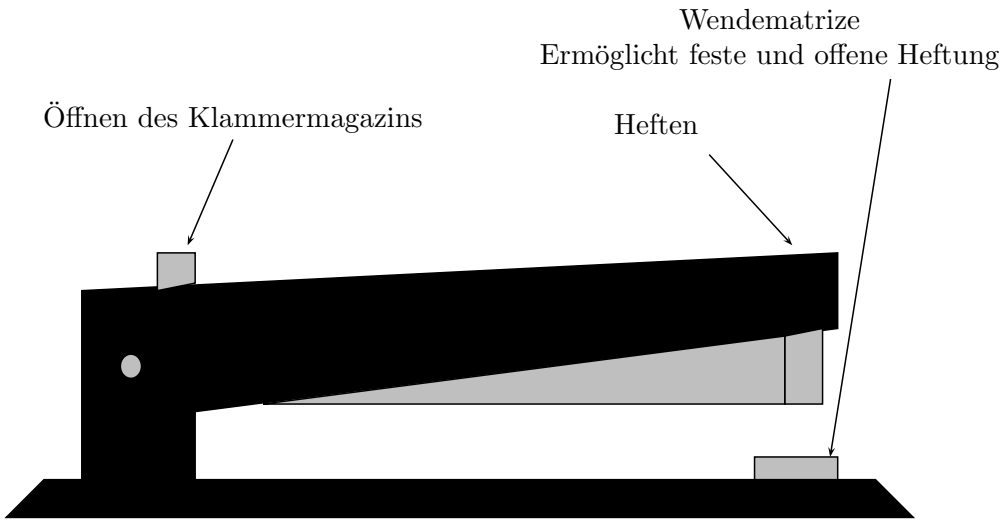
Chapter 3

Das objektorientierte Konzept von Java

3.1 Einführung: Objekte der realen Welt

3.1.1 Fachbegriffe

Um die Frage zu klären, was Objekte nun genau sind, machen wir einen kurzen Ausflug von der Computer- in die „richtige“ Welt. Ein Beispiel für ein Objekt, das sich gut als Anschauung für Objekte in Java eignet, ist ein Klammergerät, wie es in der folgenden Abbildung zu sehen ist:



Ein solches Klammergerät ist ein **Objekt**. Jedes Klammergerät hat einen bestimmten **Zustand**. Bei unserem Klammergerät könnten wir den Zustand folgendermaßen beschreiben:

- Klammermagazin offen oder geschlossen.

- Anzahl der Klammern im Magazin.
- Wendematrize eingestellt für offene oder geschlossene Heftung.

Andere Zustände (zerkratzt, verbeult, kaputt) vernachlässigen wir einmal. Ferner kann man mit einem Klammergerät bestimmte Aktionen durchführen. In der objektorientierten Sprache heißt das, ein Objekt hat bestimmte **Methoden**. Generell unterscheidet man zwischen *Abfragemethoden*, die den Zustand der Objekts nicht ändern und *Modifizierern*, die den Zustand des Objekts ändern. Modifizierer sind in unserem Beispiel:

- Klammern. Der Zustand wird geändert, da eine Klammer aus dem Magazin entfernt wird. Wenn noch Klammern vorhanden waren, wird außerdem der Zustand des Papierstapels (oder des Daumens) verändert.
- Öffnen des Klammermagazins.
- Schließen des Klammermagazins.
- Hinzufügen oder Entfernen von Klammern aus dem Magazin (geht nur bei geöffnetem Magazin).
- Drehen der Wendematrize.

Abfragemethoden sind

- Feststellen des Zustands der Wendematrize.
- Feststellen des Zustands des Klammermagazins (offen/geschlossen).
- Ermitteln der Anzahl der Klammern im Magazin. Wichtig ist, dass am Ende der ursprüngliche Zustand wiederhergestellt wird, d.h. wenn vorher das Klammermagazin geschlossen war, muss es am Ende auch wieder geschlossen werden.

Objekte und Klassen

Jedes Objekt ist ein Objekt einer bestimmten **Klasse**. In unserem Beispiel heißt die Klasse *Klammergerät*. Ein Objekt der Klasse *Klammergerät* ist ein bestimmtes Klammergerät. Zwei verschiedene Objekte unterscheiden sich möglicherweise durch ihren Zustand, also beispielsweise durch die Anzahl von Klammern in ihrem Magazin. Aber auch wenn ihr Zustand gleich ist, sind zwei Klammergeräte doch zwei verschiedene Objekte. Man sagt dann, die Objekte sind *Kopien* oder *Klone* voneinander.

3.1.2 Anwender- und Entwicklersicht

Wichtig bei der Objektorientierung ist die Unterscheidung zwischen *Anwender-* und *Entwicklersicht*. In unserem Beispiel haben wir uns bisher nur mit der Anwendersicht beschäftigt. Der Entwickler des Klammergeräts hat aber einen umfassenderen Blickwinkel:

- Er kennt verborgene Details, wie z.B. Materialeigenschaften, mit denen sich der Anwender nicht beschäftigt.
- Ihm steht eine größere Funktionalität zur Verfügung, z.B. setzt er ein Klammergerät zusammen oder nimmt es auseinander.
- Er legt, und das ist eine wichtige Aufgabe, die Methoden fest, die der Anwender ausführen darf. Er entscheidet z.B., ob er dem Anwender die Möglichkeit geben will, das Klammergerät auseinanderzunehmen. Die Liste der Anwender-Möglichkeiten, die der Entwickler festlegt, ist die *Anwenderschnittstelle*. Die gängige Abkürzung in der Software-Entwicklung dafür ist **API** (application programming interface, deutsch: „Schnittstelle zur Anwendungsprogrammierung"). Die Anwenderschnittstelle für das Klammergerät sind in unserem Beispiel die oben aufgelisteten Methoden.

Andere Sichten

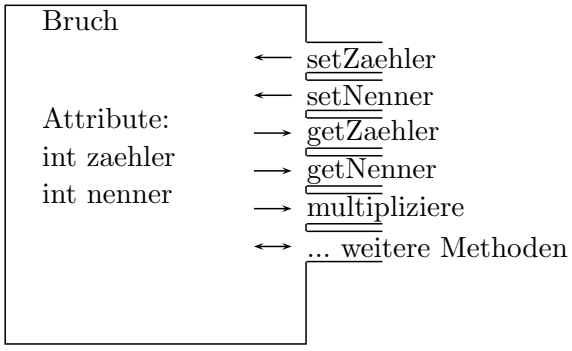
Später werden wir die Rolle des Anwenders noch differenzieren müssen: In den *Projektentwickler*, den *Endanwender* und den *Wiederverwerter*. Diese Differenzierung gibt unser anschauliches Beispiel aber noch nicht her.

3.2 Software-Objekte aus Anwendersicht

3.2.1 Einleitung

Mathematische Brüche als Objekte

In der Software-Entwicklung werden wir natürlich keine Klammergeräte, sondern passendere Objekte verwenden. Wir nehmen als erstes Beispiel eine Klasse namens `Bruch`, die rationale Zahlen repräsentiert. Diese Klasse ist uns vorgegeben und befindet sich in den Vorlesungsunterlagen. Die Attribute dieser Klasse sind der Zähler und der Nenner des Bruchs. Es gibt Methoden zum Abfragen und Setzen der Attribute und für diverse mathematische Operationen.



Wir werden sie in diesem Kapitel komplett betrachten, anwenden und ihre Eigenschaften untersuchen. In einem späteren Kapitel werden wir sie schließlich selbst nachprogrammieren. Bis auf Weiteres sehen wir die Klasse `Bruch` aber rein aus Anwendersicht.

Aufgabenstellung

Zum Testen der Klasse Bruch formulieren wir uns eine erste kleine Aufgabenstellung. Wir wollen ein Programm schreiben, das den Endanwender nach einem Bruch fragt und diesen Bruch gekürzt als gemischten Bruch auf dem Bildschirm ausgibt. Also z.B.:

```
Eingabe: 5/4  ->  Ausgabe:  1+1/4
Eingabe: 8/6  ->  Ausgabe:  1+1/3
Eingabe: 2/2  ->  Ausgabe:  1
Eingabe: 2/8  ->  Ausgabe:  0+1/4
Eingabe: 2/3  ->  Ausgabe:  0+2/3
```

Negative Brüche berücksichtigen wir der Einfachheit halber nicht.

3.2.2 Die Anwenderschnittstelle (API)

Wir wollen zu unserer Berechnung die Klasse `Bruch` benutzen. Sehen wir uns zunächst die entsprechende API an. Sie liegt in der sogenannten *javadoc*-Form vor. Das ist die Form, in der auch die Java-API geschrieben ist. Die vollständige API ist auf der Kurs-Webseite zu finden.
Am Anfang der Javadoc befindet sich eine kurze Beschreibung der Klasse. Sie lautet:
Repräsentiert einen Bruch mit Integer-Werten als Zähler und Nenner. Invariante: Der Bruch ist immer gekürzt. Der Nenner ist immer größer als 0.
Eine Invariante ist eine Eigenschaft, die aus Anwendersicht immer erhalten bleibt. Unsere Bruch-Objekte werden also immer gekürzt sein. Das ist für unser Programm schon einmal sehr praktisch. Wir entnehmen ferner, dass der Nenner nicht 0 werden darf und dass negative Nenner umgewandelt werden. Der Bruch $\frac{1}{-2}$ wird also zu $\frac{-1}{2}$. Das werden wir hier nicht brauchen, da wir der Einfachheit halber negative Brüche ganz ausgeschlossen haben.

3.2.3 Variablen und Objekte

Konstruktoren

Zunächst sehen wir uns aus der API die sogenannten *Konstruktoren* (englisch Constructor) an. Aus der Tabelle entnehmen wir die folgenden drei Konstruktoren:
`Bruch(int zaehler, int nenner)`
Erzeugt einen Bruch mit dem gegebenen Zähler und Nenner.
`Bruch(String s)`
Erzeugt einen Bruch aus einem String der Form Zähler/Nenner.
`Bruch(Bruch r)`
Erzeugt eine Kopie (Klon) des übergebenen Bruchs (Copy-Konstruktor).
Über die Konstruktoren können wir den Anfangswert des Bruchs festlegen. Wir haben die Auswahl, ob wir Zähler und Nenner als Integer-Wert übergeben wollen oder als String oder ob der neue Bruch eine Kopie eines schon bestehenden sein soll.

Deklaration und Erzeugung von Objekten

Der ganze Vorgang hat zwei Stufen. Zunächst einmal wird hier ohne weitere Begründung der Vorgang der Deklaration und Erzeugung erklärt. Die Begründung folgt später in mehreren Schritten.

- Zunächst wird eine Variable vom Typ `Bruch` *deklariert*:

```
Bruch b;
```

- Dann wird mit dem *new*-Operator und einem der Konstruktoren ein neues Objekt erzeugt (hier ein Bruch vom Wert $\frac{1}{3}$).

```
b = new Bruch(1,3);
```

- Beide Schritte kann man auch in einem zusammenfassen:

```
Bruch b = new Bruch(1,3);
```

Start des Testprogramms

Mit dieser Information können wir beginnen, uns um das Testprogramm zu kümmern. Zuerst fragen wir den Endanwender in einer JOptionPane nach einer Bruchzahl. Danach verwenden wir zweckmäßigerweise den Konstruktor mit dem String, um ein Bruch-Objekt zu erzeugen.

```
public static void main(String[] args) {
  Bruch b;
  String s = JOptionPane.showInputDialog("Bruch eingeben (Zaehler/Nenner)");
  b = new Bruch(s);
}
```

Solange die Benutzereingabe ein korrekter Bruch ist, klappt das.

3.2.4 Exceptions

Geben wir in die JOptionPane statt Zahlen Buchstaben ein, so bricht das Programm ab und auf dem Bildschirm erscheint die Fehlermeldung:

```
Exception in Thread "main" java.lang.NumberFormatException: Format
muss zaehler/nenner sein.
```

Als nächstes versuchen wir, einen Bruch mit dem Nenner 0 zu erzeugen. Auch hier bricht das Programm ab und die Fehlermeldung ist:

```
Exception in Thread "main" java.lang.ArithmeticException: Nenner
gleich 0.
```

Es wäre sicher sinnvoll, die Eingabe wiederholen zu lassen. Um zu verstehen, wie man das erreicht und was das Wort *Exception* bedeutet, müssen wir etwas ausholen:

Verbotene Operationen

Wir haben schon die sogenannte *Invarianten* angesprochen: Eigenschaften des Objekts, die aus Anwendersicht immer erhalten bleiben. Versucht der Anwender, eine Invariante zu verletzen, also hier den Nenner explizit auf Null zu setzen, darf das Objekt dem nicht Folge leisten.
Die Frage ist allerdings, was das Objekt statt dessen machen soll. Soll es den Nenner so lassen, wie er ist? Eine Fehlermeldung auf dem Bildschirm ausgeben? Soll es den Nenner auf den kleinstmöglichen Wert setzen?
Egal, was das Bruch-Objekt macht, es wird das gesamte Programm in einen Zustand versetzen, der so nicht vorgesehen war. Das Objekt könnte zwar eine Fehlervariable setzen, aber dann wäre es Aufgabe des Anwenders, zu überprüfen, ob das Objekt noch in einem korrekten Zustand ist.
Wenn der Umgang mit dem Objekt aber *sicher* sein soll, dann muss das Objekt dem Anwender(-programm) sagen, dass ein Fehler aufgetreten ist, und zwar so, dass der Anwender es zur Kenntnis nehmen *muss*. Das tut es zur Zeit auch, in dem es das Programm einfach abbricht. Aber natürlich darf ein Programm auch nicht bei jedem Fehler sofort abbrechen.

Verfeinerung der Rollenperspektiven

Jetzt ist es an der Zeit für eine Verfeinerung der Perspektive des Anwenders:

- Den *Entwickler* der Klasse `Bruch` nennen wir zur besseren Unterscheidung jetzt *Modulentwickler*.
- Wir entwickeln zur Zeit ein komplettes Programm, das die Klasse `Bruch` benutzt. Wir sind sozusagen *Anwender* der Klasse `Bruch`. Gleichzeitig sind wir aber auch der Entwickler eines fertigen Programms (es gibt daher auch eine main-Funktion). Unsere Rolle nenner wir daher *Projekt-Entwickler*. Damit ist gemeint, dass wir Module anwenden und zu einem Projekt zusammenfassen.
- Unser Programm wird am Ende von einem *Endanwender* bedient. Der Endanwender ist kein Programmierer, sondern benutzt lediglich die Bedienoberfläche.
- Der *Wiederverwender* ist ein Anwender, der Objekte für einen anderen Zweck benutzen will, als den, für das sie eigentlich gedacht sind. Dazu muss er die Klasse erweitern oder modifizieren. Dazu ist das Prinzip der Objektorientierung besonders geeignet. Wir werden in einem späteren Kapitel darauf zurückkommen.

Wir sitzen also zwischen Modul-Entwickler und Endanwender. Führen wir uns folgende Überlegungen vor Augen:

- Wir haben im Bruch-Objekt eine verbotene Operation ausgelöst. Der Modul-Entwickler kann nicht wissen, wie wir darauf reagieren wollen und meldet uns den Fehler.

- Der Endanwender (der vielleicht eine falsche Eingabe gemacht hat) will eine sinnvolle Fehlermeldung sehen und keinesfalls einen Programmabsturz mit einer Java-Fehlermeldung.

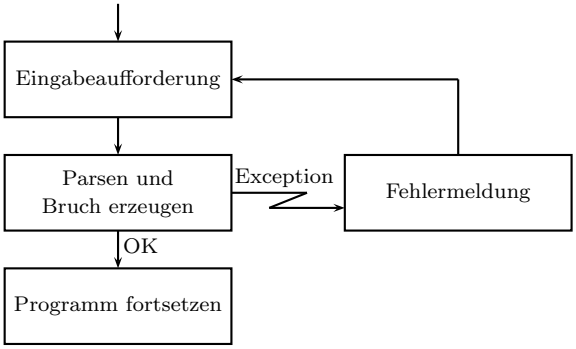
Das heißt zusammengefasst: Sie als Modul-Anwender müssen auf die Fehlermeldung des Objekts reagieren und dem Endanwender eine sinnvolle Fehlermeldung präsentieren.

Fehlerbehandlung

Das Bruch-Objekt löst genau genommen eine sogenannte *Exception* (deutsch: Ausnahme) aus. Umgangssprachlich sagt man: Die Klasse „wirft eine Exception". Es gibt verschiedene Arten von Exceptions. Wir müssen hier mit *ArithmeticExceptions* und *NumberFormatExceptions* rechnen.[1] Eine ausgelöste Exception muss von uns als Modul-Anwender *gefangen* werden, ansonsten kommt es zum Programmabbruch. Die Lösung sieht wie folgt aus:

```
try {
  b = new Bruch(s);
} catch (ArithmeticException e) {
  //Fehlerbehandlung, wenn der Nenner 0 ist
} catch (NumberFormatException e) {
  //Fehlerbehandlung, wenn sich s nicht parsen laesst
}
```

Wir finden hier zwei neue Schlüsselworte *try* und *catch*. Wenn im ersten Block, dem sogenannten *try-Block* eine Exception ausgelöst wird, dann wird gesucht, ob für diese Exception ein zugehöriger *catch-Block* existiert. Wenn ja, dann werden die Programmzeilen im catch-Block ausgeführt und das Programm hinter dem letzten catch-Block fortgesetzt.
Wir wollen dem Endanwender nach einer Fehleingabe eine Fehlermeldung ausgeben und ihn die Eingabe wiederholen lassen. Dazu gibt es einen Trick, der sehr gut die Funktionsweise des try-catch-Blocks zeigt. Das Flussdiagramm und der zugehörige Java-Code ist wie folgt:




```
//break bei korrekter Benutzereingabe
```

[1] Diese Information steht in dem Teil der javadoc, der im Skript nicht abgebildet ist

```
while(true) {
  try {
    String s = JOptionPane.showInputDialog("Bruch eingeben (Zaehler/Nenner)");
    b = new Bruch(s);
    //Bis hierhin kommt das Programm nur, wenn keine Exception auftrat
    break;  //while(true) wird verlassen
  } catch(NumberFormatException e) {
    System.out.println("Eingabeformat stimmt nicht.");
  } catch(ArithmeticException e) {
    System.out.println("Nenner darf nicht Null sein.");
  }
  //Nach einer Exception wird hier fortgesetzt und
  //der naechste Schleifendurchlauf begonnen
}
```

Die Einleseroutine steht in einer Endlosschleife. Der Trick ist, dass nach einem catch-Block das Programm hinter den catch-Blöcken fortgesetzt wird, was bedeutet, dass ein neuer Schleifendurchlauf beginnt. Der `break` -Befehl wird nur erreicht, falls beim Erzeugen des Bruch-Objekts keine Exception aufgetreten ist.
Die genaue Notation des try- und des catch-Blocks nehmen wir zunächst einfach hin. Wir werden sie später noch genauer untersuchen.

3.2.5 Methoden

Methoden aufrufen

Wir haben also jetzt unser Bruch-Objekt erzeugt und können es unter dem Variablennamen `b` ansprechen. Nun wollen wir die gemischten Brüche berechnen. Negative Brüche lassen wir der Einfachheit halber außen vor. Zunächst bestimmen wir den ganzzahligen Anteil des Bruchs. Er berechnet sich aus der Integer-Division von Zähler und Nenner. Diese beiden Werte erhalten wir, indem wir vom Bruch `b` die Methoden `getZaehler()` und `getNenner()` aufrufen:

```
int ganzzahl = b.getZaehler() / b.getNenner();
```

Das Format ist also Objektname – Punkt – Methodenname.
Der Zähler des Restbruchs (also des Bruchs abzüglich des ganzzahligen Anteils) berechnet sich zu *Zähler mod Nenner*. Wir setzen ihn mit der Methode `setZaehler(int zaehler)`. Dabei ist der Wert in der Klammer der sogenannte *Übergabeparameter*.

```
int restZaehler = b.getZaehler() % b.getNenner();
b.setZaehler(restZaehler);
```

Wir haben den Vorteil, dass der Bruch bereits automatisch von der Bruch-Klasse gekürzt wird und wir das nichts mehr selbst machen müssen. Wir müssen nur noch den ganzzahligen Anteil und den Rest auf dem Bildschirm ausgeben:

```
System.out.print(ganzzahl);
if (restZaehler>0) {
  System.out.print("+");
  System.out.println(b);
} else {
  System.out.println();
}
```

Die Zeile `System.out.println(b);` wird intern automatisch in

```
System.out.println(b.toString());
```

umgewandelt. Es wird also der String ausgegeben, den man von der Methode `toString` zurückerhält.
Zusammenhängend ergibt sich also für die Verarbeitung:

```
int ganzzahl = b.getZaehler() / b.getNenner();
int restZaehler = b.getZaehler() % b.getNenner();
b.setZaehler(restZaehler);
System.out.print(ganzzahl);
if (restZaehler>0) {
  System.out.print("+");
  System.out.println(b);
} else {
  System.out.println();
}
```

Beispiel 2

Wir stellen uns eine weitere Aufgabe. Wir wollen ein Programm schreiben, das vom Benutzer die Eingabe einer natürlichen Zahl n verlangt und daraufhin die Summe

$$s_n = \sum_{i=1}^{n} \frac{1}{i}$$

als Bruch auf dem Bildschirm ausgibt. Hier sparen wir uns der Einfachheit halber die Benutzereingabe und nehmen an, `n` sei schon besetzt. Zunächst erzeugen wir uns eine Bruch-Variable für die Summe und eine für das Reihenglied.

```
Bruch summe = new Bruch(0,1);
Bruch element = new Bruch(1,1);
```

Nun setzen wir in einer Schleife den Nenner des nächsten Elements und addieren es zur Summe. Wieder haben wir den Vorteil, dass die Brüche schon automatisch gekürzt werden.

```
for (int i=1; i<=n; i++) {
  element.setNenner(i);
  summe.add(element);
}
```

Am Ende wird das Ergebnis (als Bruch und als `double`-Wert) ausgegeben:

```
System.out.println(summe);
System.out.println(summe.getDoubleWert());
```

Es fehlen uns allerdings noch einige wichtige Details zur Benutzung der restlichen Methoden der Bruch-Klasse. Diese werden wir im nächsten Kapitel betrachten.

Polymorphie

In der API finden wir zwei Versionen der Methode `mult`:

`Bruch`	`mult(Bruch r)` Multipliziert den Bruch mit dem übergebenen Bruch.
`Bruch`	`mult(int faktor)` Multipliziert den Bruch mit dem angegebenen Faktor.

Generell kann es mehrere Methoden mit gleichem Namen aber unterschiedlichen Übergabeparametern geben, d.h. mit Übergabeparametern unterschiedlicher Anzahl oder unterschiedlichen Typs. Je nachdem, welche Parameter man beim Aufruf übergibt, wird automatisch die passende Methode herausgesucht.
Man sagt, die Methode ist *überladen*. Dies ist ein Beispiel, dass ein Methode mit gleichem Namen mehrfach implementiert ist. Allgemein nennt man diese Eigenschaft der Objektoriertierung *Polymorphie*, d.h. Vielgestaltigkeit.

Unveränderliche Objekte

Es gibt ein weiteres Paar von Befehlen, nämlich

`Bruch`	`getInverse()` Gibt die Inverse des Bruchs zurück.
`void`	`inverse()` Invertiert den Bruch.

Beide Befehle berechnen die Inverse eines Bruchs. Der Unterschied wird im folgenden Codestück deutlich.

```
//Version 1
Bruch b = new Bruch(1,2);
b.inverse();  //invertiert b. b hat jetzt den Wert 2/1.

//Version 2
Bruch x = new Bruch(1,2);
Bruch y = x.getInverse(); //x hat immer noch den Wert 1/2.
//y hat den Wert 2/1.
```

Version 1 (`inverse()`) invertiert das Objekt selbst. Version 2 (`getInverse()`) lässt das Objekt unverändert und erzeugt ein zweites Objekt, das den Wert der Inversen hat.
Dazu ist zu bemerken:

- Version 1 ist schneller, da kein neues Objekt erzeugt werden muss.

- Der Umgang mit einer Klasse wird vereinfacht, wenn entweder durchgehend Version 1 oder Version 2 benutzt wird.[2] Wenn eine Klasse *ausschließlich* Methoden der Version 2 hat, dann ist der Inhalt der Objekte *unveränderlich*, kann also einmal im Konstruktor gesetzt und danach nie wieder verändert werden. Wie wir noch sehen werden, hat die Klasse `String` diese Eigenschaft und ebenso viele andere Klassen der Java-Bibliothek.[3]

- Version 2 ermöglicht das sogenannte *daisy chaining*[4]. Damit ist gemeint, dass man mehrere Operationen verketten kann, wie in dem folgenden Beispiel gezeigt wird:

```
String a = "new Bruch(2,3);
Bruch b = a.getInverse().getInverse();  //Daisy chaining
//Damit ist jetzt b=1/(1/a), also b=a.
//Kann z.B. in String sehr sinnvoll eingesetzt werden.
```

3.2.6 Methoden und Funktionen

Methoden- und Funktionsaufrufe unterscheiden sich nur leicht. Funktionen können mit der Syntax „Klassenname.Methodenname“ aufgerufen werden, also z.B.

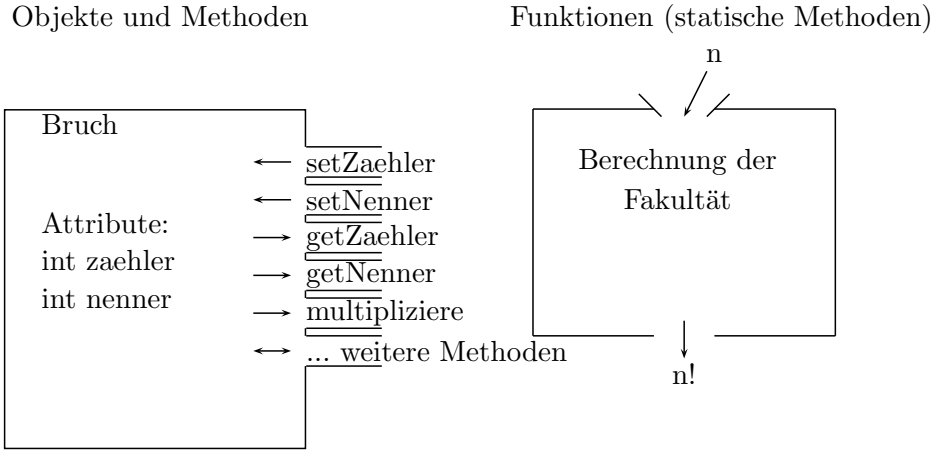
```
Math.sqrt(10);
```

Methoden gehören dagegen immer zu einem Objekt. Der Aufruf ist daher *Objektname.Methodenname*. Ein Beispiel ist:

```
Bruch b = new Bruch(2,4);
b.setNenner(5);
```

Vom Konzept her sind Funktionen und Methoden aber doch sehr unterschiedlich, wie in der folgenden Grafik zu sehen ist. Eine Methode bietet Zugriff auf die Attribute eines Objekts. Eine Funktion berechnet aus einer Menge von Eingabeparametern ein Ergebnis.

[2] Die vorliegende Klasse `Bruch` hat nur zu Lehrzwecken Methoden beider Versionen.
[3] Ein spätere Übungsaufgabe wird sein, auch die Klasse `Bruch` unveränderlich zu machen.
[4] *daisy chain* heißt auf deutsch *Gänseblümchenkette*.



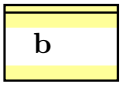
3.2.7 Aliasing

Klebezettel - ein anschauliches Modell

Hier eine erste Erläuterung, was bei der Deklaration und Erzeugung genau passiert. Die folgende Erläuterung hat den Vorteil, sehr anschaulich zu sein. Später werden wir noch eine exakte (aber nicht so anschauliche) Erläuterung sehen.
Bisher haben wir Variablen als eine Art Behälter gesehen, der einen Namen einen Typ und natürlich auch einen Inhalt hat. Diese Veranschaulichung passt für primitive Datentypen gut; bei Objekten führt sie aber in die Irre. Wir brauchen statt dessen eine neue Veranschaulichung:
Die Deklaration

```
Bruch b;
```

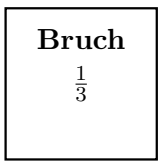
stellen wir uns so vor, dass wir einen *Klebezettel* erzeugen, der mit dem Namen `b` beschriftet ist.



Die Anweisung

```
new Bruch(1,3)
```

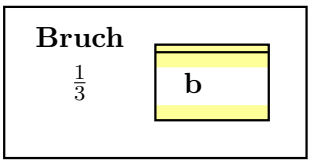
erzeugt ein Bruch-Objekt mit dem Wert $\frac{1}{3}$.



Mit der Zuweisung zur Variablen `b`

```
b = new Bruch(1,3);
```

kleben wir den Klebezettel auf das Objekt. Damit ist das Bruch-Objekt unter dem Namen `b` ansprechbar.

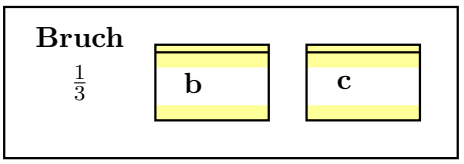


Aliasing

Wir haben uns also einen Bruch `b` erzeugt. Wichtig ist, zu verstehen, was in der Zeile

```
Bruch c = b;
```

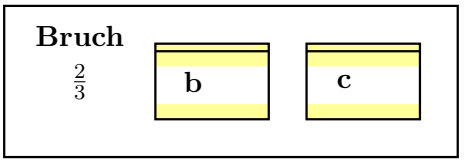
passiert. Diese Zeile ist eine direkte Zuweisung. Hier passiert folgendes: Wir erzeugen uns einen neuen Klebezettel mit der Aufschrift `c` und kleben ihn an unser Bruch-Objekt.



Die nächste Zeile ist:

```
b.setZaehler(2);
```

Wir haben nun den Wert unseres Objekts verändert. Die Situation ist nun:



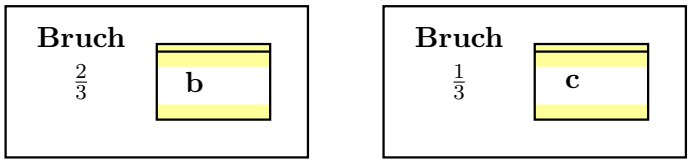
Damit sollte klar sein, dass beide Zeilen

```
System.out.println(b);
System.out.println(c);
```

den Wert 2/3 zurückliefern. Das Objekt hat zwei *Alias-Namen*, unter denen es ansprechbar ist. Davon leitet sich auch der Begriff *Aliasing* ab, der den ganzen Vorgang beschreibt. Will man Aliasing vermeiden, muss man ein zweites Objekt erzeugen und das erste dort hinein kopieren. Gewöhnlich gibt es dafür einen speziellen Konstruktor, den sogenannten *Copy-Konstruktor*. Die Zeilen

```
Bruch b = new Bruch(1,3);
Bruch c = new Bruch(b);
b.setZaehler(2);
```

ergeben im Unterschied zum ersten Beispiel die folgende Situation:



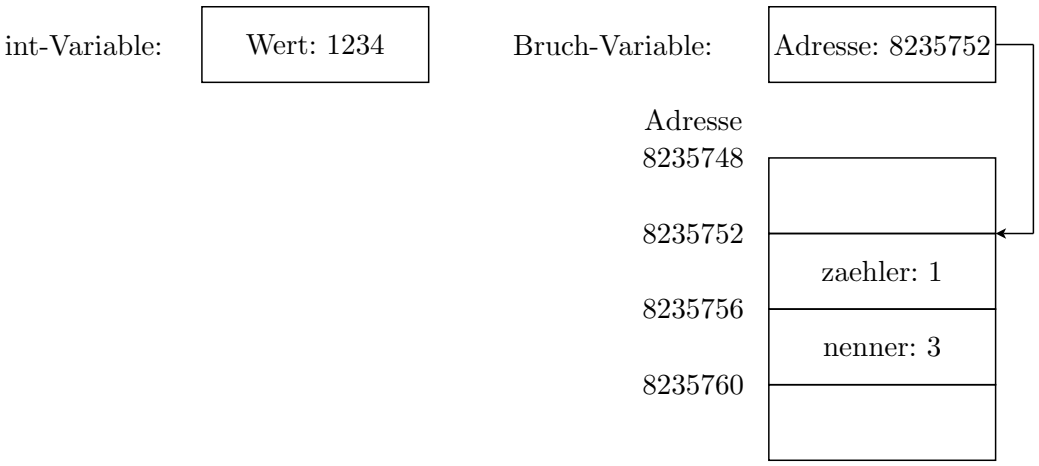
Damit wirkt sich eine Änderung von `b` nicht mehr auf `c` aus.

3.3 Interne Darstellung

Letztlich enthielt das letzte Kapitel nur eine leicht zu begreifende Anschauung, wie man in Java mit Objekten umgeht. Für uns ist es aber auch wichtig zu sehen, was in Java *wirklich* passiert.

3.3.1 Interne Darstellung

Es gibt einen großen Unterschied zwischen Objekten und primitiven Datentypen. Dazu sehen wir uns an, wie Variablen vom Typ *Bruch* im Vergleich zu Variablen vom Typ *int* angelegt werden. Eine *int*-Variable erhält 4 Bytes an Speicherplatz, in denen der *int*-Zahlenwert steht. Eine *Bruch*-Variable erhält ebenfalls 4 Byte an Speicherplatz. In diesen 4 Byte steht eine *Speicheradresse*. Das bedeutet folgendes: Der gesamte Speicher des Rechners ist von 0 aufwärts durchnummeriert. Eine Speicheradresse bezeichnet also ein ganz bestimmtes Byte des Speichers. In unserem Fall ist die Speicheradresse der Beginn des Bereichs, an dem das Objekt *tatsächlich* steht. Bei Objekten enthalten die Variablen also keinen Wert, sondern einen Verweis (Referenz, Zeiger, Pointer) auf das Objekt.



Auch hier gibt es wieder spezielle Ausdrücke:

- Variablen können entweder ein *Wert (value)* oder eine *Referenz (reference)* sein.
- In Java sind primitive Datentypen Werte; Objekte und Felder sind Referenzen.

- Da man die Speicheradressen in Java nicht auslesen und nicht mit ihnen rechnen kann, heißen sie *implizite Referenzen*. Explizite Referenzen kennt Java nicht. Eine typische Programmiersprache für explizite Referenzen ist C.
- Referenzen werden auch *Zeiger* oder englisch *Pointer* genannt. [5]

3.3.2 Lebensdauer eines Objekts

Das `null`-Objekt

Sehen wir uns den Vorgang der Erzeugung eines Objekts genau an (für Felder, die auch Objekte sind, gilt das gleiche).

- Nach der Deklaration:

```
Bruch t;
```

hat die Variable t den Wert `null`. `null` ist ein Java-Schlüsselwort für den speziellen Wert „kein Objekt“. „null“ bedeutet englisch „ungültig“ und wird klar von der Zahl Null unterschieden. Im Deutschen ist das gerade in der gesprochenen Sprache natürlich viel schwieriger. Hier spricht man zur Unterscheidung von der Zahl 0 oft vom „Null-Pointer“ oder spricht `null` als „nall“ aus.
Intern ist `null` eine Speicheradresse, von der Java weiß: Dort *kann kein Objekt stehen*. In Java gibt es keine Möglichkeit, den genauen Wert von `null` zu erfahren. Er kann je nach Implementation unterscheidlich sein, ist oft aber tatsächlich die Zahl 0.
Man kann auch explizit

```
Bruch t = null;
```

schreiben. Wenn man versucht, die Werte eines Objekts `null` auszulesen, erhält man eine Fehlermeldung *NullPointerException.*

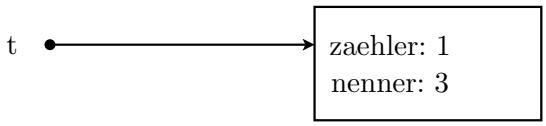
```
public static void main(String[] args) {
  Bruch t = null;
  t.zaehler = 10;  //NullPointerException
} //main
```

Interne Vorgänge beim Erzeugen mit `new()`

```
t = new Bruch(1,3);
```

erzeugt ein neues Objekt im Speicher und weist der Variablen `t` die Anfangsadresse dieses Objekts zu. In diesem Konstruktor werden anschließend auch Zähler und Nenner belegt.

[5] Manchmal wird „Zeiger“ auch im Sinne von „explizite Referenz“ benutzt.

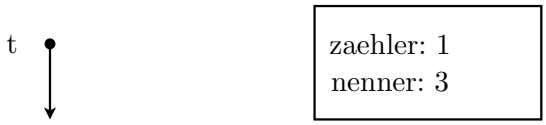


Garbage Collection

Durch

```
t = new Bruch(1,3);
t = null;
```

sieht die Lage so aus:



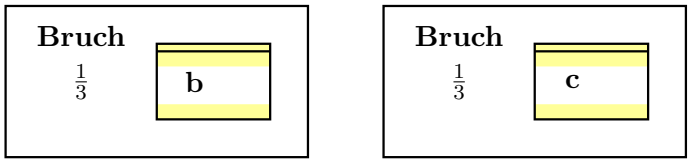
Das eigentliche Objekt ist vom Programm nicht mehr erreichbar und aus Sicht des Programms Datenmüll. Java startet automatisch in regelmäßigen Abständen ein kleines Programm *Garbage Collector*, dessen Aufgabe es ist, solchen Müll im Speicher zu löschen, damit der Speicher wieder für andere Zwecke genutzt werden kann.
In der Klebezettel-Darstellung würde `t=null` bedeuten, dass man den Klebezettel `t` vom Objekt abreißt und anschließend einen Klebezettel hat, der auf keinem Objekt klebt und ein Objekt, auf dem kein Klebezettel klebt. Letzteres wird vom Garbage-Collector gelöscht.

3.3.3 Wann sind Objekte gleich?

Wann primitive Datentypen gleich sind, ist einfach zu beantworten. Für primitive Datentypen (z.B. `int`) gilt, dass sie genau dann gleich sind, wenn ihre Werte gleich sind. Bei Objekten ist das etwas komplizierter. Nehmen wir die folgende Situation:

```
Bruch b = new Bruch(1,3);
Bruch c = new Bruch(b);
```

Das ergibt in der Klebezettel-Darstellung:



`b` und `c` sind also Klone voneinander. Jetzt kann man Gleichheit auf unterschiedliche Arten definieren:

1. Zwei Variablen sind gleich, wenn sie Aliase für das gleiche Objekt sind. In diesem Fall wären `b` und `c` *nicht* gleich.

2. Zwei Variablen sind gleich, wenn sie Aliase oder Klone sind. In diesem Fall wären `b` und `c` gleich.

Beide Fälle können in Java überprüft werden:

```
if (b==c) {
  System.out.println("b und c sind Aliase");
}
if (b.equals(c)) {  //gibt NullPointerException bei b==null
  System.out.println("b und c sind Aliase oder Klone");
}
if (Objects.equals(b, c)) {  //Funktioniert auch bei b==null
  System.out.println("b und c sind Aliase oder Klone");
}
```

Dazu gibt es verschiedene Anmerkungen:

- Eine typische Java-Anfänger-Falle ist es `==` zu benutzen, wenn man `equals` benutzen will. Es ist also wichtig, dass sie sich den Unterschied klarmachen.
- Im zweiten Fall kann man natürlich auch `c.equals(b)` benutzen.
- Wie `equals` funktioniert, hängt davon ab, wie der Modulentwickler die Methode programmiert hat. Es ist eine Konvention, sie so wie beschrieben zu programmieren. Java hindert einen nicht daran, die Methode anders (oder fehlerhaft) zu implementieren. Lesen sie in Zweifelsfall lieber noch einmal in der API nach.

3.4 Felder (Arrays)

Eine Liste von Variablen gleichen Typs kann in Java in einer *Feld-Variablen* untergebracht werden. Eine Feldvariable hat neben ihrem Namen und ihrem Typ auch eine bestimmte Länge, die angibt, wie viele Elemente in ihr untergebracht werden können. Jedes Element hat einen Index, unter dem es angesprochen werden kann. Der Index des ersten Elements ist immer 0. Die folgende Abbildung zeigt als Beispiel ein Feld mit dem Namen `count` und mit 5 Elementen, die Werte zwischen 3 und 8 besitzen.

Name: count Typ der Elemente: int Länge: 5					
Index:	0	1	2	3	4
Wert:	8	3	3	7	5

3.4.1 Grundfunktionen

Felder sind Klassen und haben viele Eigenschaften, die wir aus dem letzten Kapitel kennen. Allerdings gibt es auch ein paar Besonderheiten, die Felder von anderen Klassen unterscheiden.

Erzeugen von Feldern

Wie Klassen werden sie zunächst **deklariert** und dann **erzeugt**. Der Code sieht ein wenig anders aus als gewohnt, da es keinen „echten“ Konstruktor gibt.

```
int[] count;          //Deklarieren
count = new int[10];  //Erzeugen und Festlegen der Groesse

int[] count = new int[10];  //Beides zusammengefasst
```

Literale für Felder

Bei der Deklaration einer Feld-Variablen kann man diese auch direkt vorbelegen. Dazu nutzt man ein Feld-Literal, das wie folgt aussieht:

```
int[] x = {1,2,3,5,8};
```

Dieses Literal ist allerdings nicht so universell einsetzbar, wie andere Literale. Es ist nicht möglich, einem bereits existierenden Feld per Literal Werte zuzuweisen.

```
x = {1,2,6,3};    //Compilerfehler
```

Mit einem Trick geht es dann allerdings doch:

```
x = new int[] {1,2,6,3};
```

Ansprechen von Elementen

Elemente werden wie in den folgenden Beispielen angesprochen:

```
count[0] = 7;
count[1] = count[0] * 2;
count[2]++;
count[3] -= 60;
```

Wird ein Feld mit einem fehlerhaften Index angesprochen, bricht das Programm mit einer Fehlermeldung ab. Genauer gesagt wird eine `ArrayIndexOutOfBoundsException` geworfen, die in einem späteren Kapitel behandelt wird. Ein Feld erhält, anders als in anderen Sprachen, automatisch einen voreingestellten Wert. Man kann also Felder gleich nach der Zuweisung des Speichers auslesen (führt in C oder Pascal zu undefinierten Werten).

Typ	Voreingestellter Wert
Zahlen (int, float, ...)	0
char	ASCII-Zeichen 0
boolean	false
Objekte	null

```
int a[] = new int[2];
System.out.println(a[0]);   //-> 0
```

Beim Zugriff auf Feldelemente kann man auch eine Integer-Variable als Index benutzen. Das ist der große Vorteil von Feldern.

```
int[] a = new int[2];
int i = 1;
a[i] = 3;
```

Besonders praktisch ist das in Kombination mit Zählschleifen. Dazu brauchen wir jedoch noch eine weitere Eigenschaft von Feldern: Es ist erlaubt, die Länge eines Feldes mit `length` auszulesen.[6] Es sind immer die Elemente 0 bis *length-1* vorhanden.

```
int[] f = new int[30];
System.out.println(f.length);    //-> 30; Elemente 0..29
```

Damit können wir ein Feld mit 100 Elementen erzeugen, deren Inhalt der Index zum Quadrat ist:

```
int[] q = new int[100];
for (int i=0; i<q.length; i++) {
  q[i] = i*i;
}
```

3.4.2 Felder kopieren

Es gibt einige Möglichkeiten, ein Feld in ein anderes Feld zu kopieren. Hier erst einmal eine kurze Übersicht. Dabei soll jeweils ein Feld von `src` nach `dst` kopiert werden.

- Kopieren in einer for-Schleife.
- `System.arraycopy(src, srcpos, dst, dstpos, length);`
- `dst = Arrays.copyOf(src, src.length);`
- `dst = Arrays.copyOfRange(src, 0, src.length);`
- `dst = src.clone();`

Am einfachsten zu merken ist sicherlich der clone-Befehl:

```
int[] x = new int[]{1,2,3};
int[] y = x.clone();
```

Am flexibelsten ist der Befehl `System.arraycopy`:

```
System.arraycopy(src, src_pos, dst, dst_pos, length);
```

[6] Ohne Klammern. Es handelt sich nicht um eine Methode, sondern um das Auslesen einer internen Variablen.

Die Parameter heißen:

Parameter	Bedeutung
`src`	Ursprung (source).
`srcpos`	Index des ersten zu kopierendes Elements in src.
`dst`	Ziel (destination).
`dstpos`	Dorthin (ab diesen Index) soll das Feld kopiert werden.
`length`	Anzahl der zu kopierenden Elemente.

```
int[] b = new int[a.length];
System.arraycopy(a, 0, b, 0, a.length);
```

3.4.3 Die Aufzählungsschleife

Sehr häufig muss ein Feld elementweise durchlaufen werden, z.B. um es auf dem Bildschirm auszugeben. Die Standard-Schleife hierzu ist (am Beispiel des Integer-Feldes `dat`):

```
for (int i=0; i<dat.length; i++) {
  System.out.println(dat[i]);
}
```

Alternativ dazu kann man auch die sogenannte Aufzählungs- oder *foreach*-Schleife verwenden. Der entsprechende Code ist:

```
for (int z: dat) {
  System.out.println(z);
}
```

Gesprochen: „Für jedes `z` im Feld `dat`“. Die Variable `z` nimmt also der Reihe nach alle Werte an, die im Feld `dat` stehen. Aus dem Satzteil „für jedes“ leitet sich auch der Name *foreach*-Schleife ab. Java benutzt allerdings statt *foreach* das „normale“ Schlüsselwort `for`.

3.4.4 Einfache Ausgabe des Feldinhalts

Ab Java 6 kann man den Inhalt eines Feldes einfach ausgeben mit

```
System.out.println(Arrays.toString(feld));
```

Bei mehrdimensionalen Feldern (siehe nächstes Kapitel) verwendet man statt dessen:

```
System.out.println(Arrays.deepToString(feld));
```

3.4.5 Mehrdimensionale Felder

Felder können auch mehrdimensional sein. Im folgenden Beispiel wird ein zweidimensionales Feld mit 3 Zeilen und 5 Spalten erzeugt.

```
double[][] feld = new double[3][5];
feld[0][0] = 5;  //Element in der linken oberen Ecke
feld[2][4] = 2;  //Element in der rechten unteren Ecke
```

Solche Felder werden gewöhnlich in 2 geschachtelten for-Schleifen beschrieben und ausgelesen. Nehmen wir als Feld

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

Die Anzahl der Zeilen erfragt man mit

```
int z = diff.length;
```

Die Länge einer Zeile (im Beispiel der Zeile 0) erfragt man mit

```
int s = diff[0].length;
```

Im folgenden Beispiel wird ein Feld erzeugt, in dessen Elementen jeweils die Differenz zwischen Zeilen- und Spaltennummer steht:

```
int[][] diff = new int[10][10];
for (int i=0; i<diff.length; i++) {
  for (int j=0; j<diff[0].length; j++)
    diff[i][j] = i-j;
  }
}
```

An der Art, wie die Länge einer Zeile bestimmt wird, kann man erkennen: Das Feld muss nicht „rechteckig“ sein. So etwas wird man allerdings nur selten finden. Hier ein kleines Beispiel dazu:

```
int[][] dreieck = new int[10][];   //letzter Index wird frei gehalten
for (int i=0; i<dreieck.length; i++) {
  dreieck[i] = new int[i+1];       //Erzeugen der Zeile Nr. i
}
```

Die Länge in Richtung des 1. Index erfragt man mit `dreieck.length`. Die Länge in Richtung des 2. Index erfragt man mit `dreieck[i].length`. Die Länge kann je nach Position i variieren. Man kann auch ein rechteckiges Feld nachträglich zu einem nicht rechteckigen machen:

```
feld[2] = new double[10];
```

Diese Besonderheiten teilt Java mit C# und Python (Lists). In C und Pascal sind mehrdimensionale Felder immer rechteckig. C# hat den Vorteil, dass es beide Varianten gibt. Dort wird unterschieden zwischen *jagged arrays* (gezackte Felder) für Felder wie in Java und *rectangular arrays* (rechteckige Felder) für Felder wie in C.

3.5 Strings

Neben Feldern sind auch Strings Objekte mit einigen Besonderheiten. Die beiden wichtigsten Besonderheit behandeln wir gleich vorweg:

3.5.1 Besonderheiten

Literale

Es gibt String-*Literale*. Das hat auch zur Folge, dass wir ein String-Objekt nicht mit `new` und einem Konstruktor erstellen müssen, sondern einfach das entsprechende Literal verwenden können.

```
String s = "Hallo";
```

Die einzigen anderen Objekte mit Literalen sind Felder.

Unveränderlichkeit

Nachdem ein String-Objekt einmal erstellt wurde, ist es *unveränderlich*. Alle Methoden zur String-Manipulation erzeugen einen *neuen*, veränderten String.

3.5.2 Methoden von Strings

Strings haben zahlreiche vorgegebene Methoden, von denen hier nur die wichtigsten aufgezählt sind:

char	charAt(int index)	Zeichen an Position *index*
boolean	equals(Object anObject)	Sind 2 Strings gleich?
int	indexOf(String str)	Position des ersten Vorkommens von *str*
int	length()	Stringlänge
String[]	split(String regex)	Teilt String in Teilstrings auf. *regex* ist das Trennzeichen, an dem aufgetrennt wird.[7]
String	substring(int beginIndex, int endIndex)	Gibt Teilstring zwischen *beginIndex* (einschließlich) und *endIndex* (ausschließlich) zurück.

Den Gebrauch der Methoden erkennt man am besten am nachfolgenden kleinen Beispiel:

```
String a = "Hallo";
String b = "tschuess";

int l = b.length();          //ergibt 8
char c = a.charAt(0);        //ergibt 'H'
int pos = a.indexOf("ll");   //ergibt 2
String[] x = b.split("u");   //ergibt ["tsch","ess"]
```

Es gibt übrigens keine Möglichkeit, einzelne Zeichen in String direkt zu verändern. Man muss hier den Umweg über die Klasse `StringBuilder` nehmen, die in einem späteren Kapitel erläutert wird.

3.5.3 Escape-Sequenzen

In Strings nimmt das Zeichen \ (Backslash) eine Sonderstellung ein. Es ist ein sogenanntes *Metazeichen* und wird speziell interpretiert. Zum Beispiel ist `\n` ein Zeilenvorschub. Die Zeile

```
System.out.println("abc\ndef");
```

erzeugt die Ausgabe

```
abc
def
```

`\n` heißt dabei *Escape-Sequenz*. Eine Auflistung der wichtigsten Escape-Sequenzen findet sich im Anhang. Will man den Backslash selbst ausgeben, muss man einen doppelten Backslash \\ angeben.

3.5.4 Reguläre Ausdrücke

Die Methode

```
String[] split(String regex)
```

verlangt als Übergabeparameter einen sogenannten *regulären Ausdruck*. Diese Ausdrücke sind Strings, in denen es weitere Metazeichen gibt, die speziell interpretiert werden:

```
[ ] ( ) { } | ? + - * ^ $ \ .
```

Damit ein Metazeichen nicht interpretiert wird, muss man einen Backslash davor setzen. Da ein einfacher Backslash für eine Escape-Sequenz gehalten werden würde, muss das in Java ein doppelter Backslash sein. Will man beispielsweise einen String nach dem Punkt-Zeichen trennen, muss man

```
String[] x = s.split("\\.");
```

schreiben. Die Metazeichen selbst werden in dieser Vorlesung nicht behandelt. Nur ein Hinweis dazu. Das Metazeichen `+` bedeutet: ein oder mehrere Exemplare des vorangegangenen Zeichens. Beispiel:

```
String s = "Unterschiedlicher     Abstand zwischen  Worten .";
String[] x1 = s.split(" ");
String[] x2 = s.split(" +");
System.out.println(Arrays.toString(x1));
System.out.println(Arrays.toString(x2));

//ergibt
//[Unterschiedlicher, , , , , Abstand, zwischen, , Worten, .]
//[Unterschiedlicher, Abstand, zwischen, Worten, .]
```

In der ersten Version wird an *einem* Leerzeichen getrennt. Folgen mehrere Leerzeichen aufeinander, entstehen im Array leere Einträge. In der zweiten Version wird an *ein oder mehreren* Leerzeichen getrennt. Hier gibt es keine leeren Feld-Einträge mehr.

Vergleich von Strings

Wie für andere Objekte gilt auch hier: Der Vergleich mit `==` überprüft, ob es sich um Aliase handelt. Der Vergleich mit `equals` überprüft, ob der Inhalt der Strings gleich ist. Dies ist der Fall, wenn es sich um Klone oder Aliase handelt. Da man gewöhnlich wissen will, ob die *Inhalte* zweier Strings gleich sind, benutzt man normalerweise `equals`.

```
if (s1.equals(s2)) {
  System.out.println("s1 und s2 haben den gleichen Inhalt");
}
```

In der Sun-Implementation liefern aber erstaunlicherweise die folgenden Zeilen den Wert `true`:

```
String s1 = "Hallo";
String s2 = "Hallo";
boolean g = (s1==s2);  //g wird true
```

`s1` und `s2` sind also Aliasse. Der Grund ist, dass beim Anlegen von `s2` Java erkennt, dass es den String `"Hallo"` schon gibt. Daraufhin legt Java kein neues String-Objekt an, sondern erzeugt einfach einen neuen Verweis auf das bestehende Objekt. Da die Objekte unveränderlich sind, ist das kein Problem (wenn sie veränderlich wären, wäre es sehr wohl eines, denn dann würde man, wenn man `s1` verändert, gleichzeitig auch `s2` verändern).

3.6 Entwicklung eines Beispiel-Objekts

3.6.1 Software-Objekte aus Entwicklersicht

In diesem Kapitel wollen wir die Klasse `Bruch`, die wir im letzten Kapitel benutzt haben, selbst nachprogrammieren. Wir wollen darauf achten, dass die Eigenschaften, die wir im letzten Kapitel aus Anwendersicht kennengelernt haben, auch in der selbst programmierten Klasse erhalten bleiben. Wir werden hier nicht die komplette API implementieren, aber genug davon, dass alle wichtigen Prinzipien klar sind.

3.6.2 Zusammengesetzte Datentypen

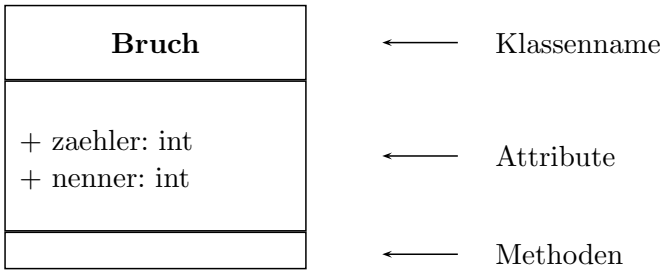
Klassen in ihrer einfachsten Form sind *zusammengesetzte Datentypen*. Sie sind in etwa vergleichbar mit einem *RECORD* in Pascal oder einem *struct* in C. Eine ganz einfache `Bruch`-Klasse in Java ist ein Datentyp, der sich aus zwei Integer-Variablen, je einer für Zähler und Nenner, zusammensetzt. Das Aussehen ist wie folgt:

```
public class Bruch {
  public int zaehler;
  public int nenner;
}
```

Das `public`-Schlüsselwort in der ersten Zeile hat nur mit externen Paketen Bedeutung.[8] Vor den beiden Variablen-Deklaratinen heißt `public`, dass die Methode für den Anwender benutzbar ist und in der API auftaucht. Wenn an dieser Stelle `private` stehen würde, könnte man sie nur in der Klasse selbst verwenden (d.h. nur der Modulentwickler).
Die Daten, die von Klassen zusammengefasst werden, heißen die *Attribute* oder die *Eigenschaften* der Klasse. Im Beispiel sind die Attribute der Klasse Bruch vom Typ int und haben die Namen `zaehler` und `nenner`.

UML-Klassendiagramme

Eine Klasse wird häufig mit Hilfe eines *UML-Diagramms* dargestellt. Das UML-Klassendiagramm für die Klasse `Bruch` sieht folgendermaßen aus:



Das „+“ steht für *public*. Ein „−“ steht für *private*. Da es keine Methoden gibt, ist das untere Kästchen noch leer.

3.6.3 Benutzung der Klasse aus Anwendersicht

Um eine Klasse aus Anwendersicht testen zu können, braucht man eine Testklasse. Diese könnte etwa so aussehen:

```
public class BruchTest {

  public static void main(String[] args) {
    //Mein Testcode
  }

}
```

Die Testklasse `BruchTest` selbst benutzt keine speziellen objektorientierten Eigenschaften. Ihr Aufbau ist so, wie wir es aus den letzten Kapiteln kennen. Formal ist sie aber dennoch eine Klasse. Wird sie von Java aus gestartet, so wird wie gewohnt die `main`-Funktion der Klasse aufgerufen.
Man könnte die `main`-Funktion auch in die Klasse `Bruch` selbst schreiben und diese dann beim Programmstart aufrufen. Das würde prinzipiell funktionieren, aber man hat dann nicht exakt die Anwendersicht, sondern darf noch zusätzlich einige Dinge, die nur aus Entwicklersicht erlaubt sind. Wir werden das später

[8] Wir werden später sehen, dass die Klasse nur dann von externen Paketen aus benutzt werden kann, wenn hier das Schlüsselwort `public` angegeben wurde.

noch genauer sehen. Einstweilen halten wir uns einfach an die Regel, dass die `main`-Funktion für den Test außerhalb der zu testenden Klasse stehen sollte.

Klassen und Dateien

Das heißt in der Regel auch, dass man zum Test einer Klasse eine eigene Datei mit dem Test-Code schreiben muss. Die Regel in Java ist nämlich, dass jede Klasse in eine eigene Datei geschrieben wird. Der Dateiname setzt sich aus dem Klassennamen und der Endung *.java* zusammen. Größere Programme sind so gut wie immer auf mehrere Klassen und damit auch auf mehrere Dateien verteilt. Diese Dateien sollten alle im selben Verzeichnis liegen.

Erzeugung und Benutzung eines Bruch-Objekts

Sehen wir uns dies an einem Beispiel an. Die Klasse `Bruch` sieht folgendermaßen aus:

```
public class Bruch {
  //Attribute
  public int zaehler;
  public int nenner;
}//public class Bruch
```

In der Testklasse benutzten wir das Objekt aus Anwendersicht:

```
public class BruchTest {
  public static void main(String args[]) {
    Bruch r1 = new Bruch();    //Deklaration und Konstruktorausruf

    r1.zaehler = 4;            //Zaehler setzen
    r1.nenner = 5;             //Nenner setzen

    //Zaehler und Nenner auslesen
    System.out.println("Zaehler = "+r1.zaehler);
    System.out.println("Nenner = "+r1.nenner);

  }
}
```

Wird die Klasse `BruchTest` gestartet, stößt Java in der ersten Zeile auf das Objekt der Klasse `Bruch`. Nun sucht Java nach der entsprechenden Datei. Dabei werden die Java-Systemordner und das aktuelle Verzeichnis durchsucht (wie man das ändert, folgt in einem späteren Kapitel). Wird die Datei gefunden, wird sie nachgeladen.
Wir können bereits ein Objekt der Klasse `Bruch` erzeugen. Dazu benutzen wir einen Konstruktor, dem keine Parameter übergeben werden. Anschließend können wir die Bestandteile des zusammengesetzten Datentyps wie dargestellt verändern oder auslesen. Vorbesetzt werden die Attribute wie bei Feldern mit 0.

Damit haben wir schon unser erstes kleines objektorientiertes Programm geschrieben.

3.6.4 Schreiben einer Methode

toString

Um unsere Klasse besser testen zu können, wollen wir als erstes die Möglichkeiten zur Darstellung eines Objekts untersuchen. Wir ändern dazu unsere Testfunktion ab:

```
public static void main(String args[]) {
  Bruch r1 = new Bruch();    //Deklaration und Konstruktorausruf

  r1.zaehler = 4;            //Zaehler setzen
  r1.nenner = 5;             //Nenner setzen

  System.out.println(r1);    //Bruch ausgeben
}
```

Eigentlich wünschen wir uns die Ausgabe `4/5`, wie in unserer Vorlage. Leider ist die tatsächliche Ausgabe aber in der Art, wie

```
Bruch@10b62c9
```

Daran erkennt man den Namen der Klasse und den Speicherplatz der Objekts. Im letzten Kapitel haben wir gelernt, dass genau der String ausgegeben wird, der von der Methode `toString()` zurückgegeben wird. Bisher haben wir nur eine Art Default-Implementierung.[9] Um die Ausgabe zu verbessern, müssen wir eine neue Methode `toString` schreiben. Diese hat das folgende Aussehen:

```
  public String toString() {
    String r = this.zaehler + "/" + this.nenner;
    return r;
  }
```

Die Kopfzeile sieht fast so aus wie die Kopfzeile einer Funktion. Allerdings fehlt das Schlüsselwort `static`. Dieses Schlüsselwort unterscheidet Methoden von Funktionen.
Wie bei Funktionen heißt `public`, dass die Methode außerhalb der Klasse aufgerufen werden kann. Damit ist sie für den Anwender benutzbar und taucht in der API auf. Wenn an dieser Stelle `private` stehen würde, könnte man die Funktion nur in der Klasse selbst verwenden (d.h. nur der Modulentwickler).
In der zweiten Zeile wird der String aus Zähler und Nenner zusammengebaut. Wichtig ist dabei das `this`-Schlüsselwort, dem ein eigener kleiner Abschnitt gewidmet sein soll:

[9] Die Default-Implementierung steht in der Klasse `Object`, von der Bruch automatisch *abgeleitet* ist. Dies werden wir noch im Kapitel über Vererbung ausführlich betrachten.

Einschub: Das this-Objekt

Vergegenwärtigen wir uns noch einmal unsere augenblickliche Rolle als Entwickler der Klasse. Wir entwickeln eine Art *Schablone*, nach der sich der Modulanwender später auf folgende Weise Objekte herstellt und benutzt:

```
Bruch x1 = new Bruch();
Bruch x2 = new Bruch();
String s1 = x1.toString();
String s2 = x2.toString();
```

Das heißt, der Anwender ruft die Methode `toString` *eines bestimmten Objekts* auf und will natürlich die Werte *dieses Objekts* zurückgegeben bekommen. Doch woher wissen wir als Entwickler der Klasse, von welchem Objekt der Anwender `toString` aufgerufen hat? Die Antwort darauf ist das Schlüsselwort `this`. Dieses Schlüsselwort verweist immer auf das Objekt, von dem der Benutzer eine bestimmte Methode aufgerufen hat. Dieses Objekt nennt man aus Entwicklersicht das *this-Objekt*. `this.zaehler` und `this.nenner` greifen damit auf die Attribute genau des Objekts zurück, das der Benutzer angesprochen hat.
In diesem einfachen Fall kann man die beiden `this`-Schlüsselworte sogar weglassen. Sie werden dann von Java automatisch hinzugefügt.
`String r = zaehler + "/" + nenner;`
Es ist aber guter Stil, sie explizit hinzuschreiben.

Fortsetzung der toString-Methode

Die letzte Zeile

```
return r;
```

beendet die Methode und sorgt dafür, dass der String `r` als Ergebnis zurückgegeben wird. Wenn wir den Testcode von oben noch einmal laufen lassen, erhalten wir jetzt das gewünschte Ergebnis `4/5`.

3.6.5 Datenkapselung

Die objektorientierte Programmierung verfährt nach den drei Basisprinzipien

- Kapselung,
- Vererbung
- und Polymorphie (Vielgestaltigkeit).

Ein Beispiel für Polymorphie haben wir schon kennengelernt. In der Klasse `Bruch` gibt es zwei Versionen der Methode `mult`, jeweils mit unterschiedlichen Übergabeparametern. Die Methode `mult` ist somit *polymorph* (vielgestaltig).
Nun begegnet uns das nächste Grundprinzip, nämlich die Kapselung.
Beginnen wir mit der bereits bekannten Klasse für Brüche:

```
public class Bruch {
  //Attribute
  public int zaehler;
  public int nenner;
} //public class Bruch
```

Der Nenner eines Bruchs muss, wie schon erwähnt, ungleich 0 sein. Im Programm ist es im Moment noch kein Problem, einen „ungültigen“ Bruch zu erzeugen:

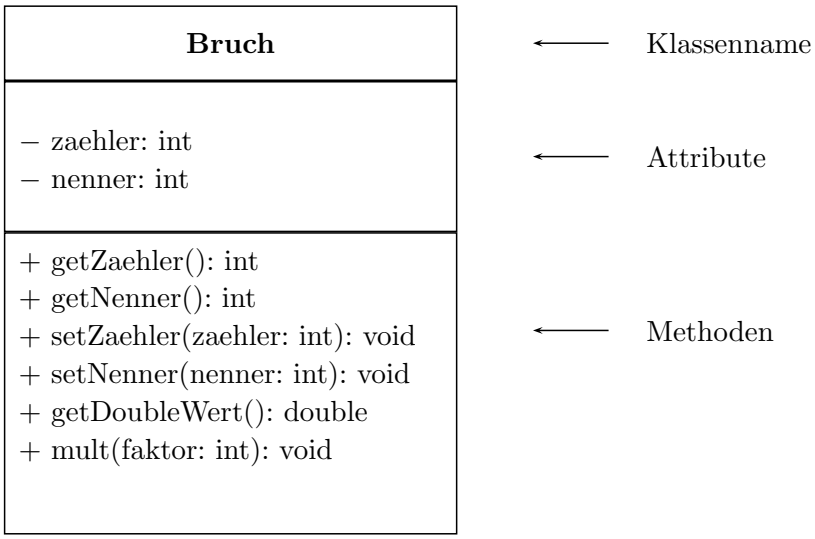
```
public static void main(String[] args) {
  Bruch r = new Bruch();
  r.nenner = 0;
}
```

Eines der Ziele der Objektorientierung ist, dass solche Operationen verhindert werden sollen. Anders ausgedrückt: Es soll erreicht werden, dass die Attribute eines Objekts keine unsinnigen Werte mehr annehmen können. Dazu muss als erstes verhindert werden, dass die Attribute frei gesetzt werden dürfen. Dazu wird das Schlüsselwort `public` durch `private` ersetzt:

```
public class Bruch {
  //Attribute
  private int zaehler;
  private int nenner;
} //public class Bruch
```

Dieses Prinzip heißt *Datenkapselung* und ist eines der erwähnten Grundprinzipien der Objektorientierung. Leider sind die Attribute jetzt etwas übertrieben gekapselt, denn der Anwender kann sie überhaupt nicht mehr ändern und auch nicht auslesen. Wir müssen also noch eine Möglichkeit implementieren, Daten kontrolliert verändern zu können. Das ist eine Hauptaufgabe der *Methoden*. Methoden sind dabei so etwas, wie die *Schnittstelle zu den Attributen*. Ein Zugriff auf die Attribute eines Objekts darf nur über seine Methoden erfolgen.

Das UML-Diagramm unserer Vorlage sieht (verkürzt) folgendermaßen aus:



3.6.6 Getter- und Setter-Methoden

Die Getter- und Setter-Methoden dienen dazu, Attribute zu ändern (Setter) oder auszulesen (Getter). Fügen wir zunächst die beiden Methoden für den Zähler hinzu:

```
public class Bruch {
  //Attribute
  private int zaehler;
  private int nenner;

  public void setZaehler(int zaehler) {
    this.zaehler = zaehler;
  }

  public int getZaehler() {
    return this.zaehler;
  }

}//public class Bruch
```

Auch hier gibt es einige interessante Punkte:

- In der Deklaration

  `public void setZaehler(int zaehler) {`

  taucht ein Übergabeparameter auf. Er wird in der Form

  `Datentyp Variablenname`

  angegeben. Das bedeutet, wenn wir die Methode aufrufen, z.B. mit

```
Bruch b = new Bruch();
b.setZaehler(2);
```

dass dann innerhalb der Methode `setZaehler` eine Variable `zaehler` verfügbar ist, die den Wert 2 hat (aber ganz normal geändert werden kann).

- Die nächste Zeile

  ```
  this.zaehler = zaehler;
  ```

  ist ebenfalls interessant. Die Variable `zaehler` enthält den Übergabeparameter, den wir gerade besprochen haben. Mit `this.zaehler` wird das Attribut *zaehler* der Klasse `Bruch` angesprochen. Es wird also das Attribut auf den Wert der Übergabevariablen gesetzt. Man muss Übergabeparameter und Attribut nicht gleich benennen. Es ist aber in Java bei Setter-Methoden üblich. In der Getter-Methode gibt es keine lokale Variable gleichen Namens. Daher könnte man dort statt `this.zaehler` auch einfach `zaehler` schreiben.

- Methoden einer Klasse können auf Attribute zugreifen, auch wenn sie *private* sind. Das heißt, innerhalb der Klasse `Bruch` ist ein Zugriff auf das Attribut `zaehler` möglich, von anderen Klassen jedoch nicht.

- Der Name `setZaehler` ist nicht zwingend vorgeschrieben, folgt aber einer Java-Konvention. Gängigerweise wählt man den Namen einer Getter- bzw. Setter-Methode wie folgt:

Setter-Methode	`set`+Variablenname
Getter-Methode	`get`+Variablenname
Getter-Methode (boolean-Variable)	`is`+Variablenname

  Beispiel: Die Methode zum Setzen der Variable `zaehler` würde man

  ```
  setZaehler(..)
  ```

  nennen. Der erste Buchstabe des Variablennamens wird dabei groß geschrieben.

  Hinweis: Die Verwendung deutscher Variablennamen führt zu englisch-deutschen Wortgemischen in den Getter- und Setter-Methoden. Dies ist ein starkes Argument für englische Variablennamen. Hier wird darauf verzichtet, da *setZaehler* und *setNenner* verständlicher sind, als die englischen Entsprechungen *setNumerator* und *setDenominator*.

Einschränkung des Zugriffs auf den Nenner

Für den Nenner sieht die Getter-Methode ähnlich aus wie beim Zähler. In der Setter-Methode erhält in unserer Vorlage der Anwender jedoch eine Exception, falls er versucht, den Nenner auf 0 zu setzen. Dieses Verhalten müssen wir

als Entwickler jetzt hinzufügen. Man sagt dazu, wir müssen eine Exception *auslösen* oder (etwas umgangssprachlicher) eine Exception *werfen*. Das geht folgendermaßen:

```
public void setNenner(int nenner) {
  if (nenner==0) {
    //Exception ausloesen
    ArithmeticException e = new ArithmeticException
                            ("Nenner darf nicht 0 werden.");
    throw e;
  } else {
    this.nenner = nenner;
  }
}
```

Zum Auslösen einer Exception erzeugt man sich ein Objekt einer Exception-Klasse (hier: *ArithmeticException*). Dem Konstruktor kann man einen String mit einer näheren Erklärung als Parameter übergeben, der dann in der Fehlermeldung erscheint. Dieses Exception-Objekt wird in der nächste Zeile durch das Schlüsselwort `throw` ausgelöst. Diese beiden Zeilen kann man auch in eine zusammenfassen:

```
throw new ArithmeticException("Nenner darf nicht 0 werden.");
```

Wir testen das Programm wieder aus Anwendersicht:

```
public class BruchTest {
  public static void main(String[] args) {
    Bruch r = new Bruch();
    r.setNenner(0);
  }
}
```

ergibt die gewünschte Fehlermeldung (die natürlich vom Anwender auch mit try-catch gefangen werden kann):

```
Exception in thread "main" java.lang.ArithmeticException:
                              Nenner darf nicht 0 werden.
at Bruch.setNenner(Bruch.java:21)
at BruchTest.main(BruchTest.java:4)
```

3.6.7 Konstruktoren

Obwohl wir in unserer Bruch-Klasse bisher keinerlei Konstruktoren implementiert haben, können wir uns bereits Bruch-Objekte erzeugen. Java stellt uns dafür automatisch einen Konstruktor zur Verfügung, nämlich den *Default-* oder *parameterlosen Konstruktor*. Parameterlos heißt, dass wir beim `new`-Aufruf

```
Bruch b = new Bruch();
```

keine Parameter übergeben haben. Die 3 Konstruktoren mit Parametern aus unserer Vorlage gibt es aber noch nicht. Das wollen wir jetzt ändern. Generell dienen Konstruktoren dazu, die Attribute gleich beim Erzeugen des Objekts setzen zu können. Fangen wir mit dem Konstruktor an, dem Zähler und Nenner als Integer-Wert übergeben werden:

```
public Bruch(int zaehler, int nenner) {
  setZaehler(zaehler);
  setNenner(nenner);
}
```

Ein Konstruktor sieht ähnlich aus wie eine Methode, unterscheidet sich aber in folgenden Punkten:

- Der „Methodenname“ ist gleich dem Namen der Klasse. In unserem Beispiel ist er also `Bruch`.
- Es gibt keinen Rückgabewert. Es wird noch nicht einmal `void` als Rückgabewert angegeben. Konstruktoren können grundsätzlich keinen Rückgabewert haben, weswegen die Angabe `void` einfach eingespart wird.

Machen wir uns noch einmal an einem Beispiel klar, dass es unmöglich ist, dass ein Konstruktor einen Rückgabewert zurückgibt. Ein Konstruktor wird *ausschließlich dann* aufgerufen, wenn ein neues Objekt mit `new` erzeugt wird, also z.B.

```
Bruch r = new Bruch(5, 3);
```

Ein Konstruktor wie

```
  public int Bruch(int zaehler, int nenner) {
    setZaehler(zaehler);
    setNenner(nenner);
    return 1;
  }
```

hätte keine Möglichkeit, die zurückgegebene 1 an eine Variable zu übergeben. `r` wird ja schon mit dem neu erzeugten Objekt belegt.

Methodenaufrufe und Exceptions in Konstruktoren

In unserem Konstruktor haben wir die Methoden `setZaehler` und `setNenner` aufgerufen. Methodenausrufe sind Konstruktoren erlaubt. Es ist hier auch sinnvoll, weil natürlich auch im Konstruktor darauf geachtet werden muss, dass der Anwender nicht mit z.B.

```
Bruch b = new Bruch(1,0);
```

einen Bruch mit dem Nenner 0 erzeugt. Um die Fehlerkontrolle nicht zweimal programmieren zu müssen, ruft man einfach die Methode `setNenner` auf. Der obige Aufruf löst dann eine Exception aus. Es ist allerdings nicht erlaubt, aus einer Methode heraus einen Konstruktor direkt (also ohne `new`) aufzurufen.
Ein weiterer Effekt ist, dass das Objekt *nicht erzeugt* wird, wenn im Konstruktor eine Exception auftritt. Die Zeilen:

```
Bruch b;
try {
  b = new Bruch(1,0);
} catch (ArithmeticException e) {
  //Fehler ignorieren
}
System.out.println(b);
```

würden als Ergebnis `null` ergeben, denn das zu `b` gehörige Objekt wurde nicht erzeugt.

Der Default-Konstruktor

Wenn wir unseren neuen Konstruktor geschrieben haben, dann funktioniert der Aufruf

```
Bruch r = new Bruch();
```

plötzlich *nicht* mehr. Was passiert hier? Der Default-Konstruktor

```
public Bruch() {
}
```

wird von Java immer genau dann hinzugefügt, wenn der Entwickler selbst keinen Konstruktor angegeben hat. Ist ein Konstruktor angegeben, fällt der Default-Konstruktor automatisch weg. Ist er dennoch gewünscht, muss er wieder hinzugefügt werden. Aussehen könnte das z.B. so:

```
  //Default-Konstruktor
  public Bruch() {
    this.zaehler = 1;
    this.nenner = 1;
  }
```

Der Vorteil ist, dass der Default-Konstruktor jetzt einen Bruch $\frac{1}{1}$ anstelle des unsinnigen Bruchs $\frac{0}{0}$ erzeugt. Natürlich ist es in diesem Beispiel noch besser, den Default-Konstruktor ganz wegzulassen.

Der Copy-Konstruktor

Dieser Konstruktor-Typ ist für Java ein Konstruktor wie jeder andere. Er hat einen speziellen Namen, weil er in der Praxis sehr häufig auftaucht. Er erzeugt ein neues Objekt, dessen Attribute aus einem zweiten, übergebenen Objekt kopiert werden. Ein Copy-Konstruktor zur Klasse `Bruch` sieht folgendermaßen aus:

```
public Bruch(Bruch r) {
  this.zaehler = r.zaehler;
  this.nenner = r.nenner;
}
```

Auch hier gibt es einige interessante Bemerkungen:

- Obwohl Zähler und Nenner `private` sind, konnten wir direkt auf `r.zaehler` und `r.nenner` zugreifen. Die Regel dazu ist folgende:

  > Aus der Klasse Bruch heraus kann auf die privaten Attribute *aller* Bruch-Objekte (nicht nur des `this`-Objekts) zugegriffen werden.

  Man kann das so verstehen: Dem Entwickler der Klasse Bruch wird die Möglichkeit eingeräumt, auf die privaten Attribute *aller* Bruch-Objekte zugreifen zu können. Der Anwender der Klasse darf direkt auf *kein* privates Attribut zugreifen.

- Wir haben hier die Attribute direkt über eine Zuweisung gesetzt und nicht über den Aufruf der Setter-Methoden. Der Grund dafür ist, dass hier schon garantiert ist, dass der Nenner von `r` ungleich 0 ist, und wir das nicht noch einmal überprüfen müssen.

Aufruf von Konstruktoren aus anderen Konstruktoren

Konstruktoren dürfen aus anderen Konstruktoren heraus aufgerufen werden. Dazu gibt es den *this-Befehl*. Zum Beispiel:

```
this(10, 10);
```

Da dies aber relativ selten vorkommt, werden wir nicht weiter darauf eingehen. Wichtig ist dabei, dass der this-Befehl der *erste* Befehl innerhalb des Konstruktors sein muss.

Sichtbarkeit von Konstruktoren

Es ist möglich, Konstruktoren `private` zu deklarieren. Dann kann ein Konstruktor nicht mehr außerhalb der eigenen Klasse aufgerufen werden. In speziellen Fällen ist das auch sinnvoll, aber hier sind wir bereits in einem fortgeschrittenen Gebiet. In den allermeisten Fällen werden Konstruktoren als `public` deklariert.

3.6.8 Invarianten

Bisher wurden in unserer Bruch-Klasse die Brüche nicht automatisch gekürzt. Außerdem kann man den Nenner problemlos auf eine negative Zahl setzen. Die beiden Bedingungen

- Der Bruch ist gekürzt.
- Der Nenner des Bruchs ist größer als 0.

nennt man die *Invarianten* unserer Vorlage. Invarianten sind Bedingungen, die jeder Methode, bevor sie sich beenden darf, erfüllen muss. Aus Anwendersicht sind die Invarianten also immer erfüllt. Aus Entwicklersicht ist das nicht ganz

der Fall, da die Invarianten *mitten in einer Methode* verletzt werden dürfen, solange sie nur vor dem Methodenende wieder hergestellt werden.
Nehmen wir als Beispiel eine Methode `setZaehler`, die die Invariante „Bruch ist gekürzt“ einhält:

- Ein Bruch `b` habe den Wert 4/9.
- Der Anwender ruft die `b.setZaehler(3)` auf. Innerhalb der Methode geschieht folgendes:
  - Der Zähler wird auf 3 gesetzt. Damit hat der Bruch den Wert 3/9. Die Invariante „Bruch ist gekürzt“ ist verletzt. Der Anwender kann dies aber nicht feststellen.
  - Der Bruch wird gekürzt. Er hat jetzt den Wert 1/3.
- Nach dem Ende der Methode sieht der Anwender das Ergebnis: 1/3. Die Invariante ist aus Anwendersicht eingehalten.

Wir programmieren im Folgenden eine Methode, die die Invarianten wieder herstellt, und die von allen Methoden, die sie verletzen könnten, am Ende aufgerufen wird. Diese Methode soll `private` sein, denn sie dient nur internen Zwecken und soll nicht in der API auftauchen.

```
private void normalisiere() {
  if (nenner < 0) {
    nenner = nenner * -1;
    zaehler = zaehler * -1;
  }
  // Bruch kuerzen
  int a = zaehler;
  if (a < 0) {
    a = -a;
  }
  int b = nenner;
  // Suchen des GGT
  // Euklidischer Algorithmus
  while (b != 0) {
    int h = a % b;
    a = b;
    b = h;
  }
  // a ist der GGT
  zaehler = zaehler / a;
  nenner = nenner / a;
}
```

Zum Kürzen verwenden wir den sogenannten *Euklidischen Algorithmus*, der hier nicht weiter erklärt sein soll. Nun müssen wir noch alle Methoden korrigieren, die die Invarianten eventuell verletzen könnten. Als Beispiel dazu diene `setZaehler`:

```
public void setZaehler(int zaehler) {
  this.zaehler = zaehler;
  normalisiere();
}
```

Damit der Konstruktor nicht zweimal normalisiert, müssen wir ihn noch anpassen:

```
public Bruch(int zaehler, int nenner) {
  this.zaehler = zaehler;  //normalisiert nicht
  setNenner(nenner);       //normalisiert
}
```

3.6.9 Andere Methoden

Die restlichen Methoden der Klasse `Bruch` sind jetzt vergleichsweise einfach. Zum Beispiel sehen `getDoubleWert` und `multipliziere(int faktor)` folgendermaßen aus:

```
public double getDoubleWert() {
  //Casten in double fuer Fliesskomma-Division
  return this.zaehler / (double) this.nenner;
}
public void multipliziere(int faktor) {
  this.zaehler = this.zaehler * faktor;
}
```

3.7 Objekte als Attribute und Parameter

3.7.1 Objekte als Attribute

Attribute können wiederum Objekte sein:

```
public class ZweiBrueche {
  public Bruch c1;
  public Bruch c2;
}
```

Die Verwendung geht wie folgt:

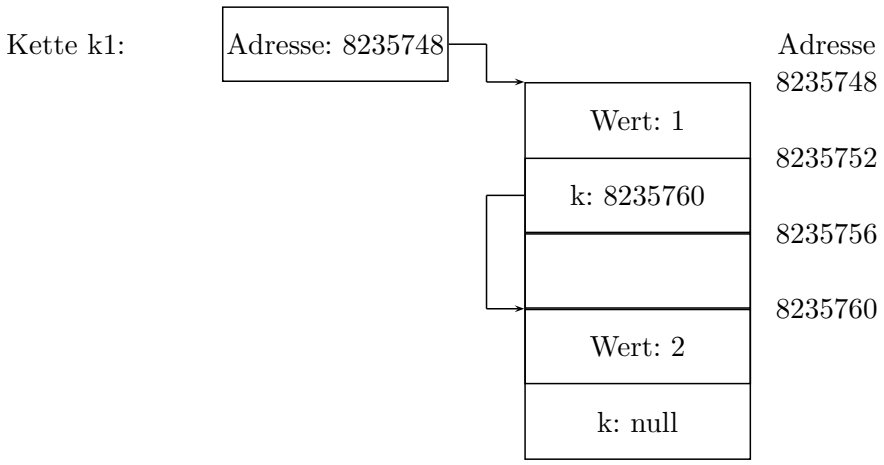
```
ZweiBrueche z = new ZweiBrueche();
z.c1 = new Bruch(3,5);
z.c1.mult(2);
```

Das Attribut eines Objekts kann auch die gleiche Klasse haben wie das Objekt selbst. Das ist kein Problem, wie folgendes Code-Beispiel zeigt:

```
public class Kette {

  Kette k;
```

```
  int wert;


  public static void main(String[] args) {
    Kette k1 = new Kette();
    k1.wert = 1;
    Kette k2 = new Kette();
    k2.wert = 2;

    k1.k = k2;
    System.out.println(k1.wert);
    System.out.println(k1.k.wert);
    System.out.println(k1.k.k);
  } //main
} //class
```

Intern sieht das so aus:

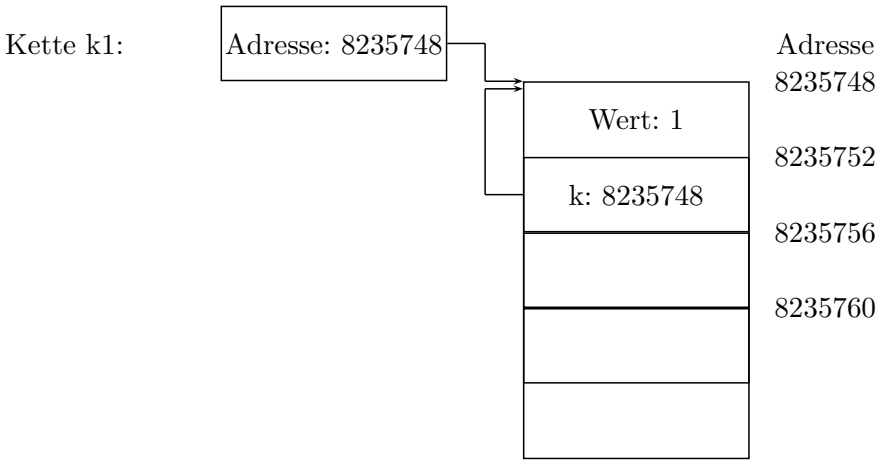


Diese Konstruktion heißt *einfach verkettete Liste* und wird in der Vorlesung *Algorithmen und Datenstrukturen* im nächsten Semester ausführlicher behandelt. Auch Referenzen auf sich selbst sind kein Problem:

```
  k1.k = k1;
```

Dies sieht intern so aus:[10]

[10] Hier wäre die Klebezettel-Darstellung überfordert.

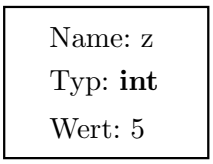


3.7.2 Objekte als Übergabeparameter

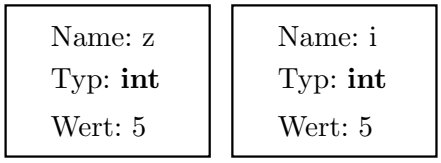
Zu diesem Kapitel benötigen wir wieder die Klasse `Bruch`. Außerdem benutzen wir eine zweite Klasse `Haupt`, die keine Attribute und nur statische Methoden hat.

```
public class Haupt {
  public static void zahlMalZwei(double i) {
    i = i * 2;
  }

  public static void bruchMalZwei(Bruch r) {
    r.multipliziere(2);
  }

  public static void main(String[] args) {
    double z= 5;
    zahlMalZwei(z);
    System.out.println(z);    //-->5

    Bruch b = new Bruch(5,1);
    bruchMalZwei(b);
    System.out.println(b);    //-->10/1
  }

}
```

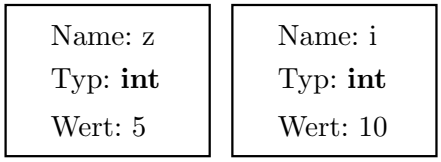
Die Parameterübergabe beim Aufruf von `ZahlMalZwei` kann man folgendermaßen darstellen:



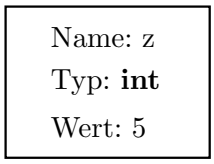
Vor dem Methodenaufruf



Am Beginn der Methode

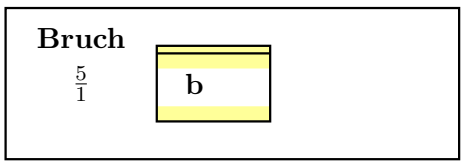


Am Ende der Methode

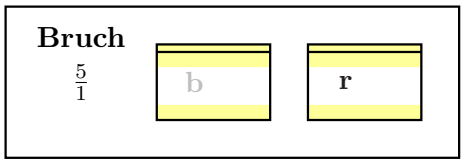


Nach dem Methodenaufruf

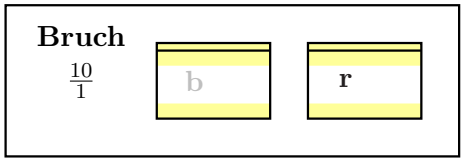
Wichtig ist, dass die Variable `z` beim Methodenaufruf in die Variable `i` *kopiert* wird. Wird `i` in der Methode verändert, hat das keine Auswirkung auf `z`. Dies nennt man „Call by value“ oder Wertübergabe.
Bei der zweiten Methode `BruchMalZwei` wird im Gegensatz dazu ein Objekt übergeben. Die Situation hier lässt sich gut mit der Klebezettel-Darstellung verstehen:[11]



Vor dem Methodenaufruf

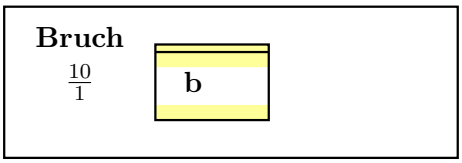


Am Beginn der Methode



Am Ende der Methode

[11] Schattierte Variablennamen bedeuten, dass diese Variable im momentanen Fokus nicht ansprechbar ist.



Nach dem Methodenaufruf

Der springende Punkt ist, dass `b` eine *Objekt-Variable* ist und Objekte als *Referenz* gespeichert werden. Bei der Übergabe des Wertes wird nicht der Wert selbst kopiert, sondern die *Speicheradresse des Werts*. `b` und `r` verweisen also auf *das gleiche Objekt*. Wenn also `r` verändert wird, wird gleichzeitig auch `b` verändert. Diese Übergabe nennt man „Call by Reference“ oder Referenzübergabe.

Bei *Call by Value* wird der Wert selbst kopiert. Bei *Call by Reference* wird nur die Speicheradresse kopiert, d.h. Änderungen des Wertes werden quasi „mit zurückgegeben“.

In Java hängt es vom übergebenen Datentyp ab, ob Wertübergabe oder Referenzübergabe ausgeführt wird.

- Primitive Datentypen sind Wertparameter: Die Übergabe erfolgt als Wertübergabe; Änderungen innerhalb der Methode wirken sich außerhalb nicht aus.
- Objekte sind Referenzparameter: Die Übergabe erfolgt als Referenzübergabe; Änderungen innerhalb der Methode wirken sich auch außerhalb aus.

Wir betrachten noch einmal eine andere Situation. Wir wollen eine Methode schreiben, die einen Bruch auf den Wert $\frac{1}{1}$ setzt. Der Aufruf soll folgendermaßen aussehen:

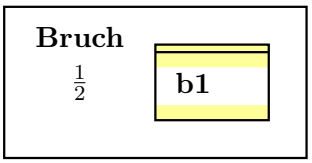
```
Bruch b1 = new Bruch(1,2); //Dummy-Wert
setToOne(b1);              //b1 wird auf 1/1 gesetzt.
```

Unser erster Versuch hat das folgende Aussehen:

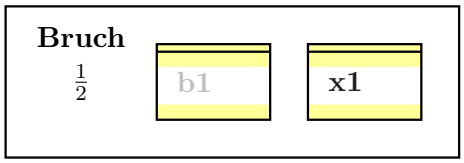
```
public void setToOne(Bruch x1) {
  Bruch z = new Bruch(1,1);
  x1 = z;
}
```

Da Brüche als Referenzen übergeben werden, hoffen wir, dass am Ende der Methode `b1` den Wert 1 besitzt. Leider ist dies nicht der Fall. Wir betrachten uns dazu für jeden Schritt die Variablen:

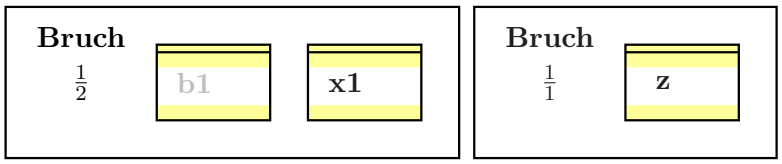
Vor dem Methodenaufruf



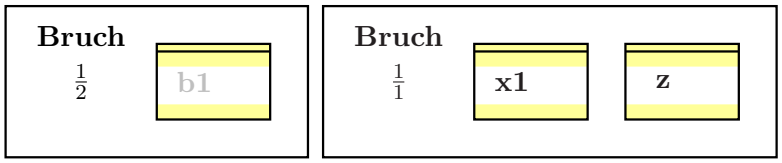
Am Anfang der Methode



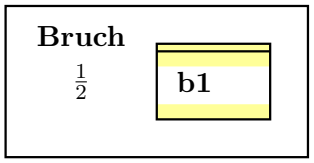
Nach `Bruch z = new Bruch(1,1);`



Nach `x1 = z;` *(Klebe x1 auf das Objekt, auf dem schon z klebt.)*



Nach dem Methodenaufruf



Wichtig ist hier, dass der Aufruf `x1 = z;` zwar den Zettel *x1* umklebt, aber nicht den Zettel *b1*. Man muss für den gewünschten Effekt das übergebene Objekt selbst verändern, so wie im folgenden Beispiel:

```
public void setToOne(Bruch x1) {
  x1.setZaehler(1);
  x1.setNenner(1);
}
```

3.7.3 Mehrere Rückgabewerte

Eine Methode oder eine Funktion kann mehrere Eingangsparameter, aber maximal einen Ausgangsparameter haben. Es gibt aber Fälle, in denen man sich eine Funktion mit mehreren Ausgangsparametern wünscht. Nehmen wir als Beispiel eine Funktion `zerlege`, die ein Bruch-Objekt `x` als Eingangsparameter erhält und zwei Bruch-Objekte zurückgeben soll, in denen der ganzzahlige Anteil sowie der Rest enthalten ist. Es gibt drei Möglichkeiten, dies zu erreichen:

- Man gibt ein Feld mit zwei Bruch-Werten zurück:

```
public static Bruch[] zerlege(Bruch x)
```

- Man schreibt für den Rückgabewert eine eigene Klasse (hier `ZerlegterBruch`):

```
public static ZerlegterBruch zerlege(Bruch x)
```

- Man verwendet einen (oder zwei) zusätzliche Übergabeparameter als Rückgabewerte. Bedingung ist, dass die zusätzlichen Übergabeparameter Objekte sind und damit als Referenz übergeben werden:

```
public void zerlege(Bruch b, Bruch ganzzahl, Bruch rest)
```

Diese Möglichkeit wird wie folgt aufgerufen:

```
Bruch b = new Bruch(7,5);
Bruch ganzzahl = new Bruch(1,1);  //Dummy-Wert
Bruch rest = new Bruch(1,1);      //Dummy-Wert
zerlege(b, ganzzahl, rest);
//innerhalb von zerlege(..) erhalten ganzzahl und rest die
//gewuenschten Werte.
```

3.8 Weitere Details

3.8.1 Statische Variablen

Auch Variablen können das Schlüsselwort `static` erhalten. Damit erhält man eine Variable, die mit

```
Klassenname.variablenname
```

aus allen Klassen heraus angesprochen werden kann. Eine solche Variable nennt man auch *globale Variable*. Sie existiert pro Klasse nur einmal, gleichgültig, wie viele Objekte es von dieser Klasse gibt. Als Beispiel wollen wir ein Programm erstellen, das zählt, wieviele Objekte einer Klasse erstellt wurden.

```
public class ObjektZaehler {

  private static int anzahl = 0;  //statische Variable

  public ObjektZaehler() {
    ObjektZaehler.anzahl++;
  }

  public static int getAnzahl() {
    return ObjektZaehler.anzahl;
  }

}
```

Es gibt die statische Variable `ObjektZaehler.anzahl`, die eindeutig ist und nur ein einziges Mal existiert (innerhalb der Klasse `ObjektZaehler` könnte man sie auch einfach mit `anzahl` ansprechen). Jedes Mal, wenn ein neues Objekt vom Typ `ObjektZaehler` erzeugt wird, wird die Variable um eins hochgezählt. Die Funktion `getAnzahl` zur Abfrage der Variablen ist ebenfalls statisch, kann auf statische Variablen (im Gegensatz zu Attributen) aber problemlos zugreifen.

3.8.2 Die Bedeutung von `public static void main(String args[])`

Auch die Funktion `main`, mit der ein Programm startet, besitzt Übergabeparameter. Um zu verstehen, welche Werte dort übergeben werden, müssen wir wissen, was intern geschieht, wenn man in einer Windows-Eingabeaufforderung oder einer Linux-Konsole ein Java-Programm mit folgender Zeile startet:

```
java ClassXYZ par1 par2
```

- Es wird die Datei *ClassXYZ.class* gesucht.
- Innerhalb dieser Klasse wird die Funktion

  `public static void main(String args[])}`

  gesucht. Wenn sie nicht gefunden wird, wird eine Fehlermeldung zurückgegeben.
- Die Strings *par1* und *par2* (die sogenannten *Kommandozeilen-Parameter* werden in ein String-Feld gepackt und stehen in der Main-Routine als args[] zur Verfügung.

In Eclipse kann man einem Java-Programm auf folgende Weise Kommandozeilen-Parameter mitgeben:

- *Run* → *Run Configurations...*
- Im Auswahlfenster Name des Java-Programms auswählen.
- Reiter *Arguments*.
- Kommandozeilen-Parameter unter *Program arguments* eintragen.

Chapter 4

Ausnahmebehandlung (Exception Handling)

Das Thema „Exceptions“ verteilt sich in diesem Skript über mehrere Kapitel. Um das Nachschlagen zu erleichtern, werden darum zunächst die Teile aus Kapitel 3 kurz zusammengefasst. Anschließend folgt eine Vertiefung des Themas. Zuletzt folgt eine Zusammenfassung der Vorgehensweise zur Erstellung eigener Exceptions, die in Kapitel 7 beschrieben ist und für die Kenntnisse zur Vererbung benötigt werden.

4.1 Auslösen und Fangen von Exceptions (Zusammenfassung von Kapitel 3)

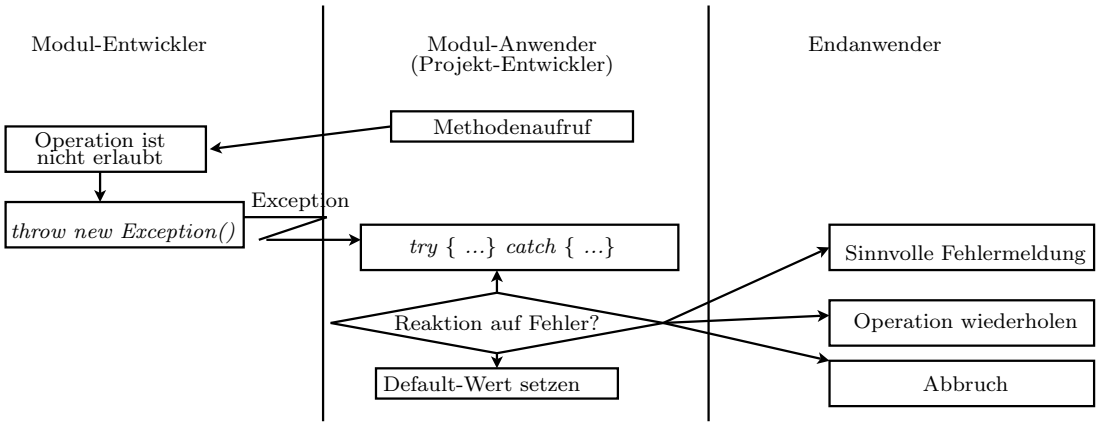
Beginnen wir noch einmal mit den verschiedenen Rollenperspektiven.

- Der *Modulentwickler* schreibt eine bestimmte Klasse K. Diese Klasse soll möglichst universell in mehreren Projekten verwendet werden.
- Der *Projektentwicker* schreibt ein komplettes Programm P und benutzt dabei die Klasse K.
- Der *Endanwender* benutzt das Programm P.

Wir nehmen an, der Projektentwickler benutzt die Klasse K fehlerhaft.
Beispiel: K ist die Klasse, die einen Bruch repräsentiert und der Projektentwickler versucht, den Nenner des Bruchs auf 0 zu setzen. Dann muss der Modulentwicker dafür sorgen, dass die Klasse K eine *Exception* auslöst. Dazu erzeugt er sich ein Objekt einer Exception-Klasse und löst sie mit dem Schlüsselwort `throw` aus. Beispiel:

```
public void setNenner(int nenner) {
  if (nenner==0) {
    throw new ArithmeticException("Nenner darf nicht 0 werden.");
  } else {
    this.nenner = nenner;
}
```

Falls der Projektentwickler die Exception nicht fängt, bricht das Programm (genauer: der Thread) mit einer Java-Fehlermeldung ab. Da der Endanwender darüber sicherlich nicht erfreut wäre, muss der Projektentwickler die Exception fangen und entsprechend darauf reagieren. Zum Beispiel könnte er dem Endanwender eine passende Fehlermeldung ausgeben.



Exceptions werden gefangen, wenn sie in einem `try`-Block stehen und es einen `catch`-Block für die passende Exception gibt. Beispiel:

```
try {                                              //1
  bruch.setNenner(x);                              //2
  System.out.println("Eingabe ok");                //3
} catch (ArithmeticException e) {                  //4
  System.out.println("Eingabe wird ignoriert");    //5
}                                                  //6
System.out.println("Weiter geht's");               //7
```

Falls der Aufruf von `setNenner` erfolgreich war, ist die Reihenfolge der durchlaufenen Zeilen 1,2,3,7. Gab es bei `setNenner` eine Exception, ist die Reihenfolge der durchlaufenen Zeilen 1,2,5,7.

4.2 Reaktion auf Ausnahmen

Im letzten Abschnitt haben wir gesehen, wie man Exceptions fängt und eigene Fehlerbehandlungsroutinen aufruft. Im Folgenden gehen wir über den Stoff von Kapitel 3 hinaus.
Die nächste Frage ist, was im entsprechenden catch-Block sinnvollerweise zu tun ist. Wir untersuchen jetzt also den Code...

```
try {
  ...
} catch (xyzException e) {
  //... der an dieser Stelle stehen sollte.
}
```

4.2.1 Analyse von Ausnahmen im catch-Block

Zunächst wollen wir möglichst viel Information über die Art des Fehlers erhalten. Es ist wichtig zu wissen, dass Exceptions eigentlich nur eine spezielle Art von Klassen sind, die die Besonderheit haben, dass sie geworfen (oder ausgelöst) werden können. Sehen wir uns an einem Beispiel den genauen Ablauf an:

```
public class ExceptionTest {
  public static void main(String[] args) {
    try {
      int c = 5/0;
      System.out.println("Division gelungen");
    } catch (ArithmeticException e) {
      System.out.println(e.getMessage()); //ergibt / by zero
    }
  }
}
```

Die Zeile `int c=5/0;` bewirkt,

- dass Java intern ein neues Objekt der Klasse *ArithmeticException* erzeugt,
- dass dieses Objekt „geworfen“ oder „ausgelöst“ und damit die Fehlerbehandlungsroutine angesprungen (oder das Programm beendet) wird
- und im catch-Block über die Variable `e` auf das Exception-Objekt zugegriffen werden kann. Es entspricht einer Konvention, als Variable *e* zu verwenden, der Name ist aber beliebig:

```
catch (ArithmeticException beliebigerName)
```

Dem Exception-Objekt kann man in der Fehlerbehandlungsroutine mehr Informationen über den aufgetretenen Fehler entlocken:

1. Die Exception-Klasse (hier *ArithmeticException*) weist schon auf die Art des Fehlers hin.
2. Mit `String s = e.getMessage();` erhält man eine nähere Erläuterung des Fehlers.
3. Mit `e.printStackTrace();` kann man die übliche Fehlermeldung auf dem Bildschirm ausgeben und mit `e.getStackTrace();` kann man die Programmzeile ermitteln, an der der Fehler aufgetreten ist (für genauere Information bitte in der API nachschlagen).

4.2.2 Reaktion auf Ausnahmen

Was schreibt man nun in einen catch-Block? Man muss sich für eine von drei Möglichkeiten entscheiden:

- Wiederholen: Zum Beispiel Benutzereingaben kann man wiederholen lassen.

- Default-Werte: Man kann versuchen, den fehlerhaften Wert durch einen Default-Wert zu ersetzen. Exisitiert z.B. eine Konfigurations-Datei nicht, kann man eine Default-Konfiguration benutzen.

- Abbrechen: Der Fehler ist so schwerwiegend, dass das Programm nicht korrekt fortgesetzt werden kann (z.B. Festplatte nicht erreichbar) oder ein Programmierfehler ist aufgetreten. Dann kann man das Programm abbrechen. Eine naheliegende Wahl ist ein Abbruch, der dem normalen Abbruch gleicht, wenn die Exception nicht gefangen wurde. Die Zeilen dazu sind:

```
....
}catch (Exception e) {
  e.printStackTrace();
  System.exit(1);
}
```

4.2.3 Mehrere unterschiedliche Exceptions

Im letzten Beispiel wurde eine `ArithmeticException` gefangen. Andere Exceptions lösen dort nach wie vor einen Programmabbruch aus. Ein try-catch-Block darf aber auch mehrere verschiedenen Exceptions fangen. Die Syntax hierzu sieht folgendermaßen aus:

```
try {
....
} catch (ArrayIndexOutOfBoundsException e) {
....
} catch (NumberFormatException e) {
....
}
```

Dann wird, je nach Exception-Klasse, eine der Fehlerbehandlungsroutinen angesprungen. Es ist möglich, alle Exceptions in einer einzigen Fehlerbehandlungsroutine abzufangen, indem man

```
try {
  ....
} catch (Exception e) { .... }
```

schreibt. Damit kann man verhindern, dass eine unerwartete Exception ein Programm zum Absturz bringt. Um dann allerdings feststellen zu können, *welche* Exception nun aufgetreten ist, braucht man den *instanceof*-Operator, der erst in den nächsten Kapiteln erläutert wird. Allgemein wird diese Methode als schlechter Stil betrachtet.

4.2.4 Verschachtelte try-catch-Blöcke

try-catch-Blöcke dürfen auch verschachtelt werden. Sowohl im try- als auch im catch-Block dürfen weitere try-catch-Blöcke eingebettet sein. Falls eine Exception auftritt, wird der innerste passende catch-Block angesprungen. Verschachtelte try-catch-Blöcke sind schlecht für die übersichtlichkeit eines Programms. Verwenden sie sie nur sparsam.

4.3 Checked und unchecked Exceptions

Die Exceptions, die wir bisher betrachtet haben, gehören alle zu den *unchecked exceptions*. Es gibt außerdem noch die *checked exceptions*, die sich etwas anders verhalten. Die Häufigsten sind die *IOException* und die *FileNotFoundException*. Diese Exceptions werden ausgelöst, wenn z.B. beim Schreiben oder Lesen von Dateien ein Fehler auftritt. Wenn *checked exceptions* nicht mit try-catch gefangen werden, lässt sich das Programm nicht compilieren (und noch weniger ausführen). Auch Eclipse zeigt gleich eine Fehlermeldung an. Es gilt also festzuhalten:

- Unchecked Exceptions *können* durch try-catch abgefangen werden. Bei nicht-Abfangen wird das Programm mit einer Fehlermeldung beendet.
- Checked Exceptions *müssen* mit try-catch abgefangen werden.

Die Philosophie der Java-Entwickler ist, dass sauber geschriebene Programme für manche Fehler eine Fehlerbehandlungsroutine beinhalten *müssen*. Ein gutes Beispiel ist das Öffnen einer Datei auf der Festplatte:

```
Scanner sc = new Scanner(new File("matrix.dat"));
```

Hier sagen die Entwickler: Wenn ein Java-Programmierer eine Datei öffnet, muss er eine Fehlerbehandlung für den Fall schreiben, dass die Datei nicht auf der Festplatte vorhanden ist. In dieser Fehlerbehandlung könnte er den Benutzer eine andere Datei auswählen lassen, Default-Werte nehmen oder das Programm abbrechen lassen.

Vergleich der Java-Syntax mit der anderer Sprachen

Java hat zwei Besonderheiten, die sich von sprachenunabhängigen Einteilungen, aber auch von C# oder Python deutlich abheben.

- Was allgemein als „Exception“ bezeichnet wird, heißt in Java *Throwable*. Throwables teilen sich auf in *Errors* und *Exceptions*. Java versucht, zwischen Systemfehlern (Errors) und Programmfehlern (Exceptions) zu unterscheiden, was in anderen Sprachen so nicht anzutreffen ist.
- *Checked* Exceptions sind ebenfalls eine Java-Spezialität. In anderen Sprachen, wie C++, C# oder Python sind alle Exceptions *unchecked*.

Ob eine Exception eine *checked* ist oder nicht, kann man in der Java-API daran erkennen, ob ganz oben in der Darstellung der Klassenhierarchie das Wort „RuntimeException“ vorkommt oder nicht.[1]
Beispiele:

```
IOException: checked

java.lang.Object
  extended by java.lang.Throwable
      extended by java.lang.Exception
          extended by java.io.IOException



ArithmeticException: unchecked

java.lang.Object
  extended by java.lang.Throwable
      extended by java.lang.Exception
          extended by java.lang.RuntimeException
              extended by java.lang.ArithmeticException
```

4.3.1 Welche Exception werfe ich?

Zunächst steht die Entscheidung an, ob die Exception *checked* oder *unchecked* sein soll. Die Frage ist: „Will ich mich und andere dazu zwingen, den Fehler zu behandeln oder nicht?“ Dazu gibt es keine klaren Regeln, nur Hinweise:

- Wenn die ArithmeticException *checked* wäre, müsste jede Integer-Division in einen try-catch-Block gesteckt werden. Das würde den Programmcode stark aufblähen.

- Es ist schlechter Programmierstil, eine Datei zu öffnen und nicht zu überprüfen, ob die Datei überhaupt vorhanden ist. Irgendwann wird die Datei einmal nicht vorhanden sein. Daher ist die FileNotFoundException *checked*.

Anschließend sieht man in der Java-API nach, ob es eine passende Exception gibt (z.B. wenn in der Java-Bibliothek ein ähnlicher Fehlerfall auftritt). Falls keine passende Exception vorhanden ist, kann man selbst eine Exception-Klasse schreiben, wozu wir aber noch weiteres Wissen über die Objektorientierung brauchen. Daher nur die Stichworte:

- *checked Exceptions* müssen von `Exception` abgeleitet werden.

- *unchecked Exceptions* müssen von `RuntimeException` abgeleitet werden.

[1] Das heißt, ob die Klasse von der Klasse `RuntimeException` abgeleitet ist oder nicht.

4.3.2 Besonderheiten beim Auslösen einer „checked exception“

Entscheidet man sich beim vorigen Beispiel für eine *checked exception*, gibt es eine Besonderheit:

```
public void setNenner(int nenner) throws IOException {
  if (nenner != 0) {
    this.nenner = nenner;
  } else {
    throw new IOException("Nenner gleich 0");
  }
}
```

Im Methodenkopf müssen alle *checked exceptions* angegeben werden, die aus der Methode „herauskommen“ können. Dies geschieht mit dem Schlüsselwort `throws`, das von den möglichen Exceptions (evtl. durch Komma getrennt) gefolgt wird. *Unchecked exceptions* können, aber müssen hier nicht angegeben werden.

4.4 Vererbung und Exceptions

Während der gr-o–ss-te Teil dieses Unterkapitels Stoff aus Kapitel 7 voraussetzt, kann man den folgenden Abschnitt auch mit den bisherigen Kenntnissen benutzen, wenn man hinnimmt, dass der genaue Mechanismus noch nicht komplett verstanden werden kann.

4.4.1 Schreiben eigener Exceptions

Oft findet man in der Java-API keine Exception mit dem passenden Namen. Manchmal will man auch von vornherein eine eigene Exception benutzen und Verwechslungen mit der Java-API ausschließen. Zum Beispiel wollen wir uns eine Exception mit dem Namen `BruchException` erzeugen. Wir haben die Wahl, ob die neue Exception *checked* oder *unchecked* sein soll. Für eine *unchecked* Exception sieht der Code folgendermaßen aus:

```
public class BruchException extends RuntimeException {
  public BruchException(String s) {
    super(s);
  }
}
```

Für eine *checked* Exception muss nur in der ersten Zeile das Wort `RuntimeException` durch `Exception` ersetzt werden:

```
public class BruchException extends Exception {
  public BruchException(String s) {
    super(s);
  }
}
```

4.4.2 Vererbungshierarchie von Exceptions

Ab hier sind Kenntnisse aus Kapitel 7 nötig. Exceptions sind in Java Unterklassen der Klasse *Exception*. Außer Exceptions können in Java auch *Errors* geworfen werden, doch dies sollte der Java-Runtime selbst vorbehalten bleiben. Die Java-Vererbungshierarchie sieht stark verkürzt wie folgt aus:

```
Throwable (unchecked)
   |
   |
   +------- Error (unchecked)
   |
   +------- Exception (checked)
               |
               |
               +--------- IOException (checked)
               |             |
               |             +-- FileNotFoundException (checked)
               |
               +--------- RuntimeException (unchecked)
               |
               +--------- ...
```

Zum Ableiten bieten sich besonders die Exception-Basisklassen an. Je nachdem, welche Basisklasse man wählt, erhält man Exceptions unterschiedlichen Typs:

Basisklasse	Bemerkung
Throwable	Nicht empfohlen.
Error	Nicht empfohlen (werden nur vom System ausgelöst).
Exception	Checked Exception ohne spezielle Eigenschaften.
IOException	Checked Exception im Zusammenhang mit IO-Operationen. Vorteil: Alle IOExceptions können mit *catch(IOException e)* gemeinsam gefangen werden.
RuntimeException	Unchecked Exception ohne spezielle Eigenschaften.

Auch in anderen objektorientierten Sprachen findet sich solch eine Hierarchie. Die Trennung in *Throwable*, *Error* und *Exception* ist in Java recht eigenwillig. In C# oder Python sind alle Exceptions von *Exception* abgeleitet.

Beispiel: Ein Programm soll aus einer Datei eine Matrix einlesen. Falls die Daten keine Matrix darstellen, soll eine *MatrixFormatException* geworfen werden, die selbst zu schreiben ist. Die Exception soll *checked* sein. Da der Fehler beim Einlesen einer Datei auftritt, soll es eine IOException sein. Der einfachste Code ist:

```
public class MatrixFormatException extends IOException {}
```

Die Exception wird dann wie eine normale Exception eingesetzt.

```
if(....) {
  throw new MatrixFormatException();
}
```

und muss entsprechend in einem try-catch-Block gefangen werden.

Exceptions aus der Java-Klassenbibliothek haben noch einen String *message* als Zusatzinformation, die der Exception im Konstruktor übergeben wird. Um das bei der selbstgeschriebenen Exception nutzen zu können, muss der Klasse ein entsprechender Konstruktor hinzugefügt werden:

```
public class MatrixFormatException extends IOException {

    public MatrixFormatException(String message) {
        super(message);
    }
}
```

Der Konstruktor ruft einfach den Konstruktor der Basisklasse auf. Es ist prinzipiell möglich (aber nur beschränkt sinnvoll), der Exception noch beliebige andere Funktionalität hinzuzufügen.

4.4.3 Fangen ganzer Gruppen von Exceptions

Es ist möglich, in einem catch-Block ganze Gruppen von Exceptions zu fangen, wenn man die Exception-Basisklassen fängt. Zum Beispiel fängt

```
try {
...
} catch (RuntimeException e) {
...
}
```

alle Runtime-Exceptions, d.h. alle Klassen, die von *RuntimeException* abgeleitet sind. Die Variable *e* im catch-Block hat dann aber auch den (statischen) Typ *RuntimeException*. Wenn man wissen will, welchen dynamischen Typ *e* hat, also welche Exception wirklich aufgetreten ist, muss man mit *instanceof* den dynamischen Typ abfragen. Noch mehr Exceptions werden abgefangen, wenn man die Typen *Exception* oder sogar *Throwable* verwendet.

Chapter 5

Die Java-Klassenbibliothek

5.1 Die Java-Laufzeitumgebung

Programme in der Programmiersprache Java nutzen die Java-Laufzeitumgebung (*Java Runtime Environment / JRE).* Die Laufzeitumgebung besteht aus der *Java Virtual Machine (JVM)*, die Java-Programme ausführt und aus der Klassenbibliothek. Programmiersprache, JVM und Klassenbibliothek hängen nicht untrennbar zusammen. Oracle bietet zum Beispiel eine alternative JVM unter dem Namen GraalVM an. Vor allem aber gibt es für die Java-Laufzeitumgebung mehrere alternative Programmiersprachen, wie Kotlin, Groovy oder Scala. Die praktische Bedeutung ist, dass man auch aus Kotlin oder Groovy heraus Funktionen wie „System.out.println“ oder „JOptionPane. showInputDialog“ benutzen kann. Microsoft hat prinzipiell ein ähnliches Konzept, trennt die Begriffe aber. Das Pendant der Java-Laufzeitumgebung ist dort *.NET*, und der Programmiersprache Java entspricht die Sprache *C#*.[1] Andere Sprachen für .NET sind C++ oder VB.NET.
Neue Programmiersprachen bauen oft auf eine der beiden Plattformen auf. Beide Plattformen sind sehr aufwendig und haben einen großen Vorsprung gegenüber Neuentwicklungen. Die Zahl der Plattformen ist daher wesentlich kleiner als die Zahl der Programmiersprachen.
Wir haben mittlerweile das Rüstzeug, um die meisten Informationen aus der Java-API zu verstehen. Sehen wir uns also einige nützliche Klassen aus der Java-Klassenbibliothek an.

5.2 Listen und Dateien

Eine Liste ist in Java etwas Ähnliches wie ein Feld. Im Unterschied zu Feldern können Listen aber wachsen und schrumpfen. Dadurch sind sie in Java, wie auch in anderen Programmiersprachen, sehr beliebt. Es gibt in Java mehrere Arten von Listen mit der gleichen Funktionalität, die aber intern unterschiedlich implementiert sind. Bemerkbar macht sich das durch unterschiedliche Laufzeiten. Dies ist jedoch ein Thema der Vorlesung *Algorithmen und Datenstrukturen.*

[1] Der Standard der Plattform heißt *Common Language Infrastructure (CLI).* Die bekanntesten Implementationen der CLI sind .NET und Mono.

In der aktuellen Vorlesung verwenden wir die meistbenutzte Listen-Klasse: `ArrayList`. Eine andere Klasse, die wir nicht verwenden, ist *LinkedList.* Es gibt auch einen Datentyp, der sowohl ArrayLists, als auch LinkedLists aufnehmen kann. Dieser Datentyp heißt einfach `List`.[2] Im Moment reichen uns die beiden folgenden Punkte:

- Wenn wir selbst eine Liste erzeugen, verwenden wir eine `ArrayList`.
- Wenn wir von Java-Funktionen ein Objekt vom Typ `List` erhalten, können wir das Objekt wie eine ArrayList verwenden. Eventuelle Performance-Probleme interessieren uns hier nicht. Sie treten bei Listen unter 100.000 Elementen gewöhnlich nicht auf.

Es gibt viele Anwendungsmöglichkeiten von Listen. Eine davon ist es, eine Datei von der Festplatte einzulesen. Zu Beginn ist oft nicht bekannt, wie groß die Datei ist. Da ist es natürlich sehr praktisch, dass eine Liste wachsen kann.

5.2.1 Aufgabenstellung

Für diesen Abschnitt stellen wir uns die folgende Aufgabe. Es gibt eine Datei `summe.dat`, die eine Liste von Geldbeträgen enthält. Die Datei kann beispielsweise so aussehen:

```
2.95
15.50
9.65
2.45
1.99
```

Pro Zeile gibt es einen Geldbetrag. Das Dezimalzeichen ist ein Punkt. Die Aufgabe ist es, der Datei einen Summenstrich (=====) und den Gesamtbetrag hinzuzufügen.

Vorgehensweise

Wir wählen dazu eine möglichst universelle Vorgehensweise. Diese sieht so aus:

- Zunächst lesen wir die Datei komplett in eine String-Liste ein. Dadurch muss die Größe der Datei vorher nicht bekannt sein.
- Dann durchlaufen wir die Liste und bestimmen die Summe.
- Anschließend fügen wir den Summenstrich und die Summe der Liste hinzu.
- Am Ende schreiben wir die komplette Liste wieder in eine Datei.

Um mit Listen umgehen zu können, brauchen wir zunächst Grundkenntnisse in zwei anderen Gebieten: Den *Generics* (oder generischen Datentypen) und den *Wrapper-Klassen.*

[2] Wir werden später sehen, dass `List` ein *Interface* ist.

5.2.2 Generische Datentypen (Generics)

Die erste Aufgabe ist, festzulegen, aus welcher Art von Daten die Liste bestehen soll. Dazu benötigt man die sogenannten *generischen Datentypen*. Ein generische `ArrayList` wird zum Beispiel so erzeugt:

```
ArrayList<String> zeilen = new ArrayList<String>();
```

`String` ist der *Typparameter*, der in zwei spitzen Klammern steht. Das Beispiel erzeugt eine `ArrayList`, die nur Strings aufnehmen kann.
Auf die Dauer stört bei generischen Datentypen, dass man bei der Initialisierung den recht langen Datentyp (hier `ArrayList<String>`) doppelt eingeben muss, obwohl dies weder nötig ist noch den Code leichter verständlich macht. Daher wurden in Java zwei Abkürzungen eingeführt. Ab Java 7 kann man in der `new`-Anweisung den generischen Typ durch den *Diamond-Operator* `<>` ersetzen. Die obige Zeile lässt sich dadurch wie folgt verkürzen:

```
ArrayList<String> zeilen = new ArrayList<>();
```

Ab Java 10 geht es noch kürzer, indem man `var` für die Deklaration verwendet. `var` und Diamond-Operator dürfen aber nicht zusammen verwendet werden.

```
var zeilen = new ArrayList<String>();
```

Das verbessert die Lesbarkeit, da überlange Code-Zeilen verkürzt werden und der Typ immer noch in der `new`-Anweisung klar erkennbar ist. Andere, nichtgenrische Klassen können zwar genauso mit `var` deklariert werden, wie hier gezeigt wird:

```
var f = "Test";
```

Aber hier stellt sich die Frage, ob eine leichte Verkürzung der Deklaration es rechtfertigt, eine unspezifischere Syntax zu verwenden. In diesem Skript wird daher `var` nur bei generischen Datentypen (und später bei anonymen inneren Klassen) verwendet.
Generic-Programmierung ist ein komplexes Thema, auf das im Rahmen dieser Vorlesung nicht weiter eingegangen wird. Sie ähneln den Templates aus C++, welche allerdings nochmal deutlich mächtiger sind. Eine Besonderheit wird allerdings noch im nächsten Kapitel angesprochen.

5.2.3 Wrapper-Klassen

In Generics dürfen nur Klassen angegeben werden, keine primitiven Datentypen. Die Zeile

```
//KEIN KORREKTES JAVA
ArrayList<int> zahlen = new ArrayList<int>();
```

ist in Java falsch. Wenn primitive Datentypen gespeichert werden sollen, muss man den Umweg über die *Wrapper-Klassen* wählen. Es gibt für jeden primitiven Datentyp eine eigene Wrapper-Klasse. Der Name der Wrapper-Klasse leitet sich

vom primitiven Datentyp ab. Gewöhnlich wird der erste Buchstabe groß statt klein geschrieben (Ausnahmen: int, char).

prim. Datentyp	**Wrapper-Klasse**
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Eine Wrapper-Klasse packt einen primitiven Datentyp in ein Objekt ein. Der wesentliche Teil der Wrapper-Klasse `Integer` sieht folgendermaßen aus:

```
public class Integer {

    private final int value;

    public Integer(int value) {
        this.value = value;
    }

    public int intValue() {
        return value;
    }
}
```

Das Schlüsselwort `final` bedeutet, dass der Wert von `value` nur ein einziges Mal gesetzt werden kann (in diesem Fall im Konstruktor). Man packt einen int-Wert mit

```
int i = 5;
Integer wr = new Integer(i);
```

ein und mit

```
int j = wr.intValue();
```

wieder aus[3].

5.2.4 Autoboxing

Das Verfahren zum Ein- und Auspacken in Wrapper-Klassen wurde mit Java 1.5 durch das sogenannte *Autoboxing*, eine Art automatischer Umwandlung zwischen primitivem Datentyp und Wrapperklasse, stark vereinfacht. Ein- und Auspacken geschieht in den folgenden Zeilen automatisch im Hintergrund:

[3] Siehe auch den folgenden Abschnitt: Autoboxing.

```
int i = 5;
Integer wr = i;  //Einpacken
int j = wr;      //Auspacken
```

Sollen primitive Datentypen in einer `ArrayList` gespeichert werden, gibt man im Generic die entsprechende Wrapper-Klasse an:

```
var liste = new ArrayList<Integer>();
```

Beim Hinzufügen von `int`-Werten werden diese automatisch in ein Wrapper-Objekt gepackt.

5.2.5 Einsatz von ArrayLists mit Wrapper-Klassen und Autoboxing

Zunächst wird eine ArrayList erzeugt. Wir erzeugen jeweils eine für Strings (ohne Wrapper-Klassen) und Integers (mit Wrapper-Klassen).

```
var stringListe = new ArrayList<String>();
var intListe = new ArrayList<Integer>();
```

Die Methode `add` fügt einer ArrayList ein neues Element hinzu:

```
stringListe.add("Hallo");
intListe.add(5);
```

Die Elemente sind innerhalb der Liste durchnummeriert. Das erste Element hat den Index 0. Die aktuelle Länge der Liste kann man mit

```
int len = stringListe.size();
```

erfragen. Das Element mit dem Index `i` kann man wie im folgenden Beispiel gezeigt auslesen und setzen:

```
int a = intListe.get(i);
String s = stringListe.get(i);

intListe.set(i, 45);
stringListe.set(i, "Tschuess");
```

Die interessantesten Methoden der Klasse ArrayList sind in der folgenden Tabelle kurz zusammengefasst. Die genaue Syntax findet sich in der Java-API.

`add(E o)`	Hängt ein neues Element hinten an.
`add(int index, E o)`	Fügt ein neues Element an der Stelle *index* ein. Nachfolgende Element rücken eine Position nach hinten.
`set(int index, E o)`	Setzt ein neues Element an die Stelle *index*. Überschreibt das alte Element.
`E get (int index)`	Gibt das Element Nr. *index* zurück.
`indexOf(Object elem)`	Gibt den Index zurück, an dem das Element zum 1. Mal vorkommt (sonst -1).
`remove(int index)`	Löscht das Element an der Position *index*.
`clear()`	Löscht die Liste.
`size()`	Gibt die Zahl der Elemente zurück.

5.2.6 Einlesen einer Datei in eine Liste

Man kann eine Datei mit einem einzigen Befehl in eine String-Liste einlesen. Der Befehl, um eine Datei `summe.dat` in eine Liste einzulesen, lautet:

```
List<String> lines = Files.readAllLines(Path.of("summe.dat"));
```

Hierbei sind mehrere Punkte zu beachten:

- `Path.of` gibt ein Objekt vom Typ `Path` zurück, welches einen Pfad zu einer Datei beschreibt. Die Pfadnamen können absolut oder relativ sein. Der Pfadname oben ist ein relativer Pfadname, da er relativ zu einem vorgegebenen Verzeichnis ist. Benutzt man Eclipse oder IntelliJ, sucht Java die Datei `summe.dat` im Projektverzeichnis. Ein absoluter Pfadname unter Windows wäre `C:\test\summe.dat`.
- Jeder Zeile der Datei wird zu einemListenelementen von `s`.
- Die Methode `readAllLines` kann eine `IOException` werfen. Diese Exception ist anders als die, die wir bisher behandelt haben. Im Gegensatz zu den bisherigen *muss* sie gefangen werden. Dazu schieben wir einen kleinen Unterabschnitt ein.

Checked Exceptions

Eine Besonderheit von Java sind die *Checked Exceptions*. „Normale“ bzw. *Unchecked* Exceptions müssen nicht gefangen werden. Wird eine Exception ausgelöst, die im Programm nicht gefangen wird, wird das Programm[4] (genauer: der Thread) abgebrochen und eine Fehlermeldung angezeigt. Wird eine Checked Exception nicht gefangen, lässt sich das Programm weder compilieren noch starten und die Entwicklungsumgebung zeigt einen Fehler an. Mehr zu Checked und Unchecked Exceptions findet sich in Abschnitt 4.3.

[4]Genauer gesagt, wird der Thread abgebrochen. Dieser Unterschied hat eine Bedeutung bei der Parallelprogrammierung in Java.

5.2.7 Verarbeitung und Ausgabe

Eine Liste kann mit einer for-Schleife oder einer foreach-Schleife durchlaufen werden. Im folgenden Beispiel werden einfach alle Listenelemente auf dem Bildschirm ausgegeben:

```
//for-Schleife
for (int i=0; i<lines.size(); i++) {
  System.out.println(lines.get(i));
}

//foreach-Schleife
for (String s: lines) {
  System.out.println(s);
}
```

Wir parsen in einer foreach-Schleife alle Zeilen in doubles und fügen den Summenstrich und die Summe hinzu. Am Ende wird mit `Files.write` die komplette Liste wieder in eine Datei geschrieben. Auch `write` kann eine IOException werfen.

```
public static void summiere(String dateiname) throws IOException {
  List<String> data = Files.readAllLines(Path.of(dateiname));
  double summe = 0;
  for (String s: data) {
    summe += Double.parseDouble(s);
  }
  data.add("=======");
  data.add(String.format(Locale.US,"%5.2f",summe));
  Files.write(Path.of(dateiname+".out"), data);
}
```

In der vorletzen Zeile, die die Summe berechnet, stecken noch einige Besonderheiten:

- Die Summenangabe soll zwei Stellen hinter dem Komma haben. Dazu benötigt man die Formatierung mit `String.format`. Zur Struktur des Formatstrings siehe Anhang A.
- Der erste Parameter `Locale.US` sorgt dafür, dass ein Dezimalpunkt ausgegeben wird und kein Dezimalkomma, auch wenn die Systemeinstellungen auf „deutsch“ stehen.

Anstatt die komplette Datei neu zu schreiben, kann man auch die zwei neuen Zeilen an die Datei anhängen. Das geht folgendermaßen:

```
data.clear();
data.add("=======");
data.add(String.format(Locale.US,"%5.2f",summe));
Files.write(Path.of(dateiname+".out"), data, StandardOpenOption.APPEND);
```

5.2.8 Verschiedene Dateiformate

In der Computerwelt unterscheiden sich Textdateien ja nach Betriebssystem. Dabei gibt es zwei Varianten: Die eine Variante wird von Windows verwendet. Die andere Variante benutzen alle anderen Betriebssysteme und das Internet. Der Kürze wegen nennen wir die zweite Variante die Linux-Variante.

Zeilenumbruch

Ein Zeilenumbruch wird unter Linux durch die Escape-Sequenz `\n` dargestellt, unter Windows ist es `\r\n`. `Files.readAllLines` erkennt beide Varianten. `Files.write` nimmt die Variante des benutzten Betriebssystems.

Umlaute

Für Zeichen, die nicht im ASCII-Standard enthalten sind, gibt es unterschiedliche Codierungen. Dies betrifft vor allem die Umlaute. Relevant sind:

- der von Windows benutzte Standard *CP-1252.*[5]
- der vom Rest der Computerwelt benutzte Standard *UTF-8.*

Per Default nimmt Java jeweils die dem Betriebssytem entsprechende Codierung. Will man das ändern, muss man die Codierung angeben, wie in den folgenden Beispielen gezeigt:

```
Files.readAllLines(Path.of(dateiname), StandardCharsets.UTF_8);
Files.readAllLines(Path.of(dateiname), Charset.forName("windows-1252"));
Files.readAllLines(Path.of(dateiname), StandardCharsets.ISO_8859_1);

Files.write(Path.of(dateiname), data, StandardCharsets.UTF_8);  //aequivalent
```

Für CP-1252 gibt es keine Konstante in StandardCharsets und man muss auf `Charset.forName` ausweichen. Vielleicht wehrt sich Oracle damit gegen reine Microsoft-Standards.

5.3 Die Klasse Scanner

5.3.1 Kurze Historie der I/O-Funktionen unter Java

Die I/O (Input/Output)-Funktionen wurden in Java in mehreren Schüben erweitert und verbessert. Dadurch gibt es unterschiedliche Wege, in Java Dateien ein- und auszulesen. In diesem Skript wird der mit der Klasse `Files` der neueste Ansatz verwendet. In alten Java-Programmen findet man aber auch heute noch häufig die älteren Ansätze. Die folgende Liste kann man als Ausgangspunkt einer Internet-Recherche verwenden:

[5] Zwei stark verwandte Standards sind ISO 8859-1 und ISO 8859-15. Vorsicht beim Euro-Zeichen: Hier unterscheiden sich die drei Standards.

ab Java-Version	Klassen zum Einlesen	Klassen zum Auslesen	Klasse für Pfadangaben
1.1	java.io.BufferedReader	java.io.PrintWriter	java.io.File
1.5	java.util.Scanner	java.io.PrintWriter	java.io.File
1.7	java.nio.file.Files	java.nio.file.Files	java.nio.file.Path

In Java 1.7 wurde zusätzlich zum Paket `java.io` das Paket `java.nio` (new io) eingeführt. Das Ziel von java.nio ist eine größere Geschwindigkeit von IO-Operationen. Die Klasse `Scanner`, die mit dem alten java.io-Paket zusammen verwendet wird, bietet andererseits einige Möglichkeiten, die mit java.nio nicht so einfach umgesetzt werden können.

5.3.2 Eingabe über die Konsole

Bisher mussten wir für die Eingabe über Tastatur immer ein Grafikfenster öffnen.
Zur Eingabe über die Konsole benötigt man ein Objekt der Klasse *java.util.Scanner*. Mit den Methoden dieses Objekts kann man primitive Datentypen und Strings einlesen. Zunächst muss das Paket *java.util* importiert werden. Dann muss ein Scanner-Objekt erzeugt werden. Im Konstruktor wird dem Scanner mitgeteilt, woher er seine Daten beziehen soll.[6]

```
Scanner sc = new Scanner(System.in);
```

erzeugt einen Scanner, der seine Daten von der Konsole einliest. Mit diesem Objekt kann man nun zeilenweise Daten von der Konsole (bzw. der Tastatur) einlesen.

```
import java.util.*;

public class Tastatur {

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Eingabe: ");
    String s = sc.nextLine();
    System.out.println("Eingabe: "+s);
  }
} //class
```

5.3.3 Prüfen eines Strings auf einen Zahlenwert

Die Funktion `Integer.parseInt` hat einen Nachteil. Wenn man prüfen will, ob ein String einen Zahlenwert enthält, ist man darauf angewiesen, die `NumberFormatException` zu fangen. Das ergibt unschönen Code.

```
String x = "5";
boolean isZahl = false;
```

[6] Noch einfacher geht die Eingabe mit `System.console().readLine()`. Leider funktioniert diese Funktion nicht innerhalb einer IDE.

```
try {
  Integer.parseInt(x);
  isZahl = true;
} catch(NumberFormatException e) {
}
```

Man sollte Exceptions *nur* zur Behandlung von Fehlern verwenden. Die bessere Alternative ist, zur Prüfung einen Scanner zu benutzen.

```
String x = "5";
Scanner sc = new Scanner(x);
boolean isZahl = sc.hasNextInt();
```

5.3.4 Zeilenweises Einlesen einer Datei

Mit diesen Zeilen liest man eine Datei Zeile für Zeile ein. Manchmal benötigt man das. Hier nur ein kurzes Beispiel (ohne Klausurrelevanz):

```
try (Scanner sc = new Scanner(new File("summe.dat"))) {
  while(sc.hasNextLine()) {
    System.out.println(sc.hasNextLine());
  }
} catch (FileNotFoundException e) {
  //Datei nicht gefunden
}
```

Beachtenswert ist die erste Zeile. Man nennt diese Form *try with resource* oder *Automatic Resource Management.* Der Effekt ist, dass die Datei `summe.dat` automatisch geschlossen wird, wenn das Programm den try-Block verlässt.

5.4 Assoziative Felder

Bisher hatten wir Felder mit fortlaufendem Index erzeugt. Wir wollen jetzt eine Tabelle anlegen, die die Städtekürzel auf Nummernschildern den jeweiligen Städten zuordnet. Sehr praktisch wäre ein Feld, dessen Index auch aus Strings bestehen kann. Ein Aufruf sieht dann aus, wie:

```
// C# ! Kein Java !
tabelle["AC"]=Äachen";
tabelle["HS"]="Heinsberg";
tabelle["DN"]="Dueren";

String x = tabelle["AC"];   //Aachen
```

Assoziative Felder sind wichtig und werden häufig gebraucht. In vielen Programmiersprachen ist exakt diese Syntax möglich. In Java gibt es zwar auch assoziative Felder, aber die Syntax ist etwas anders. Zur besseren Beschreibung führen wir zunächst einige Begriffe ein. In der Zeile

```
// C# ! Kein Java !
tabelle["AC"]="Aachen";
```

ist `AC` der *Schlüssel* (bei normalen Feldern würde man „Index“ sagen). `Aachen` ist der *Wert*, der zum Schlüssel `AC` gehört. Ein Assoziatives Feld speichert also immer Schlüssel-Wert-Paare.
Für assoziative Felder gibt es in Java mehrere Klassen. Die meistgebrauchte davon heißt *HashMap*. Im Konstruktor wird zunächst per Generic (siehe das Kapitel über dynamische Felder) festgelegt, von welchem Datentyp Schlüssel und Wert sein sollen. Im folgenden Beispiel sind Schlüssel und Wert beide vom Datentyp String.

```
var tabelle = new HashMap<String, String>();
```

Dann kann man mit `put(Schluessel, Wert)` Werte unter einer Schlüsselbezeichnung speichern:

```
tabelle.put("AC","Aachen");
```

und anschließend auf das entsprechende Feldelement zugreifen:

```
String s = tabelle.get("AC");
```

Die Abfrage eines Wertes, der nicht vorhanden ist

```
String s = tabelle.get("K");
```

gibt `null` zurück.
Man kann auch alle anderen Klassen für Schlüssel und Wert wählen. Wählt man Integer als Schlüssel, kann man ein Feld erstellen, deren Indizes nicht fortlaufend sein müssen, wie das folgende Beispiel einer Liste von Aachener Buslinien zeigt:

```
var linien = new HashMap<Integer, String>();
linien.put(2, "Eilendorf - Bushof");
linien.put(5, "Uniklinik - Driescher Hof");
linien.put(33, "Fuchserde - Vaals");
```

HashMaps ohne Generics

Ebenso wie ArrayLists können HashMaps erzeugt werden, ohne Generics anzugeben:

```
HashMap h = new HashMap();
```

Eine solche HashMap kann als Schlüssel wie auch als Wert *alle* Objekte aufnehmen. Auch hier fehlen uns dazu noch einige Grundlagen der Objektorientierung.

Geschwindigkeit

Die der Klasse `HashMap` zugrundeliegene Datenstruktur ist die *Hashtabelle*, die in der Vorlesung „Algorithmen und Datenstrukturen“ erklärt wird. Diese Datenstruktur ermöglicht, als Schlüssel beliebige Datentypen zu verwenden und trotzdem sowohl den Speicherverbauch gering zu halten, als auch relativ schnell den einem Schlüssel zugehörigen Wert zu finden. Es geht *deutlich* schneller als das ganze Feld zu durchsuchen.

Für Experten

Es gibt außer `HashMap` zwei weitere ähnliche Klassen: `Hashtable` und `TreeMap`. Alle diese drei Klassen können wie oben angegeben benutzt werden. Die Unterschiede liegen eher im Detail.

- `HashMap` und `Hashtable` sind sehr ähnlich. Der Hauptunterschied besteht darin, dass auf `Hashtable` nur ein einziger Thread gleichzeitig zugreifen kann. Die dazu nötigen Synchronisationsmechanismen machen `Hashtable` langsamer als `HashMap`.
- `TreeMap` hat als interne Datenstruktur einen Rot-Schwarz-Baum, während `HashMap` eine Hashtabelle besitzt. Eine Hashtabelle ist deutlich schneller als ein Rot-Schwarz-Baum und auch als die binäre Suche in einem Feld. Ein Rot-Schwarz-Baum hat allerdings den Vorteil, dass die Elemente sortiert vorliegen.

5.5 StringBuilder

Möglicherweise haben sie bereits eine Funktion vermisst, die einzelne Buchstaben in Strings ändern kann. Eine Art `setChar(..)` gibt es in der Klasse String nicht, wohl aber in der Klasse `StringBuilder`.[7] Zunächst wird ein StringBuilder für einen String erzeugt:

```
String t = "test";
StringBuilder sb = new StringBuilder(t);
```

Jetzt kann mit der Methode `setCharAt(int index, char c)` ein einzelnes Zeichen verändert werden:

```
sb.setCharAt(0, 'f');  //ergibt "fest"
```

Anschließend kann der StringBuilder mit `toString()` wieder in einen String zurückverwandelt werden:

```
t = sb.toString();
```

StringBuilder können auch die Ausführungsgeschwindigkeit eines Programms verbessern. Wird z.B. folgende Zeile ausgeführt:

```
String hw = "Hello"+"world";
```

dann werden zunächst intern die beiden Strings „Hello“ und „world“ erzeugt, beide in StringBuilder umgewandelt, sie mit append aneinander gehängt und das Ergebnis wieder in einen String gewandelt.
Betrachten wir zum Vergleich zwei Methoden, die beide einen String zurückgeben, indem sie `count` Mal einen String `x` hintereinanderhängen. Die zweite Variante arbeitet mit einem StringBuilder, die erste mit der Addition von Strings.

[7] Es gibt das historisch ältere Pendant `StringBuffer`, das aber heute nicht mehr empfohlen wird.

```
public static String mult1(String x, int count) {
  String ret = "";
  for (int i=0; i<count; i++) {
    ret += x;
  }
  return ret;
}

public static String mult2(String x, int count) {
  StringBuilder xbuild = new StringBuilder(x);
  StringBuilder ret = new StringBuilder();
  for (int i=0; i<count; i++) {
    ret.append(xbuild);
  }
  return ret.toString();
}
```

Bei hohen Werten von `count` ist die Variante mit einem StringBuilder erheblich schneller. Eine Zeitmessung ergab für 100000 Aneinanderkettungen:

mult1 (String)	26 s
mult2 (StringBuilder)	0.01 s

Die Klasse StringBuilder hat noch einige andere nützliche Methoden, die man in der Klasse String vermisst. Wichtig beim Umgang ist:

> Ein String kann nicht nachträglich verändert werden. Die String-Methoden erzeugen neue Strings. Ein StringBuilder kann dagegen sehr wohl nachträglich verändert werden.

`Beispiel:`

```
String s = "Hallo";
String t = s.substring(2,3); //Erzeugt neuen String,
                             //s bleibt unveraendert

StringBuilder sb = new StringBuilder("Hallo");
sb.substring(2,3);           //sb wird veraendert
```

Eine Auswahl der nützlichen Methoden ist:

Methode	Beschreibung
`delete(int start, int end)`	Entfernt die Zeichen von start bis end-1.
`deleteCharAt(int index)`	Entfernt Zeichen an Stelle index.
`insert(int offset, String s)`	Fügt String an Position offset ein.
`replace(int start, int end, String s)`	Ersetzt Zeichen von start bis end durch s (ohne reguläre Ausdrücke).

Chapter 6

Interfaces

6.1 Einführung

6.1.1 Problemstellung

Bei unserer Problemstellung gehen wir wieder von der Klasse `Bruch` aus Kapitel 3 aus. Die Klasse repräsentiert eine rationale Zahl. Es gibt unter anderem einen Konstruktor, dem Zähler und Nenner übergeben werden und eine `toString`-Methode zur Ausgabe. Die Zeilen

```
Bruch b = new Bruch(2,3);
System.out.println(b);
```

geben `2/3` auf dem Bildschirm aus. Jetzt kommt eine neue Anforderung auf. Das Ausgabeformat soll umstellbar sein. Es sollen auch die Formate

```
2
-
3
```

```
0.666666666666666
```

möglich sein. Erschwerend kommt hinzu, dass der Modulbenutzer sich auch eigene Formate definieren will.

6.1.2 Einfacher Lösungsansatz

Wir beginnen mit einem intuitiven, noch nicht optimalen Lösungsansatz. Wir fügen der Klasse Bruch ein neues Attribut für das Ausgabeformat hinzu:

```
private int ausgabeformat;
```

Dazu kommen noch die entsprechenden Getter- und Setter-Methoden.[1] Außerdem definieren wir uns einige statische Konstanten:

[1] Im Grunde sollte man statt dem `int`-Attribut eine Enumeration (Schlüsselwort `enum`) verwenden, aber Enumerations werden in dieser Vorlesung nicht behandelt.

```
public static final int EINZEILIG = 0;
public static final int MEHRZEILIG = 1;
public static final int DOUBLE = 2;
```

und bauen in die `toString`-Methode ein switch-Konstrukt ein:

```
public String toString() {
  switch(ausgabeformat) {
    case EINZEILIG:
      return zaehler+"/"+nenner;
    case MEHRZEILIG:
      return ......
    case DOUBLE:
      return ""+getDoubleValue();
  }
}
```

Dann können wir einfach das Ausgabeformat umstellen:

```
Bruch b = new Bruch(2,3);
b.setAusgabeformat(Bruch.DOUBLE);
System.out.println(b);     //ergibt 0.666666666666666
```

Probleme

Das Problem bei dieser Vorgehensweise ist folgendes: Wenn der Anwender ein neues, selbstdefiniertes Format hinzufügen will, muss er an mindestens zwei Stellen den Code der Bruch-Klasse ändern:

- Er muss eine neue Konstante hinzufügen.
- Er muss der switch-Anweisung einen neuen Zweig hinzufügen.

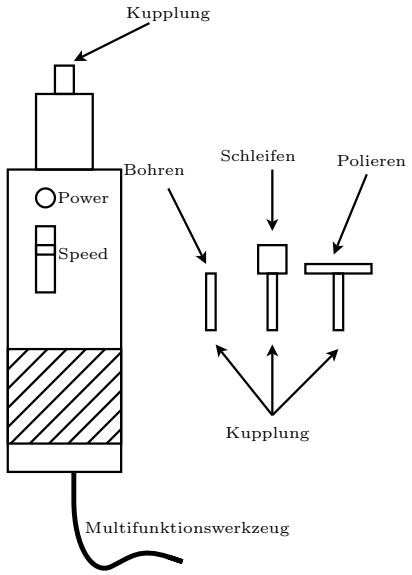
Ein Prinzip der objektorientierten Programmierung ist aber gerade, dass der Modulanwender den Code der Modul-Klassen *nicht* ändern soll. Man sagt:

- Der Source-Code der Klasse soll für den Anwender *geschlossen* sein, also nicht änderbar (und möglicherweise gar nicht verfügbar).
- Dennoch soll die Klasse *offen* sein für Anpassungen.

Bei gr-o–ss-eren Klassen würde es den Anwender auch viel Zeit und Mühe kosten, sich in den Code eines fremden Programmierers einzudenken. Und was nach einigen Änderungen diverser Anwender dann als Code herauskommt, will auch keiner mehr wirklich haben. Also brauchen wir eine andere Lösung.

6.1.3 Verbesserter Lösungsansatz

Als Modell für unseren objektorientierten Ansatz nehmen wir ein Multifunktionswerkzeug. Ein solches Werkzeug kann man sich als einen kleinen Bohrer vorstellen, an deren Spitze man ganz unterschiedliche Aufsätze befestigen kann. Es gibt zum Beispiel Aufsätze zum Bohren, zum Schleifen oder zum Polieren.



Wir unterscheiden drei Teile:

- Die *Maschine* ist das eigentliche Werkzeug mit den Bedienelementen.
- Der *Aufsatz* ist das Teil, das auswechselbar ist.
- Die *Kupplung* ist das Teil, an dem Maschine und Aufsatz zusammengesteckt werden. Wichtig dabei ist dass die Kupplungen von Maschine und Aufsatz zueinander passen.

Dieses Schema kann man mit etwas Überlegung gut auf unser Problem mit der Bruch-Klasse übertragen. Wir identifizieren zunächst die Bruch-Klasse mit der Maschine. Der Aufsatz dazu wandelt Zähler und Nenner in einen String um, beinhaltet also eine Methode, wie:

```
public String getStringDarstellung(int z, int n) {
  return z+"/"+n;
}
```

Dieser Aufsatz soll von der Maschine (dem Bruch-Objekt) aus aufgerufen werden. Die Kupplung, über die das geschieht, ist das Aussehen des Methodenaufrufs:

```
public String getStringDarstellung(int z, int n)
```

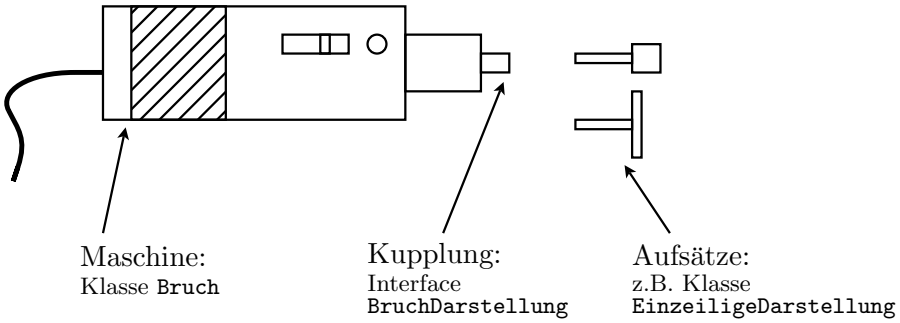
Diese Kupplung wird in Java (und auch in C#) durch ein sogenanntes *Interface* definert. Ein Interface umfasst die Definition eines oder mehrerer Methodenaufrufe. Der Java-Code dazu sieht wie folgt aus:

```
public interface BruchDarstellung {
  String getStringDarstellung(int z, int n);
}
```

An der Stelle, an der sonst `class` steht, findet sich jetzt das Schlüsselwort `interface`. Von der Methode ist nur die Kopfzeile angegeben, gefolgt von einem Semikolon. Außerdem fehlt bei der Methode das Schlüsselwort `public`, das man weglassen kann, weil Methoden eines Interfaces schon automatisch public sind.
Im nächsten Schritt wird der Aufsatz mit einer Kupplung verbunden. Ein Aufsatz ist eine Klasse mit der Methode `getStringDarstellung`. Außerdem muss Java wissen, dass der Aufsatz zu der angegebenen Kupplung gehört. Dazu dient das Schlüsselwort `implements`:

```
public class EinzeiligeDarstellung implements BruchDarstellung {
  public String getStringDarstellung(int z, int n) {
    return z+"/"+n;
  }
}
```

Dieses Schlüsselwort gibt an, dass die Klasse `EinzeiligeDarstellung` ein Aufsatz ist, der zur Kupplung `BruchDarstellung` gehört. Der Java-Compiler verlangt dann auch, dass die Klasse die Methode `getStringDarstellung` implementiert.



Als nächstes müssen wir auch an der Bruch-Klasse noch die Kupplung anbringen und uns überlegen, wie Aufsätze ausgetauscht werden sollen. Zuletzt müssen wir dafür sorgen, dass der Aufsatz innerhalb der `toString`-Methode auch angesprochen wird.
Die Kupplung in der Bruch-Klasse ist ein Attribut vom Typ des Interfaces. Zum Auswechseln des Aufsatzes benötigen wir eine entsprechende Setter-Methode. Eine Getter-Methode ist hilfreich, aber nicht elementar nötig.

```
public class Bruch {
  ...
  private BruchDarstellung darstellung;

  public void setDarstellung(BruchDarstellung darstellung) {
    this.darstellung = darstellung;
  }
  ...

}
```

Am Ende ändern wir die `toString`-Methode so ab, dass auf die `getStringDarstellung`-Methode eben dieses Objekts zugegriffen wird.

```
public String toString() {
  String ret = darstellung.getStringDarstellung(this.zaehler, this.nenner);
  return ret;
```

Damit ist aus Entwicklersicht alles erledigt.

Verwendung aus Benutzersicht

Die Verwendung des Aufsatzes aus Benutzersicht sieht so aus:

```
public static void main(String[] args) {
  Bruch b = new Bruch(1,2);

  EinzeiligeDarstellung brda = new EinzeiligeDarstellung();
  b.setDarstellung(brda);

  System.out.println(b);
```

Bemerkenswert sind die beiden mittleren Zeilen.

- Aufsatz erzeugen: In der 2. Zeile wird ein Aufsatz erzeugt. Es handelt sich um ein Objekt der Klasse `EinzeiligeDarstellung`.

- Aufsatz an Maschine anstecken: Das Aufsatz-Objekt wird der Methode `setDarstellung` übergeben. Das funktioniert, obwohl als Übergabetyp der Typ des Interfaces angegeben ist. Es funktioniert immer dann, wenn das übergebene Objekt das Interface implementiert. Wir könnten also auch ein Objekt eines anderen Typs `MehrzeiligeDarstellung` übergeben, solange es das Interface `BruchDarstellung` implementiert. Das übergebene Objekt wird dann im entsprechende Attribut abgespeichert.

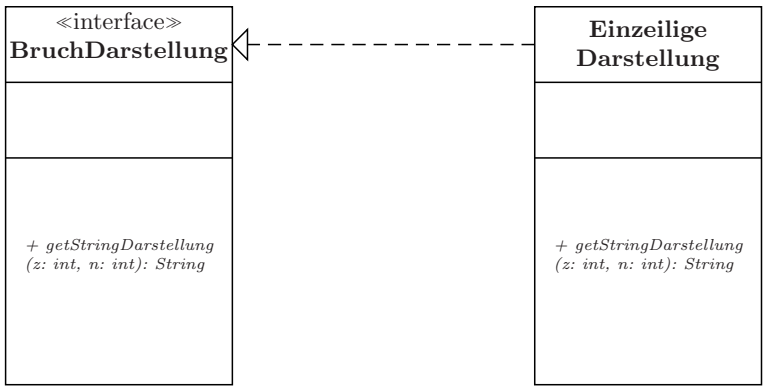
- Maschine starten: Beim Aufruf der `toString`-Methode wird die Methode des Aufsatz-Objekts angesprochen und ihr Ergebnis zurückgegeben.

6.2 Grundlegende Definitionen

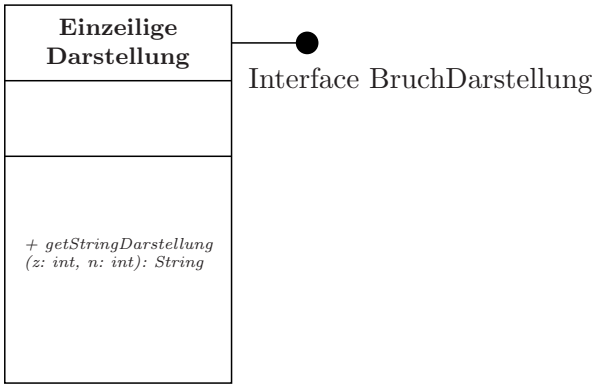
Terminologie

Der deutsche Ausdruck für Interface ist *Schnittstelle*. Die Vererbung eines Interface nennt man *Schnittstellenvererbung* oder *Subtyping*. Speziell in Java spricht man statt einer Vererbung meist von der *Implementation* eines Interfaces (so auch im weiteren Skript).

UML-Diagramm



Im Kopf des UML-Diagramms wird mit ≪interface≫ angezeigt, dass es sich um ein Interface handelt. Die Methodennamen werden kursiv geschrieben. Wird ein Interface von einer anderen Klasse implementiert, so wird das durch eine gestrichtelten Linie mit nicht ausgefülltem Pfeil angezeigt. Falls viele Klassen mehrere Interfaces implementieren, kann dies zu einem „Pfeilsalat“ führen. Deshalb ist die folgende Kurzschreibweise („Lollipop-Form“) möglich:



Variablen- und Objekttyp

Hier stoßen wir zum ersten Mal auf die Tatsache, dass der Typ einer Variablen und der Typ des Objekts, auf das die Variable zeigt, nicht übereinstimmen müssen. Die Zeile

```
BruchDarstellung bd = new EinzeiligeDarstellung();
```

funktioniert unter der Voraussetzung, dass `EinzeiligeDarstellung` das Interface `BruchDarstellung` implementiert. Allerdings kann man von der Variablen `bd` jetzt nur noch die Methoden aufrufen, die im Interface definiert sind. Ausführlich werden wir das Ganze noch im nächsten Kapitel (Vererbung) betrachten.

6.2.1 Entwurfsmuster (Programming Patterns)

Entwurfsmuster sind Lösungs-Schablonen für wiederkehrende Probleme aus der Software-Entwicklung. Eigentlich sind Entwurfsmuster ein Thema aus der Vorlesung „Software Engineering“, aber wir können an ihnen jetzt schon gut nachvollziehen, wie Interfaces typischerweise eingesetzt werden. Andersherum gesagt: Wenn man Interfaces an einem sinnvollen Beispiel erläutert, landet man fast automatisch bei einem Entwurfsmuster. Der Schritt, dieses dann auch korrekt zu benennen, ist dann nur noch klein. Trotzdem sei hier noch eine kurze allgemeine Erläuterung vorangestellt. Es gibt bei der Java-Programmierung unterschiedliche Ebenen, die nacheinander erlernt werden müssen:

- Zunächst muss man die Sprachkonstrukte als solche beherrschen.
- In der zweiten Ebene werden kleine algorithmische Probleme in Java wiedergegeben und gelöst.
- In der dritten Ebene werden gr-o–ss-ere Probleme mit objektorientierten Methoden strukturiert und gelöst.
- Die vierte Ebene, die wir jetzt angehen, ist, typische Muster aus mehreren Objekten zusammenzustellen.

Es gibt sehr viele solche Muster. Am bekanntesten sind aber die ca. 20 Muster, die in dem Standardwerk *Design Patterns* von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides zusammengestellt sind.[2] [3]

6.2.2 Das Entwurfsmuster Strategie (Strategy)

Am Anfang des Kapitels hatten wir ja bereits ein sinnvolles Beispiel für den Einsatz eines Interfaces. Damit haben wir, ohne es zu kennen, bereits das erste Entwurfsmuster programmiert. Es heißt *Strategie*. Die formale Definition nach Freeman ist:

> *Das Strategy-Muster definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar. Das Strategy-Muster ermöglicht es, den Algorithmus unabhängig von den Clients, die ihn einsetzen, variieren zu lassen.* [4]

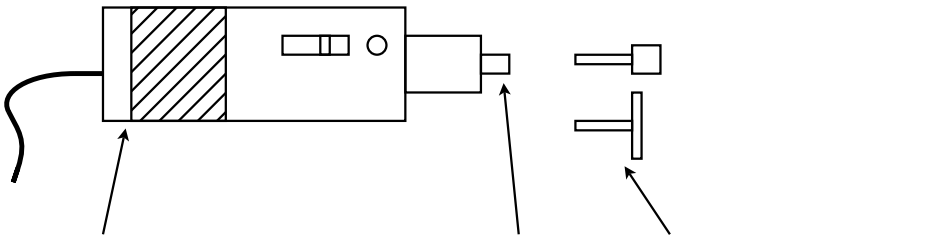
Diese Definition ist ziemlich einfach zu verstehen, wenn man sie in unser Multifunktionsgerät-Beispiel übersetzt:

---

[2] Die vier Autoren sind auch als *Gang of Four*, abgekürzt *GoF* bekannt.

[3] Empfehlenswerter für Anfänger ist das Buch *Entwurfsmuster von Kopf bis Fuß* von Eric Freemann und Elisabeth Freeman.

[4] Die Bemerkung dazu im Buch selbst ist: Nehmen sie DIESE Definition, wenn sie Freunde beeindrucken wollen oder Vorgesetzte beeinflussen müssen.



Dass man den Algorithmus unabhängig von den Clients variieren lassen kann, bedeutet nichts anderes, als dass man die Aufsätze austauschen kann, ohne die Maschine selbst verändern zu müssen.

6.3 Interfaces und Lambdas

Sehen wir uns die Wahl der Bruchdarstellung noch einmal aus Benutzersicht an. Wir wollen als Benutzer eine neue Darstellung implementieren, die Zähler und Nenner in runde Klammern setzt, z.B. (1)(2) für $\frac{1}{2}$. Dazu müssen wir:

- Eine neue Klasse schreiben:

```
public class KlammerDarstellung implements BruchDarstellung {
  public String getStringDarstellung(int z, int n) {
    return "("+z+")("+n+")";
  }
}
```

- Ein Objekt der neuen Klasse erzeugen und im Bruch setzen:

```
Bruch b = new Bruch(1,2);
BruchDarstellung klammer = new KlammerDarstellung();
b.setDarstellung(klammer);
```

Es gibt mehrere Möglichkeiten, diesen Code zu verkürzen. Die Standard-Methode bis Java 7 benutzt *anonyme innere Klassen*, die in Kapitel 7 erklärt werden. Ab Java 8 gibt es die noch kürzeren, aber nicht ganz so flexiblen *Lambdas*. Diese werden nachfolgend beschrieben.
Das Interface `BruchDarstellung` umfasst genau eine Methode. Solche Interfaces nennt man *Functional Interface* oder seltener auch *Single Abstract Method (SAM) Interface*. Solche Interfaces können für Lambdas genutzt werden. Um zu garantieren, dass ein Interface tatsächlich ein Functional Interface ist, kann man die Annotation `@FunctionalInterface` verwenden:

```
@FunctionalInterface
public interface BruchDarstellung {

  String getStringDarstellung(int z, int n);
}
```

Fügt man jetzt dem Interface eine weitere Methode hinzu, erzeugt der Compiler einen Fehler. Variablen vom Typ eines Functional Interfaces kann man mit einer von mehreren Lambda-Kurzschreibweisen belegen:

- Die folgenden Codezeilen belegen `b1` und `b2` mit je einer Funktion, der zwei `int`-Werte `z` und `n` übergeben werden und die den Ausdruck auf der rechten Seite des Pfeiloperators `->` zurückgibt. Die Typen des Übergabe- und des Rückgabewerts müssen mit dem Interface übereinstimmen.

```
BruchDarstellung b1 = (int z, int n) -> "("+z+")("+n+")";
BruchDarstellung b2 = (int z, int n) -> String.format("(%d)(%d)",z,n);
```

- Da der Typ des Übergabeparameters durch das Interface schon festliegt, kann man ihn im Lambda auch weglassen. Es ist aber sauberer, ihn hinzuschreiben. Bei nur einem Übergabeparameter kann man auch die runden Klammern weglassen.

  ```
  BruchDarstellung b1 = (z, n) -> "("+z+")("+n+")";
  ```

- Man kann vorhandene Funktionen direkt mit dem Funktionsnamen angeben. Zu beachten ist, dass zwischen Klassennamen und Funktionsnamen zwei Doppelpunkte stehen müssen. Die Typen der Übergabe- und Rückgabeparameter müssen übereinstimmen.

  ```
  //Funktion Integer.toString(int x1, int x2) wird aufgerufen.
  BruchDarstellung b1 = Integer::toString;
  ```

- Längere Codestücke können in geschweifte Klammern eingebettet werden. Der Rückgabewert wird mit `return` zurückgegeben.

  ```
  BruchDarstellung b1 = (int z, int n) {
    if (z>=0) {
      return "positiver Bruch";
    } else {
      return "negativer Bruch";
    }
  }
  ```

- Auch nichtstatische Methoden können aufgerufen werden. Das ist in vielen Fällen sehr sinnvoll.

  ```
  BruchDarstellung b = this::kryptoDarstellung;
  ...

  public String kryptoDarstellung(int z, int n) {
    String ret;
    //..wie auch immer das jetzt aussehen mag
    return ret;
  }
  ```

Falls die Zahl der Übergabeparameter von eins abweicht, benutzt man folgende Schreibweisen:

Anzahl der Übergabeparameter	Schreibweise (Beispiel)
0	`() -> 3.1415926`
2	`(int i, String s) -> s.charAt(i)-'0'`
2	`(i,s) -> s.charAt(i)-'0'`

6.4 Benutzung vordefinierter Interfaces

6.4.1 Sortieren von Brüchen

Die Aufgabe, die wir uns in diesem Abschnitt stellen, ist folgende: Wir haben eine ArrayList von Brüchen

```
public ArrayList<Bruch> bruchListe;
```

und wollen diese Liste nach dem Wert der Brüche sortieren. Mit dem Datentyp `ArrayList<String>` hatten wir dafür einfach die Methode

```
Collections.sort(liste);
```

benutzt. Nur leider funktioniert das hier nicht. Eclipse (bzw. der Java-Compiler) gibt uns eine Fehlermeldung namens *Bound mismatch*. Die Ursache dafür ist, dass Java nicht weiß, wie Bruch-Objekte überhaupt zu sortieren sind. Dazu gibt es eine `Collections.sort`-Methode mit zwei Parametern. Der zweite Parameter ist vom Typ `Comparator`. Comparator ist ein Interface mit einer Methode `compare`, die zwei Werte vergleicht und ein Ergebnis zurückgibt, das die folgende Bedingung erfüllt:

$$\text{compare(a,b)} = \begin{cases} \text{Wert größer } 0 & (\text{falls } a > b) \\ 0 & (\text{falls } a = b) \\ \text{Wert kleiner } 0 & (\text{falls } a < b) \end{cases}$$

Diese Vergleichsoperation kann die `sort`-Methode zur Sortierung benutzen. Die Typen von `a` und `b` werden durch Generics bestimmt und müssen mit den zu sortierenden Datentypen übereinstimmen. Für den Inhalt verwenden wir der Kürze halber gleich Lambda-Ausdrücke:

```
Comparator<Bruch, Bruch> comp = (Bruch a, Bruch b) -> {
  double a = a.getDoubleValue();
  double b = b.getDoubleValue();
  if (a>b) {
    return 1;
  } else if (a==b) {
    return 0;
  } else {
    return -1;
  }
};
```

Jetzt können wir mit

```
Collections.sort(liste, comp);
```

die Bruch-Liste sortieren.

Zweites Beispiel: Rechtecke

Nachfolgend ein Beispiel für das Interface Comparator zu einer Klasse `Rechteck`. `compare` vergleicht zwei Rechtecke nach ihrem Flächeninhalt.

```
import java.util.*;

public class Rechteck {
  public double hoehe, breite;

  public Rechteck (double hoehe, double breite) {
    this.hoehe = hoehe;
    this.breite = breite;
  }
}

public static void main(String[] args) {
  ArrayList<Rechteck> liste = new ArrayList<>();
  ....

  Collections.sort(liste, (Rechteck r1, Rechteck r2) ->
      Double.compare(r1.hoehe*r1.breite,r2.hoehe*r2.breite));
  ...
}
```

Die Methode `Double.compare` kann man einfach nutzen, wenn zwei `double`-Werte verglichen werden sollen.

Weitere Interfaces aus der Java-API

- Comparable: Ein Objekt ist mit einem anderen Objekt (der gleichen oder einer anderen Klasse) vergleichbar. Die Methode `compareTo(Object o)` muss implementiert sein. Implementation eines Default-Verhaltens, wenn beim Sortieren der Comparator-Parameter weggelassen wird.
- Cloneable: Ein Objekt kann mit dem Befehl `Object clone()` geklont (bzw. kopiert) werden.
- Serializable: Ein Objekt kann mit einem einzigen Befehl (`ObjectStream.writeObject()`) auf der Festplatte gespeichert bzw. von dort gelesen werden.

Andere Beispiele: Adjustable, Appendable, Callable, Closeable, Destroyable, Flushable, Formattable, Iterable, Joinable, Pageable, Printable, Readable, Referencable, Refreshable, Runnable, Scrollable, Streamable, Transferable, ...

Chapter 7

Vererbung

7.1 Grundlagen

7.1.1 Beispielhafte Problemstellung

Neue Rolle: Der Wiederverwender Nehmen wir an, sie haben den Auftrag, ein mathematisches Paket zu entwickeln. Da sie nicht die Zeit haben, alles von Grund auf neu zu programmieren, versuchen sie, möglichst viele Klassen aus vorherigen Projekten wiederzuverwenden. Damit nehmen sie eine neue Rolle ein: die des *Wiederverwenders*. Also sehen sie sich auch die Klasse `Bruch` aus Kapitel 3 an (die Erweiterungen aus Kapitel 6 ignorieren wir der Einfachheit halber hier). Die könnten sie gut gebrauchen, wenn da nicht ein paar typische Wiederverwendungs-Probleme auftreten würden:

- Die Klasse `Bruch` bräuchte unbedingt noch zwei Methoden `getInt` und `getFrac`, um den ganzzahligen Anteil des Bruchs und den Rest auszugeben.

  Beispiel: Der ganzzahlige Anteil von $\frac{5}{3}$ ist 1, der Rest ist $\frac{2}{3}$.

- Sie haben nicht die Zeit oder die Möglichkeit, den Source-Code der Klasse zu ändern.

- Der Entwickler der Klasse kann ihnen dabei auch nicht helfen. Er muss sich voll auf ein anderes Projekt konzentrieren (freundlichere, aber unrealistische Alternative: Er macht gerade Urlaub in der Karibik).

Sie brauchen also eine Möglichkeit, die Klasse zu erweitern, ohne den Source-Code der Klasse ändern zu müssen. Genau das ist eine der Hauptstärken der objektorientierten Programmierung. Sie können Klassen erweitern, obwohl sie nur die API, aber nicht den Source-Code kennen.

Die Klasse `XBruch`

Wir nennen die erweiterte Klasse `XBruch` für *erweiterten (extended) Bruch*. Dieser Klasse fügen wir die Methoden `getInt()` und `getFrac()`:

```
public class XBruch extends Bruch {

  public int getInt() {
    return getZaehler()/getNenner();
  }

  public int getFrac( {
    return getZaehler()%getNenner();
}
```

Neu ist in der ersten Zeile das Schlüsselwort `extends`. Es bedeutet soviel wie „erweitert“. Die Klasse `XBruch` erweitert die Klasse `Bruch`. Ein Objekt der Klasse `XBruch` besitzt neben den eigenen Attributen und Methoden auch die Attribute und Methoden aus `Bruch`. Sie können angesprochen werden, sofern sie nicht *private* sind. Ehe wir die Konsequenzen daraus betrachten, gehen wir der Klarheit wegen einige Begriffe aus der objektorientierten Programmierung durch.

7.1.2 Terminologie

- Die Klasse *Bruch* heißt die **Basisklasse** (Oberklasse, Superklasse). In den Basisklassen stehen allgemeine Eigenschaften.
- Die Klasse *XBruch* ist die **Unterklasse** (Subklasse, abgeleitete Klasse) von Bruch. In den Unterklassen werden diese Eigenschaften ausgebaut und **spezialisiert**.
- Die Unterklasse wird von der Basisklasse **abgeleitet**.
- Die Unterklasse **erbt** die Eigenschaften (d.h. Methoden und Attribute) der Basisklasse.
- Die Beziehung zwischen Unter- und Basisklasse nennt man **„ist-Beziehung“**. Beispiel: Ein XBruch „ist“ ein Bruch.

7.1.3 Konstruktoren

War es das jetzt schon? Wir testen unsere erweiterte Klasse einmal aus:

```
XBruch v = new XBruch(2,3);
System.out.println(v.getDoubleValue());  //Methode aus Bruch
System.out.println(v.getInt());          //Methode aus XBruch
```

Im Grunde sieht das schon ganz gut aus. Es fehlt aber noch eine Kleinigkeit. Wir können zwar die Methoden aus `Bruch` benutzen, aber nicht die Konstruktoren. Diese werden nämlich prinzipiell nicht vererbt. Wir müssen der Klasse `XBruch` noch selbst einen Konstruktor hinzufügen. Unser erster Versuch ist:

```
public XBruch(int zaehler, int nenner) {
  setZaehler(zaehler);
  setNenner(nenner);
}
```

Mal ganz abgesehen davon, dass die Klasse `XBruch` Probleme beim Kürzen bekäme: Der Java-Compiler akzeptiert diesen Code nicht. Der Grund ist folgender:
Jeder Konstruktor einer Unterklasse ruft automatisch als erstes den Konstruktor seiner Basisklasse auf. Normalerweise ist dies der *parameterlose* Konstruktor der Basisklasse. In der Klasse `Bruch` gibt es aber keinen parameterlosen Konstruktor. Also meldet uns Java einen Fehler. Die Lösung dazu ist, dass wir Java auffordern können, einen anderen Konstruktor als den parameterlosen zu verwenden. Dazu gibt es das Schlüsselwort `super`:

```
public XBruch(int zaehler, int nenner) {
  super(zaehler, nenner);
}
```

Der `super`-Befehl muss im Konstruktor an erster Stelle stehen. Jetzt wird der Basisklassen-Konstruktor mit zwei int-Parametern anstelle des parameterlosen Konstruktors ausgerufen. Als Parameter übergeben wir den Zähler und den Nenner des Bruchs. Damit werden jetzt auch die Attribute richtig gesetzt. Auch der oben gezeigte Testcode funktioniert jetzt.

7.1.4 Lesbarkeit

Die Unterklasse kann auf alle Attribute und Methoden der Oberklasse zugreifen, sofern sie nicht *private* sind, also z.B. auf alle *public*-Methoden.
Zur besseren Differenzierung gibt es einen weiteren Zugriffs-Modifizierer: *protected*. Auf eine protected-Methode oder ein protected-Attribut dürfen zugreifen:

- Objekte der eigenen Klasse.
- Objekte einer Unterklasse.
- Als Besonderheit in Java (im Gegensatz zu C#) Objekte der Klassen des eigenen Packages. Packages dienen zur Abgrenzung gr-o–ss-erer Programmteile. Sie werden allgemein in einem späteren Kapitel eingeführt. Bisher arbeiten wir nur im sogenannten *Default-Package*.

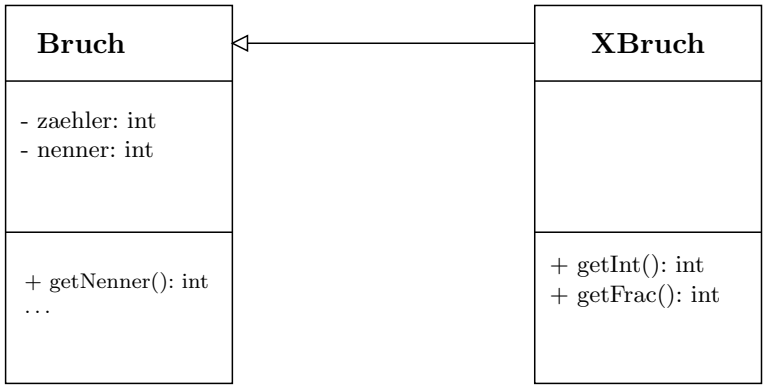
Wenn wir als Entwickler der Klasse `Bruch` dem *Wiederverwender* die Möglichkeit geben wollen, auf die Attribute zuzugreifen, dem *Anwender* das aber verbieten wollen, können wir die beiden Attribute auf `protected` setzen:

```
public class Bruch {
  protected int zaehler;
  protected int nenner;
}
```

Natürlich könnte sich der Anwender als Wiederverwender „tarnen“, indem er die Klasse `Bruch` überschreibt, und dann munter auf die Attribute zugreifen. Deshalb muss man sich als Entwickler gut überlegen, ob man eine solche Möglichkeit zulässt.

7.1.5 UML-Diagramm

Im UML-Diagramm wird die Verebungsbeziehung durch einen Pfeil ausgedrückt, der von der Unterklasse zur Basisklasse zeigt. Er bedeutet: Attribute und Methoden (sofern nicht private) der Basisklasse können mitbenutzt werden. Der Zugangsmodifizierer *protected* wird durch ein Doppelkreuz `#` statt durch ein `+` (public) oder ein `-` (private) kenntlich gemacht.



7.1.6 Einfach- und Mehrfachvererbung

Eine Java-Klasse kann nur eine Basisklasse haben (Einfachvererbung). In manchen Sprachen, wie C++ oder Python gibt es auch *Mehrfachvererbung*, d.h. eine Klasse kann auch mehrere Basisklassen haben. In Java oder C# ist das nicht möglich. Hier besitzt die Klassenhierarchie eine Baumstruktur. Es gibt eine „Wurzel-“-Klasse, die in Java *Object* heißt. Von dieser Wurzel-Klasse werden verschiedene Unterklassen abgeleitet, die wiederum Unterklassen und Unter-Unterklassen besitzen, was sich bis in beliebige Tiefe fortsetzen kann. Jede Klasse hat dabei (bis auf die Wurzelklasse) genau eine (direkte) Basisklasse.

7.1.7 Methoden überschreiben

Wir wollen eine neue Anforderung an unsere XBruch-Klasse stellen. Die `toString`-Methode soll einen String zurückgeben, der die Form “Ganzzahl + Rest“ hat. Beispiel: Der Bruch hat den Wert $\frac{7}{3}$. Dann soll die Ausgabe `2+1/3` sein. Bei $-\frac{7}{3}$ soll die Ausgabe `-2-1/3` sein.
Dabei könnten wir gut das Interface aus Kapitel 6 gebrauchen. Es steht uns aber hier nicht zur Verfügung. Uns steht als Wiederverwerter aber noch ein anderer Weg offen. In Java ist es möglich, eine Methode der Basisklasse durch eine Methode der Unterklasse zu „ersetzen“. Die genaue Bedeutung dieses Begriffs lernen wir später noch kennen. Der Fachausdruck dafür heißt ü*berschreiben*. Dazu implementieren wir die Methode in der Unterklasse einfach noch einmal.

```
public class XBruch extends Bruch {

  @Override
  public String toString() {
```

```
    Bruch frac = getFrac();

    if (frac().getZaehler()>=0) {
      return getInt()+"+"+frac.getZaehler()+"/"+frac.getNenner();
    } else {
      return getInt()+frac.getZaehler()+"/"+frac.getNenner();
    }
  }

}
```

Die Zeile `@Override` ist eine *Annotation*. Man könnte sie auch weglassen, aber sie garantiert, dass `toString` wirklich eine gleichnamige Methode überschreibt und ist damit eine Absicherung gegen Tippfehler im Methodennamen.
Tippfehler an dieser Stelle sind nämlich problematisch. Würde man versehentlich `tostring` schreiben, würde das Programm immer noch laufen, aber der Aufruf von `toString` würde immer noch die alte Methode wählen.

7.1.8 Zugriff auf die überschriebene Methode

Wenn eine Methode überschrieben ist, bedeutet das, dass es ohne weiteres nicht möglich ist, an die Methode der Basisklasse heranzukommen. Um dies dennoch zu erreichen, gibt es in den meisten objektoriertierten Sprachen ein spezielles Schlüsselwort. In Java heißt es `super`. Man kann von der Unterklasse aus mit dem Aufruf *super.methodenname(...)* auf Methoden der Basisklasse zurückgreifen, auch wenn diese überschrieben sind. Als einfaches Beispiel überschreiben wir die Klasse `Bruch` mit einer `toString`-Methode, die vor und hinter den Bruch ein „X“ setzt, also z.B. statt $\frac{5}{3}$ den String $X\frac{5}{3}X$ zurückgibt.

```
public class XBruch extends Bruch {

  public String toString() {
    return "X"+super.toString()+"X";
  }
}
```

7.2 Beispiele aus der API

7.2.1 Object

Die Klasse `Object` ist die Wurzelklasse, von der alle anderen Klassen erben. Mit anderen Worten: Auch wenn in einer Klasse keinerlei Methoden und Attribute implementiert werden, bekommt man automatisch von der Klasse `Object` einige mitgeliefert.
Eine Auswahl davon sind:

- `public Class getClass()`: Liefert ein `Class`-Objekt, das die Klasse des Objekts beschreibt. Damit sind zum Beispiel die Methoden `getName()` oder `getMethods()` möglich.

- `public void finalize()`: Wird automatisch aufgerufen, wenn das Objekt vom Garbage-Collector gelöscht wird. Die Methode in `Object` ist leer. Damit etwas passieren soll, muss sie überschrieben werden.
- `public int hashCode()`: Liefert eine eindeutige ID für jedes Objekt. Die Beschreibung in der Java-API lautet:

  As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)
- `public String toString()`: Liefert eine String-Darstellung des Objekts. Wird automatisch aufgerufen, falls das Objekt mit `System.out.println()` ausgegeben wird. Da diese Methode sehr lehrreich ist, werden wir im nächsten Kapitel ausführlicher auf sie eingehen.

Beispiel:

```
public class Minimal {}

public class Test {

  public static void main(String[] args) {
    Minimal m = new Minimal();
    System.out.println(m.getClass().getName());
    System.out.println(m.hashCode());
    System.out.println(m.toString());
  }
}
```

7.2.2 Die toString()-Methode

Wenn ein Objekt mit `System.out.println()` ausgegeben wird, dann wandelt Java das Objekt zunächst mit `toString()` in eine Stringdarstellung um. Weil `toString()` bereits in der Klasse `Object` vorhanden ist, hat jedes Objekt eine `toString()`-Methode. Diese Methode dient als Default und sieht in der Sun-Java Klassenbibliothek so aus:

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

So ergibt die Zeile

```
System.out.println(System.out);
```

dieses Ergebnis (die Zahl am Ende kann von Durchlauf zu Durchlauf variieren)

```
java.io.PrintStream@10b62c9
```

Wenn wir die `toString()`-Methode einer Klasse implementieren, dann ü*berschreiben* wir die `toString()`-Methode von *Object* und es wird statt dessen unsere `toString()`-Methode aufgerufen. In manchen Klassen überschreibt Java die `toString()`-Methode auch selbst. Darum werden z.B. bei Exceptions, bei StringBuilder- oder bei File-Objekten andere, lesbarere Darstellungen ausgegeben.

7.3 Bindungsarten

7.3.1 Statischer und dynamischer Typ

In Java ist die folgende Variablendeklaration möglich:

```
Bruch b = new XBruch(1,2);
```

Der Variablen vom Typ `Bruch` wird ein Objekt vom Typ `XBruch` zugewiesen. Dies ist möglich, weil `XBruch` eine Unterklasse von `Bruch` ist.
Bevor wir über die Konsequenzen daraus nachdenken, müssen wir uns etwas mit der korrekten Terminologie beschäftigen. In unserem Beispiel nennt man `Bruch` den *statischen Typ* der Variablen `b`. Dieser Typ ist, nachdem er einmal festgelegt wurde, nicht wieder veränderbar. Der Typ des Objekts, also `XBruch` heißt *dynamischer Typ* oder Laufzeittyp von `b`. Dieser Typ kann sich während des Programms ändern. Zum Beispiel kann man mit der Zeile

```
b = new Bruch(1,5);
```

den dynamischen Typ von `b` in `Bruch` ändern. Der Name *Laufzeittyp* kommt daher, dass zur Compile-Zeit dieser Typ nicht bekannt ist. Er könnte es theoretisch sein, aber dazu müsste der Compiler das gesamte Programm analysieren, was vom Aufwand her nicht zu schaffen ist. Dagegen ist der statische Typ dem Compiler bekannt. Daher merkt er, wenn man zum Beispiel mit

```
b = b.getSquare();
```

eine Methode aufrufen will, die gar nicht exisitiert.

Überprüfung des dynamischen Typs

Es gibt mehrere Möglichkeiten, den dynamischen Typ zu überprüfen. Der Ausdruck

```
br instanceof XBruch
```

gibt *true* zurück, falls der dynamische Typ von *br* die Klasse `XBruch` ist und anderenfalls *false*. Der Ausdruck

```
br instanceof Bruch
```

gibt *true* zurück, falls der dynamische Typ von *br* `Bruch` *oder* `XBruch` ist. Der Operator *instanceof* prüft ob die angegebene Variable vom Typ der angegebenen Klasse *oder einer ihrer Unterklassen* ist. Eingesetzt wird der Operator gewöhnlich, um sicherzustellen, dass ein Cast keine Exception verursacht:

```
if (br instanceof XBruch) {
  XBruch xbr = (XBruch) br;
  //Sonderbehandlung fuer XBrueche
}
```

Seit Java 16 lässt sich Überprüfung und Cast kompakt in einer Zeile zusammenfassen:

```
if (br instanceof XBruch xbr) {
  //Hier kann die Variable xbr verwendet werden
}
```

Es gibt auch noch eine ganz andere Methode zum Überprüfen des dynamischen Typs. Der Aufruf:

```
getClass().getName()
```

liefert für jedes Objekt eine String-Darstellung des dynamischen Typs. Zum Beispiel ist die Variable *System.in* vom statischen Typ *InputStream* (was man mit Eclipse leicht feststellen kann). Die String-Darstellung liefert aber den dynamischen Typ *BufferedInputStream*, der von InputStream abgeleitet ist (je nach Java-Version kann auch ein anderer Typ zurückgeliefert werden).

7.3.2 Überschreiben von Elementen

Die Frage, die sich uns jetzt stellt, ist: Was passiert bei den Aufrufen

```
Bruch k = new XBruch(5,3);
        //Statischer Typ:  Bruch
        //Dynamischer Typ: XBruch

int x = k.getInt();
String s = k.toString();
```

Für die Zeile `k.getInt()` gibt es prinzipiell zwei mögliche Antworten:

1. Es könnte die Methode der statischen Klasse (Bruch) ausgeführt werden. Dann gäbe es einen Fehler, denn diese Methode existiert gar nicht.

2. Es könnte aber auch die Methode der dynamischen Klasse (XBruch) ausgeführt werden. Dann würde `x` den Wert 1 erhalten.

Ähnlich verhält es sich mit der zweiten Zeile `k.toString()`:

1. Es könnte die Methode der statischen Klasse (Bruch) ausgeführt werden. Dann würde `s` der String `5/3` zugewiesen.

2. Es könnte aber auch die Methode der dynamischen Klasse (XBruch) ausgeführt werden. Dann würde `s` der String `1+2/3` zugewiesen.

Die Antwort darauf ist zum Verständnis der Objektorientierung wichtig. Sie heißt:

In Java bestimmt die statische Klasse, *welche* Methoden aufgerufen werden können und die dynamische Klasse, *wie* diese Methoden implementiert sind.

Anders ausgedrückt:

- Ist die Methode nur in der dynamischen Klasse vorhanden, aber nicht in der statischen Klasse, gibt es einen Compiler-Fehler (bzw. Eclipse meldet einen Fehler).
- Ansonsten wird *immer* die Methode der dynamischen Klasse aufgerufen.

Bei der Zeile `k.getInt()` ist also Antwort 1 richtig, bei der Zeile `k.toString()` Antwort 2.
Wir werden die Grundlagen dazu im nächsten Abschnitt näher betrachten.

7.3.3 Bindungsarten

Wenn beim Abarbeiten eines Java-Programms das Java-Laufzeit-System auf einen Methodenaufruf triff, geht es wie folgt vor:

1. Es sieht nach, welchen dynamischen Typ die Variable hat und
2. ruft die Methode des dynamischen Typs auf.

Der Fachausdruck dafür heißt **dynamische Bindung** (späte Bindung, Laufzeitbindung). Das Gegenstück dazu heißt **statische Bindung** (frühe Bindung), bei der die Methode des statischen Typs ausgerufen wird. Dieser Typ und die dazugehörige Methoden werden bereits vom Compiler ermittelt. Da so der Typ während der Laufzeit schon bekannt ist, geht dieser Methodenaufruf schneller.

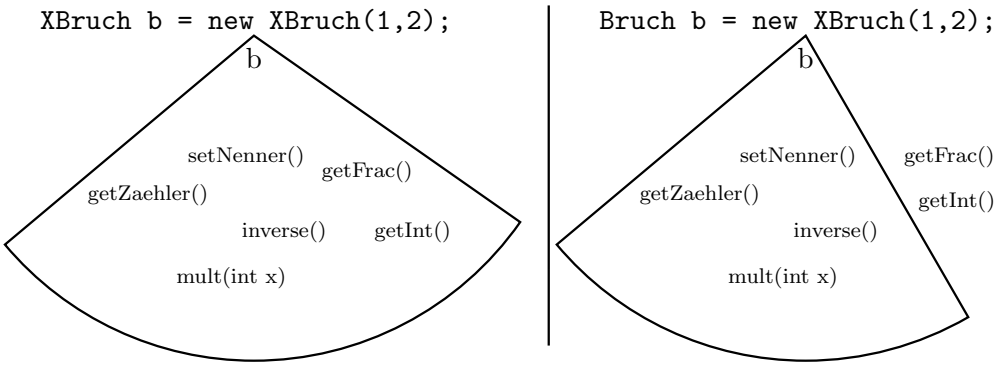
	statische Bindung	dynamische Bindung
Compiler	Ermittelt Methode des statischen Typs.	-
Laufzeitsystem	Ruft ermittelte Methode auf.	Ermittelt dynamischen Typ und ruft die Methode des dynamischen Typs auf.
Schnelligkeit	hoch	niedrig

7.3.4 Vorteile der dynamischen Bindung

Wenn die statische Bindung nun schneller ist, warum bindet Java dann Methoden im Regelfall dynamisch? Nehmen wir dazu noch einmal unsere Bruch-Klasse:

```
Bruch br = new XBruch(1,2);
```

Durch die dynamische Bindung verhält sich das `XBruch`-Objekt stets und überall wie ein `XBruch`-Objekt. Das heißt, die `toString`-Methode liefert immer den gemischten Bruch als Ergebnis. Wenn ein Objekt sein Verhalten ändern würde, sobald sich der statische Typ der Variablen ändert, würde das Verhalten weitaus komplizierter werden.
Das Einzige, was sich bei der Zuweisung zum Typ `Bruch` ändert, ist, dass der Blickwinkel auf das Objekt eingeschränkt wird. Man sieht nämlich nur noch die Methoden, die in `Bruch` definiert sind.



Typumwandlungen

Nach der Zeile

```
Bruch br = new XBruch()
```

wissen wir, dass die Variable *br* den dynamischen Typ `XBruch` hat. Wenn man das Objekt wieder einer Variable vom Typ `XBruch` zuweisen will, benötigt man einen expliziten Cast. Das heißt:

```
XBruch xb = new XBruch(1,2);
Bruch b = new Bruch(1,2);

b = xb;              //XBruch -> Bruch: Geht ohne expliziten Cast
xb = (XBruch) b;     //Bruch -> XBruch: Braucht expliziten Cast
```

Der Unterschied zwischen beiden Fällen ist: Ein XBruch ist immer auch ein Bruch, aber ein Bruch ist noch lange nicht automatisch ein XBruch. Der Compiler kann nicht von selbst ermitteln, welchen dynamischen Typ *b* hat. Es wäre theoretisch möglich, aber die Möglichkeiten des Compilers reichen dazu nicht aus. Daher verlangt der Compiler einen expliziten Cast.
Was passiert nun in folgendem Fall?

```
Bruch br = new Bruch();
XBruch xbr = (XBruch) br;
```

Es wird versucht, *br* in eine Variable vom statischen Typ XBruch zu casten, obwohl *br* gar kein XBruch ist. Das Ergebnis ist eine *ClassCastException*.

7.3.5 Speichern in Feldern und Listen

Als Nächstes wollen wir Objekte der Klassen `Bruch` und `XBruch` gemeinsam in einer `ArrayList` speichern.

```
ArrayList<Bruch> reihe;
```

Für Objekte vom Typ `Bruch` geht das wie gewohnt. Objekte vom statischen Typ `XBruch` können ebenfalls problemlos eingefügt werden:

```
XBruch v = new XBruch(3,7);
reihe.add(v);
```

Holt man Elemente aus der Liste heraus, so haben die Elemente den statischen Typ *Bruch*. Handelt es sich um einen XBruch, kann man ihn mit einem expliziten Cast umwandeln.

```
Bruch s = reihe.get(0);2
XBruch xs = (XBruch) s;
```

Statt einer Liste kann man auch ein Feld

```
Bruch[] feld;
```

verwenden. Das Prinzip bleibt das gleiche.

Felder und Listen vom Typ Object

Noch flexibler sind Listen vom Typ `Object`:

```
ArrayList<Object> liste1 = new ArrayList<Object>();
ArrayList liste2 = new ArrayList();
```

Die zweite Form ist dabei eine Kurzform für die erste. In solchen Listen kann man alle Java-Objekte speichern, unabhängig davon, welche Klasse sie haben. Beim Zugriff auf die Elemente muss man sich allerdings im Klaren darüber sein, auf welches Objekt man zugreift.

```
liste1.add("Hallo");
liste1.add(new Bruch(1,2));

Object ob1 = liste1.get(0);
Object ob2 = liste1.get(1);
String s = (String)ob1;
Bruch st = (Bruch)ob2;
```

7.3.6 Wann wird statisch bzw. dynamisch gebunden?

Wenn wir uns etwas von der Programmiersprache Java lösen, kann man prinzipiell für jede Methode (und auch für jedes Attribut) frei zwischen statischer und dynamischer Bindung wählen. Allerdings:

- Vom Ü**berschreiben** einer Methode spricht man nur, wenn die Methode dynamisch gebunden ist.
- Wenn die Methode statisch gebunden ist, wird sie von einer Unterklasse nicht überschrieben, sondern **verdeckt** (geht in Java nicht).

Zur Übersicht:

	Methode überschreiben	Methode verdecken
Bindungsart	dynamisch	statisch
Aufruf	Methode des dynamischen Typs	Methode des statischen Typs

Für Attribute gilt generell das Gleiche wie für Methoden. Attribute können dynamisch oder statisch gebunden sein. Je nach Bindung können Attribute überschrieben oder verdeckt werden. Das Überschreiben bzw. Verdecken ist für Attribute aber nicht so wichtig wie bei Methoden.

7.3.7 Zusammenfassung: Überschreiben und Verdecken

Java

- Java bindet Methoden generell *dynamisch*. Methoden werden ü*berschrieben* und bei der Ausführung wird die Methode des dynamischen Typs ausgewählt.
- Mit dem Schlüsselwort *final* kann man Java anweisen, eine Methode statisch zu binden:

  ```
  public final double getName() {...}
  ```

  Eine solche Methode kann aber in Java nicht verdeckt (und natürlich auch nicht überschrieben) werden. Dieser Zusatzeffekt gibt dem Schlüsselwort *final* (endgültig) seinen Namen. Die Vorteile von *final* sind also:

  - Die Funktionalität der Methode kann durch Überschreiben oder Verdecken nicht verändert werden.
  - Durch die statische Bindung wird der Methodenaufruf schneller.
- Attribute werden in Java generell *statisch* gebunden. Sie können *verdeckt* werden und bei der Ausführung wird das Attribut der statischen Klasse verwendet. Die Kombination von überschriebenen Methoden und verdeckten Variablen kann dazu führen, dass man nur schwer die Übersicht behält, welche Variablen denn nun tatsächlich verwendet werden. Da es kaum nötig ist, Attribute zu verdecken, rate ich stark, folgende Regel einzuhalten:

Finger weg vom Verdecken von Attributen.

- Statische Methoden werden ebenfalls statisch gebunden und verdeckt. Das Problem ist aber recht gering, solange man statische Methoden auch statisch aufruft (d.h. mit *Klassenname.Methodenname(..)*).

Andere Programmiersprachen

Gerade hier unterscheidet sich das Konzept von Sprache zu Sprache stark, so dass eine Kenntnis der Grundlagen sehr wichtig ist. Es gibt grob 3 Gruppen. In die Java-Gruppe fallen beispielsweise Swift oder Kotlin. Die zweite Gruppe mit C++ und C# verhält sich teilweise entgegengesetzt dazu. Die dritte Gruppe besteht aus Skriptsprachen wie Python. Hier gibt es keinen statischen Typ und damit auch keine Unterschiede zwischen überschreiben und verdecken. In Python wird oft der Ausdruck „ersetzen“ (redefine, replace) statt „überschreiben“ (override) oder „verdecken“ (hide) benutzt.

7.4 Anonyme innere Klassen

Wie wir in den letzten Abschnitten gesehen haben, können wir in einer Unterklasse

- Methoden der Basisklasse überschreiben und
- der Basisklasse weitere Methoden und Attribute hinzufügen.

Wenn wir uns auf das Überschreiben beschränken, können wir auch eine sehr kompakte Schreibweise benutzen: Die *anonyme innere Klasse*. Als erstes Beispiel erzeugen wir eine anonyme innere Klasse, die von `Bruch` abgeleitet ist und die `setZaehler`-Methode überschreibt, so dass der Zähler nicht auf 0 gesetzt werden kann:

```
Bruch b = new Bruch(1,2) {
  public void setZaehler(int zaehler) {
    if (zaehler!=0) {
      super.setZaehler(zaehler);
    } else {
      throw new ArithmeticException("Zaehler auf 0 gesetzt");
    }
  }
}
```

Die entscheidende Zeile ist die erste Zeile des Codes. Diese endet im Gegensatz zu einem normalen Konstruktoraufruf mit dem Zeichen { anstatt eines Semikolons. Das bedeutet folgendes: Die Variable b mit dem statischen Typ Bruch zeigt auf ein Objekt einer *anonymen Unterklasse* von Bruch. Anschließend kann man alle gewünschten Methoden überschreiben, bevor die Erzeugung mit einem } abgeschlossen wird. `b` verhält sich jetzt wie ein ganz normaler Bruch, nur `b.setZaehler` ruft die überschriebene Methode auf.

Anonyme Klassen als statischer Typ

Der anonymen Unterklasse kann man auch Methoden hinzufügen, die nicht in der Basisklasse enthalten sind:

```
Bruch b = new Bruch(1,2) {
  public void neueMethode() {
    System.out.println("Neue Methode");
  }
}
```

Da der statische Typ von `b` aber `Bruch` ist, ist der Aufruf von `b.neueMethode()` nicht möglich. Dazu braucht man die anonyme Unterklasse als statischen Typ, also so etwas, wie

```
AnonymeUnterklasseVonBruch b = new Bruch(1,2) {
```

Das ist erst seit Java 10 mit der `var`-Deklaration möglich:

```
var b = new Bruch(1,2) {
  public void neueMethode() {
    System.out.println("Neue Methode");
  }
}
b.neueMethode();
```

Bemerkenswert ist außerdem:

- Die anonyme Klasse wird *innerhalb* einer weiteren Klasse erzeugt. Daher heißen diese Klassen *anonyme innere Klassen*.[1]
- Anonyme innere Klassen haben Zugriff auf alle Methoden und Attribute der äußeren Klasse. Dabei sind auch private Methoden und Attribute eingeschlossen, was eine weitere Besonderheit anonymer innerer Klassen ist. Darauf wird noch ausführlich eingegangen.
- Man kann in anonymen inneren Klassen keine eigenen Konstruktoren definieren. Wie soll das auch gehen, wenn es keinen Klassennamen gibt? Hier greift eine Sonderregelung: Anonyme innere Klassen benutzen die Konstruktoren der Basisklasse, im obigen Beispiel also die der Klasse `Bruch`.
- Anonyme innere Klassen können auch von einem Interface abgeleitet sein. Das ist sogar recht häufig zu finden und wird im folgenden Beispiel gezeigt. Eine anonyme innere Klasse, die von einem Interface erbt, erhält automatisch den Default-Konstruktor.

Funktionsparameter als anonyme innere Klassen

Im Kapitel über Interfaces hatten wir die `toString` -Darstellung von Brüchen austauschbar gemacht. Wir hatten dazu *Lambdas* verwendet und mit ihnen sehr elegant Funktionszeiger realisiert. Die Alternative hinzu sind anonyme innere Klassen, die aus zwei Gründen wichtig sind:

[1] Alle anonymen Klassen sind in Java innere anonyme Klassen. Andererseits gibt es in Java drei weitere Arten von inneren Klassen, von denen wir in dieser Vorlesung nur die *statische innere Klasse* kennenlernen werden.

- Bis einschließlich Java 7 waren Lambdas noch nicht möglich. Entsprechend häufig sind anonyme innere Klassen noch in Java-Codes zu finden.
- Interfaces mit mehreren Methoden sind für Lambdas nicht geeignet. Anonyme innere Klassen funktionieren aber auch hier.

Die Bruch-Darstellung mit Hilfe einer anonymen inneren Klasse sieht wie folgt aus:

```
BruchDarstellung b1 = new BruchDarstellung() {
  public String getStringDarstellung(int z, int n) {
    return z+"/"+n;
  }
}

//Zum Vergleich: Lambda

BruchDarstellung b2 = (int z, int n) -> z+"/"+n;
```

Zugriff auf äußere Klasse

Wir betrachten den folgenden Code, in der in der Methode `makeAnonym()` der Klasse `Aussen` eine anonyme innere Klasse erzeugt wird, die vom Interface `TestInterface` erbt.

```
public interface TestInterface {
  public void start();
}
public class Aussen {

  TestInterface innen;

  private void aussenMethode() {
    System.out.println("Testmethode");
  }

  public void makeAnonym() {
    //Erzeugen der inneren Klasse
    innen = new TestInterface() {
      public void start() {
        //Kann auf Attribute und Methoden der
        //aeusseren Klasse zugreifen
        aussenMethode();
        //this.aussenMethode(); funktioniert nicht!
        Aussen.this.aussenMethode();  //funktioniert
      }
    };
  }
}
```

Die Besonderheit ist, dass aus der `start`-Methode der inneren Klasse auch auf die Attribute und Methoden des umgebenden äußeren Objekts zugegriffen werden kann, selbst wenn diese `private` sind.
Die Klasse `Aussen` erzeugt in der Methode `makeAnonym` den Funktionszeiger `innen` als anonyme innere Klasse. Von dort wird die Methode `aussenMethode` der äußeren Klasse aufgerufen. Java sucht zunächst in der inneren Klasse nach der Methode `aussenMethode` und, wenn es sie dort nicht gibt, anschließend in der äußeren.
Beim Ausdruck `this.aussenMethode` bezieht sich das `this` auf die innere Klasse und funktioniert daher nicht. Will man sich explizit auf die äußere Klasse beziehen, muss man

```
Aussen.this.aussenMethode()
```

schreiben, d.h. den Klassennamen der äußeren Klasse vor das `this` setzen.
Lambdas sind zwar auch eine Art innerer Klasse, hier bezieht sich `this` jedoch immer auf die äußere Klasse.

7.5 Abstrakte Klassen

Wir kommen ein weiteres Mal auf unsere Klasse *Bruch* zurück. In den letzten beiden Kapiteln haben wir zwei Möglichkeiten gelernt, die String-Ausgabe eines Bruches variabel zu gestalten:

1. Mit dem Entwurfsmuster *Strategie* und einem Interface.
2. Mit einer Unterklasse von `Bruch` und dem Überschreiben der `toString`-Methode.

Die zweite Möglichkeit war bisher eher ein Nebeneffekt der Objektorientierung und in erster Linie für den Wiederverwender gedacht. Der Entwickler der Klasse kann sie aber auch sehr bewusst für den Anwender konzipieren und damit wesentlich näher an das Entwurfsmuster *Strategie* rücken.
Er könnte sich zum Beispiel folgendes überlegen: Die `toString`-Methode der Klasse `Bruch` ist eine Default-Methode und gleichzeitig ein sogenannter *Hook*, also ein Haken, an dem sich der Anwender mit seiner eigenen Methode einhängen kann, indem er eine Unterklasse bildet. Der Schritt liegt nahe, die Default-Methode wegzulassen und den Anwender zu *zwingen*, eine Unterklasse zu bilden. Damit rückt man noch ein Stück näher an das Entwurfsmuster Strategie heran. Folgende Änderungen sind nötig: Man entfernt die Implementation der `toString`-Methode und ersetzt sie durch eine Methodendefinition ähnlich wie in einem Interface.

```
public abstract class Bruch {

  public abstract String toString();

  //...weitere Methoden
}
```

Neu ist das Schlüsselwort `abstract`. Es taucht zweimal auf. Bei der Klassendefinition bedeutet es, dass es nicht erlaubt ist, ein Objekt dieser Klasse zu erzeugen. Wohl aber kann man die Klasse überschreiben, muss dann aber die als abstrakt deklarierten Methoden (in unserem Beispiel `toString`) implementieren.[2] Das heißt, wir können den Aufsatz in einer Unterklasse hinzufügen:

```
public class DoubleAusgabeBruch extends Bruch {

  public String toString() {
    return ""+getDoubleWert();
  }
}
```

Der Unterschied zum Entwurfsmuster *Strategie* ist allerdings: Der Aufsatz lässt sich nachträglich nicht mehr austauschen. Er bildet mit der Maschine eine Einheit, die der Anwender nicht mehr zertrennen kann.

Eigenschaften von abstrakten Klassen

Abstrakte Klassen und Methoden haben die folgenden speziellen Eigenschaften:

- Abstrakte Klassen können nicht instanziiert werden (es können keine Objekte von abstrakten Klassen erzeugt werden).

- Abstrakte Methoden müssen überschrieben werden, damit sie ausgeführt werden können. Von abstrakten Methoden wird nur der Methodenkopf angegeben. Dem Methodenkopf folgt statt der geschweiften Klammern ein Semikolon. Der Methodenrumpf (-inhalt) fehlt.

- Nur abstrakte Klassen können abstrakte Methoden besitzen. In den Unterklassen **muss** eine abstrakte Methode überschrieben werden (es sei, denn, die Unterklassen sind ebenfalls abstrakt).

Abstrakte Klassen können neben abstrakten Methoden auch Attribute und „normale“ Methoden enthalten. Alle drei folgenden Fälle sind möglich:

- Es gibt abstrakte Klassen, bei denen alle Methoden abstrakt sind.

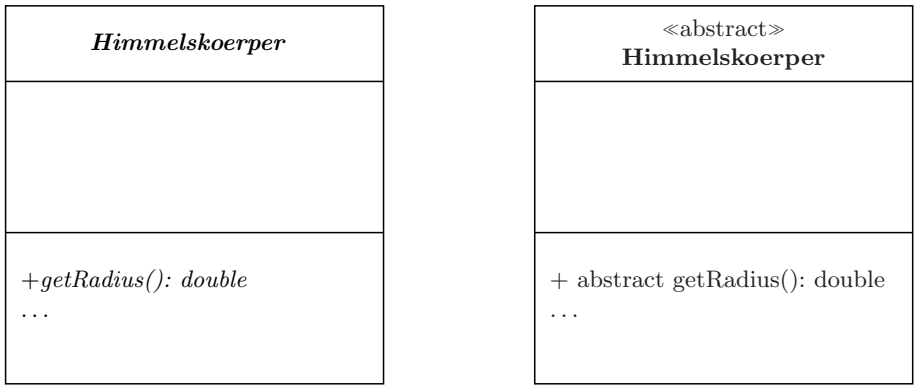
- Andere abstrakte Klassen (und das ist der Hauptteil) haben sowohl abstrakte als auch normale Methoden. Nur die abstrakten Methoden müssen überschrieben werden.

- Es gibt aber auch abstrakte Klassen ohne abstrakte Methoden. In diesem Fall ist zwar die Klasse als abstrakt deklariert, aber keine Methode. Diese Klasse kann nicht instanziiert werden, man muss beim Ableiten aber keine Methode überschreiben.

[2] Man kann alternativ auch die Unterklasse wieder abstrakt machen.

UML-Diagramm

Im UML-Diagramm gibt es zwei Schreibweisen. In der ersten wird der Klassenname kursiv geschrieben, zum Zeichen dafür, dass die Klasse abstrakt ist. Dies ist speziell bei handschriftlichen Diagrammen unpraktikabel. Daher gibt es noch die Möglichkeit, das Wort ≪abstract≫ vor den Klassennamen zu setzen. Auch abstrakte Methoden werden kursiv geschrieben. Ich rate dazu, bei handgeschriebenen UML-Diagrammen das Schlüsselwort *abstract* vor den Methodennamen zu setzen. Dies entspricht zwar nicht dem Standard, vermeidet aber Verwechslungen durch schlechte Handschrift.



Insgesamt werden abstrakte Klassen deutlich seltener eingesetzt, als die verwandten Interfaces. Interfaces haben den großen Vorteil, dass eine Klasse mehrere Interfaces implementieren *und* zusätzlich eine Basisklasse haben kann. Dagegen kann eine Klasse, die eine abstrakte Basisklasse hat, von keiner weiteren Klasse mehr erben.

Chapter 8

Rekursive Algorithmen

8.1 Einleitung

Ein Algorithmus heißt rekursiv, wenn es sich selbst als Teil enthält oder mithilfe von sich selbst definiert ist. Ein mathematischer rekursiver Zusammenhang ist z.B. eine mögliche Definition der Fakultät:

$$n! = \begin{cases} 1 & ; \quad n = 0 \\ n \cdot (n-1)!; & n > 0 \end{cases}$$

Solch eine Definition kann in einem Java-Programm umgesetzt werden. Dabei ruft eine Funktion sich selbst wieder auf. Wir werden das in 2 Schritten umsetzen: Erstens

$$fakultaet(n) = \begin{cases} 1 & ; \quad n = 0 \\ n * fakultaet(n-1); & n > 0 \end{cases}$$

Zweitens:

```
public long fakultaet(int n) {
  if (n==0) {
    return 1;
  } else {
    return n*fakultaet(n-1);
  }
}
```

Wichtig bei der Rekursion ist die *Abbruchbedingung*. Man muss dafür sorgen, dass die Rekursionsaufrufe nicht bis in alle Ewigkeit (d.h. bis der Speicher überläuft) fortgesetzt werden, sondern irgendwann abbrechen.
Wenn eine Funktion sich selbst direkt rekursiv aufruft, nennt man das *direkte Rekursion*. Wenn eine Funktion a eine weitere Funktion b aufruft, die wiederum a aufruft, ist das ebenfalls eine Rekursion, die man *indirekte Rekursion* nennt. Wenn eine Methode eines Objekts dieselbe Methode eines anderen Objekts der gleichen Klasse aufruft, ist das ebenfalls eine Rekursion.

8.2 Interne Umsetzung im Stack

Zum Verständnis der internen Abläufe bei einem Rekursionsaufruf erinnern wir uns zunächst an die Art, wie Java Daten speichert (in anderen Programmierspachen ist es ähnlich):

- Programmcode und statische Variablen werden in der „Method Area“ gespeichert.
- Attribute eines Objekts werden auf dem „Heap“ gespeichert.
- Lokale Variablen (die in einer Methode definiert wurden) und Übergabeparameter werden auf dem „Stack“ gespeichert.

Bei der „klassischen“ Rekursion wird nur mit lokalen Variablen und Übergabeparametern gearbeitet. Uns interessiert daher von den drei Bereichen nur der Stack. *Stack* bedeutet im Deutschen *Stapel*. Gestapelt werden sogenannte *Stack-Frames*. In einem Stack-Frame stehen alle in einer Methode benutzten lokalen Variablen, wobei gilt:

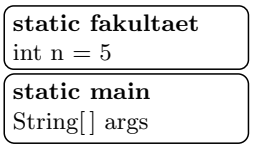
- Bei primitiven Datentypen stehen die Werte der Variablen im Stack-Frame.
- Bei Objekten steht nur ein Verweis (die Adresse) des Objekts im Stack-Frame. Das Objekt selbst steht im Heap.[1]

Jedesmal, wenn eine neue Untermethode aufgerufen wird, wird der Stapel um ein Frame erhöht. Immer, wenn eine Methode beendet wird, wird der entsprechende Frame vom Stapel genommen. Mit diesem Wissen können wir schon das Beispiel `Fakultaet` aus dem letzten Abschnitt analysieren. Der vollständige Code lautet:

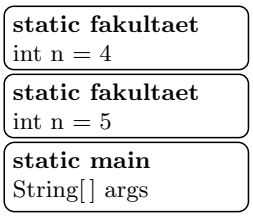
```
public class Fakultaet {

  public static long fakultaet(int n) {
    if (n==0) {
      return 1;
    } else {
      long m = n * fakultaet(n-1);
      return m;
    }
  }

  public static void main(String[] args) {
    long f = fakultaet(5);
    System.out.println(f);
 }

}
```

[1] Wir werden, um die Beispiele einfach zu halten, zunächst nur primitive Datentypen verwenden

Am Beginn der Programms wird ein Stack-Frame *main* auf den Stack gelegt. In diesem Frame ist zu Beginn nur die Variable *args* abgelegt. In der ersten Zeile der main-Methode wird die Methode `fakultaet` aufgerufen. Damit wird ein neuer Frame angelegt, der auf den Stapel wandert und dort den Frame *main* überdeckt. Der neue Frame besitzt zu Beginn nur eine Variable, nämlich den Übergabeparameter `n`.



In der `fakultaet`-Methode wird, wenn $n > 0$ ist, `fakultaet(n-1)` aufgerufen und damit ein weiteres Stack-Frame angelegt.



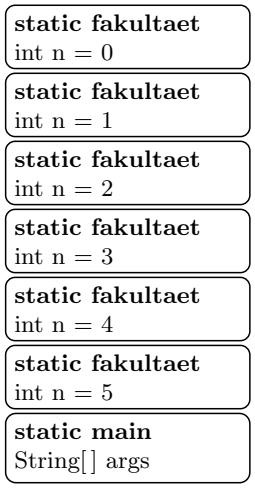
Ohne Abbruchbedingung würden jetzt immer weiter Stack-Frames aufeinandergestapelt, bis der Speicher für den Stack überläuft. Anschließend würde eine Fehlermeldung wie

```
Exception in thread "main" java.lang.StackOverflowError
at Fakultaet.fakultaet(Fakultaet.java:7)
at Fakultaet.fakultaet(Fakultaet.java:7)
at Fakultaet.fakultaet(Fakultaet.java:7)
at Fakultaet.fakultaet(Fakultaet.java:7)
    ...
```

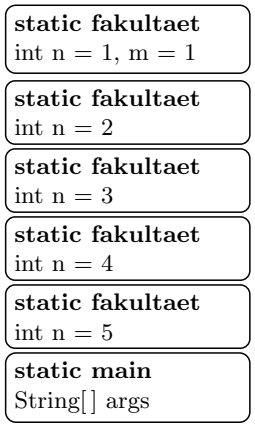
ausgegeben werden, wobei zahlreiche weitere gleiche Zeilen folgen. Jede Zeile entspricht einem verschachtelten Methodenaufruf.[2] Die Meldung bedeutet, dass der Speicher für die Stack-Frames übergelaufen ist.

Daher ist die Abbruchbedingung wichtig. In unserem Beispiel wird sie erreicht, wenn $n = 0$ ist. Der Stack sieht jetzt folgendermaßen aus:

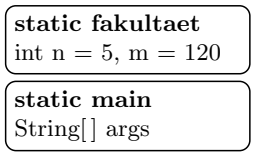
[2]Es werden nur die letzten 1024 Zeilen ausgegeben.



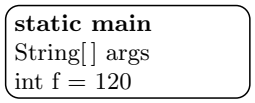
Da $n = 0$ ist, wird der Wert 1 zurückgegeben. Das oberste Stack-Frame wird aufgelöst und das darunterliegende Frame kommt zum Vorschein. Dort wird der Rückgabewert mit der Variablen n multipliziert und in einer neu erzeugen Variable m gespeichert.



Die Variable m wird aus der Methode zurückgegeben und in der darunterliegenden Methode mit n aus dem entsprechenden Stack-Frame multipliziert. Dies setzt sich fort, bis die oberste `fakultaet`-Methode erreicht ist:



Dieser Wert wird an die `main`-Methode zurückgegeben. Im zugehörigen Stack-Frame wird eine neue Variable `f` angelegt, in der der Rückgabewert gespeichert wird.



8.3 Verwendung von rekursiven Algorithmen

8.3.1 Rekursive und iterative Algrorithmen

Den rekursiven Algorithmen stehen die nicht-rekursiven oder iterativen Algorithmen gegenüber, die Schleifen verwenden. Es gilt:

1. Jeder iterative Algorithmus lässt sich auch rekursiv schreiben. Die iterative Losung ist meistens klarer und schneller programmiert. Allerdings ist das zum Teil auch Geschmackssache. Andersherum lassen sich rekursive Algorithmen leicht iterativ schreiben, falls in der Methode nur ein einziger Rekursionsaufruf erfolgt.

2. Gibt es mehrere Rekursionsaufrufe in einer Methode, kann ein entsprechendes iteratives Programm deutlich länger und komplizierter werden.

Zu (1): Es ist möglich, jeden Algorithmus ganz ohne Schleifen und lediglich mit Rekursionsaufrufen zu schreiben. Die *funktionalen Programmiersprachen* besitzen keine Schleifenkonstrukte und benutzen statt dessen Rekursionsaufrufe. Die bekanntesten funktionalen Sprachen sind Lisp und Haskell.

Das folgende einfache Beispiel gibt die Zahlen von 1 bis 10 auf dem Bildschirm aus. Die bekannte iterative Lösung ist:

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

Die rekursive Lösung basiert auf einer Methode

```
printZahl(int start, int end)
```

die alle Zahlen von `start` bis `end` auf dem Schirm ausgibt. Die Funktionsweise lässt sich als Formel so beschreiben:

$$\text{printZahl(int start, int end)} = \begin{cases} start == end\text{: Gib}start\text{aus} \\ start < end\text{: Gib}start\text{aus} \\ \qquad\qquad\qquad \text{printZahl(start+1,end)} \end{cases}$$

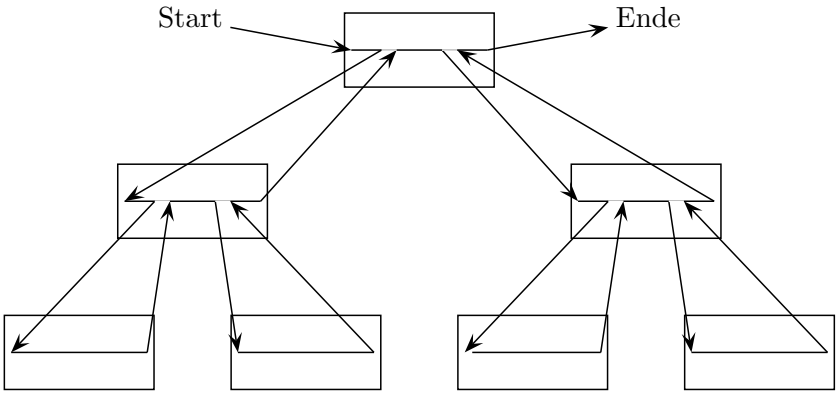
Das ergibt als Java-Code:

```
public void printZahl(int start, int end) {
  System.out.println(start);
  if (start<end) {
    printZahl(start+1, end);
  }
}
```

Die rekursive Version ist schwerer zu lesen und außerdem langsamer als die iterative Version, denn für jeden Rekursions-Durchlauf muss ein neues Stack-Frame angelegt werden. Auch das Programm zur Fakultäts-Berechnung ist in der folgenden iterativen Version kompakter und schneller als rekursiv.

```
 public long fakultaet(int n) {
  int result = 1;
  for (int i=1; i<=n; i++) {
    result *= i;
  }
  return result;
}
```

Je nach Neigung wird man einzelnen Fällen dennoch eine rekursive Lösung vorziehen.
Zu (2): Rekursive Programmierung ist dort tatsächlich sinnvoll, wo mehrere Rekursionsaufrufe in einer Methode verwendet werden. Hier ist eine iterative Lösung allgemein deutlich komplizierter. Zu den typischen Anwendungen ist es hilfreich, sich das Ablaufschema eines solchen Programms anzusehen. Im folgenden Bild wird das Beispiel eines rekursiven Programms gezeigt, das in einer Methode zwei Rekursionsaufrufe hat.



Die Rekursionsaufrufe bilden eine Baumstruktur. Rekursive Programmierung eignet sich vor allem dann, wenn das Problem in eine Baumstruktur abbildbar ist. Im nächsten Semester werden wir eine Reihe von Problemen und Lösungskonzepten kennenlernen, die auf eine Baumstruktur hinauslaufen. In dieser Veranstaltung werden wir uns auf einfache Übungsbeispiele beschränken: Die rekursive Berechnung eines Elements der Fibonacci-Folge und die Berechnung von Variationen von Zahlen, ein Problem, das sich nicht so einfach iterativ lösen lässt.

8.4 Beispiele

8.4.1 Die Fibonacci-Folge

Die Fiboniacci-Folge hat ihren Namen nach einer Aufgabe, die von Fibonacci 1202 veröffentlicht wurde. Sie heißt sinngem-a–ss-:
Ein Kaninchenpaar wirft vom zweiten Monat an ein junges Paar und in jedem weiteren Monat ein weiteres Paar. Die Nachkommen verhalten sich ebenso. Wie viele Kaninchenpaare gibt es nach n Monaten?

In Worten ausgedrückt ist die Antwort: Nach n Monaten gibt es alle Kaninchenpaare aus dem (n-1)ten Monat zuzüglich der Anzahl der Paare, die im (n-2)ten Monat schon gelebt und ein weiteres Paar geboren haben. Die Fibonacci-Folge ist also definiert als:

$$f_x = \begin{cases} 0; & x = 0 \\ 1; & x = 1 \\ f_{x-1} + f_{x-2}; & x > 1 \end{cases}$$

Dies lässt sich direkt in ein rekursives Programm übersetzen (es wird vorausgesetzt, dass keine negativen Zahlen übergeben werden):

```
public static int fibonacci(int x) {
  if (x<=1) {
    return x;
  }
  return fibonacci(x-1)+fibonacci(x-2);
}
```

Die ersten Elemente sind:

```
 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
```

8.4.2 Variationen

Ein einfaches Beispiel, das sich nur schwer iterativ umsetzen lässt, ist folgendes: Schreiben Sie eine Funktion

```
public static void printVar(int n)
```

die alle n-stelligen Zahlen ausgibt, die nur die Ziffern 1 bis 3 enthalten. Für $n = 2$ wäre das Ergebnis

```
11,12,13,21,22,23,31,32,33.
```

Für $n = 3$ wäre das Ergebnis

```
111,112,113,121,122,123,131,132,133,211,212,213,221,222,223,231,
232,233,311,312,313,321,322,323,331,332,333.
```

Falls man sich auf ein festes n festlegen kann, ist die Funktion kein Problem. Zum Beispiel könnte man für $n = 3$ schreiben:

```
for (int i=1; i<=3; i++) {
  for (int j=1; j<=3; j++) {
    for (int k=1; k<=3; k++) {
      System.out.println(""+i+j+k);
    }
  }
}
```

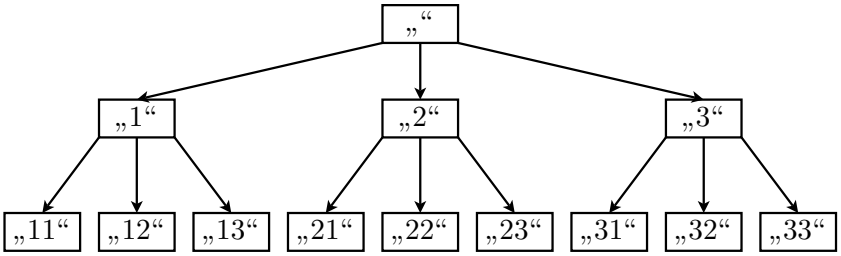
Wenn n aber variabel sein soll, bräuchte man n ineinandergeschachtelte for-Schleifen, wobei n erst zur Laufzeit bekannt ist. Ein solches Konstrukt gibt es in Java nicht. Es gibt zwar mögliche iterative Ansätze, aber die haben ein deutlich anderes Aussehen. Rekursiv kann man das Problem aber recht direkt lösen. Die Funktion `printVar` ruft zuerst eine zweite Funktion

```
private static void printRekVar(String s, int n)
```

auf. Dabei ist `s` der Ausgabestring, der nach und nach zusammengebaut werden soll. `n` ist nach wie vor die Anzahl der Stellen, die das Ergebnis haben soll. Die rekursive Funktion prüft zunächst, ob der zusammengebaute String schon lang genug ist. Wenn ja, wird der String ausgegeben. Dies ist gleichzeitig die Abbruchbedingung. Wenn nein, dann wird dreimal rekursiv `printRekVar` aufgerufen, wobei an `s` jeweils eine 1, eine 2 oder eine 3 angehängt wird. Der komplette Code ist:

```
private static void printRekVar(String s, int n) {
  if (s.length()==n) {     //Abbruchbedingung
    System.out.println(s);
    return;
  }
  for (int i=1; i<=3; i++) {
    printRekVar(s+i,n);   //Rekursiver Aufruf, String-Addition
  }
}

public static void printVar(int n) {
  printRekVar("", n);
}
```

Wenn man sich die einzelnen Funktionsaufrufe ansieht, erhält man wieder eine Baumstruktur. In der folgenden Grafik werden die nötigen Funktionsaufrufe für $n = 2$ abgebildet. In den Kästchen steht jeweils der Übergabeparameter `s`. Beim Durchlauf wird an der Wurzel (hier oben abgebildet) begonnen. Anschließend werden die Verzweigungen nach unten von links nach rechts durchlaufen. Das gleiche geschieht auch auf den nächsten Ebenen, bis es keine weiteren Verzweigungen nach unten mehr gibt. Man ist dann an den sogenannten *Blättern* des Baums angelangt. Dort wird der String jeweils ausgegeben.



Einige ähnliche Probleme (z.B. das *Rucksack-Problem*) werden wir noch ausführlich in der Vorlesung „Algorithmen“ behandeln.

Chapter 9

Größere Programmeinheiten

9.1 Bibliotheken

Die gr-o–ss-te Programmeinheit in Java ist die *Bibliothek (Library)*. Alle Java-Klassen werden automatisch einer Bibliothek zugeordnet. Eine Bibliothek kann zwei verschiedene Formen annehmen:

1. eine jar- oder zip-Datei
2. ein Verzeichnis (einschließlich der Unterverzeichnisse).

9.1.1 Einbinden von Bibliotheken

Java unterscheidet zwischen den Java-Standardbibliotheken *(bootstrap classes)* und den benutzerspezifischen Bibliotheken und legt dazu zwei getrennte Bibliotheks-Listen an. Die Liste der Standardbibliotheken sollte niemals verändert werden, wogegen die Liste der benutzerspezifischen Bibliotheken veränderbar ist. Beide Listen können ausgegeben werden mit:

```
//Benutzerspezifische Bibliotheken
System.out.println(System.getProperty("java.class.path"));
//Standardbibliotheken
System.out.println(System.getProperty("sun.boot.class.path"));
```

Per Default steht in der Liste der benutzerspezifischen Bibliotheken nur der aktuelle Pfad (bzw. in Eclipse der Projekt-Pfad). Das heißt, die einzige benutzerspezifische Bibliothek umfasst alle Dateien im aktuellen Verzeichnis und in den Unterverzeichnissen.[1] Um weitere Bibliotheken hinzuzuladen, muss der sogenannte *Classpath* erweitert werden. Dazu gibt es mehrere Möglichkeiten:

- Setzen der Umgebungsvariablen `CLASSPATH`: Da die Umgebungsvariable den Default-Wert überschreibt, muss unbedingt der aktuelle Pfad mit gesetzt werden, sonst werden die Java-Dateien im aktuellen Pfad nicht mehr gefunden. Beispiel:

[1] Die Dateien in den Unterverzeichnissen liegen in Paketen, siehe dazu den entsprechenden Abschnitt.

```
Linux:   export CLASSPATH=.:/lib/javalibs/mathlib.jar
Windows: set CLASSPATH=.;C:\lib\javalibs\mathlib.jar
```

- Start eines Java-Programms mit

```
java -cp <Liste der Bibliotheken> <Startdatei>
Auch hier muss der aktuelle Pfad mit angegeben werden, damit die Java-Dateien
im aktuellen Pfad noch gefunden werden.
```

- Die Vorgehensweise in Eclipse wird an einem Beispiel im folgenden Abschnitt beschrieben.

Benutzen einer Mathematikbibliothek mit Java

Im Folgenden soll ein kleines Beispielprogramm erstellt werden, das die Mathematikbibliothek *JScience* nutzt. JScience enthält Klassen für Lineare Algebra, Brüche, Polynome und komplexe Zahlen. Zunächst laden wir die Bibliothek herunter. Die JScience-Seite im Web ist `http://jscience.org`. Sie benötigen die Librarys *JScience* und *Javolution* (jeweils eine jar-Datei).
Zum Test legen wir die beiden jar-Dateien in dem Verzeichnis `C:\JScience` ab. Jetzt müssen die Dateien zur Liste der benutzerspezifischen Bibliotheken hinzugefügt werden. Der einfachste Weg ist, die Umgebungsvariable `CLASSPATH` zu setzen:[2]

```
set CLASSPATH=".;C:\JScience\jscience-5.0-SNAPSHOT.jar;
                C:\JScience\javolution-5.5.1.jar"
```

Der Punkt am Anfang ist wichtig, da sonst Java nicht mehr im aktuellen Pfad sucht und damit wahscheinlich das eigentliche Programm nicht mehr findet.
Alternativ kann die Liste auch beim Aufruf von *java* oder *javac* mit dem Kommandozeilenparameter `-cp` angegeben werden.
Wir wollen für unser Programm (wie üblich) Eclipse benutzen. Dazu öffnen wir eine neues Eclipse-Projekt und erzeugen dort eine Klasse `Mathtest.java`. Dort geben wir ein:

```
import org.jscience.mathematics.number.*;

public class Mathtest {

  public static void main(String[] args) {
    Complex c1 = Complex.valueOf(1, 5);
    Complex c2 = Complex.valueOf(3, 2);
    Complex c3 = c1.times(c2);
    System.out.println(c3);
  }
}
```

[2] Hier mit den im Juli 2010 aktuellen Versionen. Die gezeigte Syntax ist die für Windows. Unter Linux ist das Trennzeichen zwischen den Pfadangaben kein Semikolon, sondern ein Doppelpunkt.

Das Programm testet die komplexen Zahlen aus dem JScience-Paket. Eclipse benötigt jetzt noch die Information, wo die JScience-Bibliothek zu finden ist. Dazu ist folgendes in Eclipse einzugeben:

(englische Version):
Menüpunkt *Project* → *Properties,*
Menüpunkt *Java Build Path* und Reiter *Libraries* wählen,
Button *Add External JARs...* drücken,
gewünschte Jar-Dateien auswählen, zweimal OK drücken.

(deutsche Version):
Menüpunkt *Projekt* → *Eigenschaften*,
Menüpunkt *Java-Erstellungspfad*
und Reiter *Bibliotheken* wählen,
Button *Externe JARs hinzuf*ügen drücken,
gewüschte Jar-Dateien auswählen, zweimal OK drücken.

Anschließend kann das Programm ganz normal gestartet werden.

9.1.2 Eigene Bibliotheken erstellen

Aufbau einer jar-Datei

Eine jar-Datei hat den gleichen Aufgau wie eine zip-Datei. Man kann eine Datei `xyz.jar` in `xyz.zip` umbennenen und sie anschließend z.B. mit WinZip öffnen. Hauptunterschied zu einer normalen zip-Datei ist, dass zusätzlich eine Datei `Mainfest.mf` mit eingepackt wird. Hauptbestandteil der Manifest-Datei ist der Klassenname der Start-Klasse. Damit ist es möglich, eine java-Datei direkt zu starten. Es wird dann die main-Methode der Start-Datei ausgeführt. Die Manifest-Datei kann von Hand editiert, vom Programm *jar*, das im Java-Paket enthalten ist, erzeugt oder auch von Eclipse automatisch generiert werden.

Erzeugung einer jar-Datei aus Eclipse

(englische Version):

- Menüpunkt *File* → *Export* und im Dialogfeld *Java* → *JAR file* wählen,
- *Next* drücken,
- die Projekte auswählen, die in die jar-Datei aufgenommen werden sollen,
- im Feld *Select the export destination* den Namen der jar-Datei angeben,
- 2 mal Next drücken,
- im Feld *Select the class of the application entry point* die Klasse angeben, dessen main-Methode gestartet werden soll (für reine Bibliotheksdateien muss hier nichts angegeben werden),

- *Finish* drücken.

(deutsche Version):

- Menüpunkt *Datei* → *Exportieren* und im Dialogfeld *Java* → *Jar-Datei* wählen,
- *Weiter* drücken,
- Die Projekte auswählen, die in die jar-Datei aufgenommen werden sollen,
- im Feld *Exportziel auswaehlen* den Namen der jar-Datei angeben,
- 2 mal Next drücken,
- im Feld *Klasse des Eingangspunkts für die Anwendung auswählen* die Klasse angeben, dessen main-Methode gestartet werden soll (für reine Bibliotheksdateien muss hier nichts angegeben werden).
- *Fertig stellen* drücken.

Eclipse „zippt“ die Dateien in eine jar-Datei und fügt automatisch eine Manifest-Datei hinzu.

Starten einer jar-Datei

Eine jar-Datei kann direkt ausgeführt werden. Es wird dann die Klasse gestartet, die in Eclipse (bzw. in der Manifest-Datei) als Start-Datei angegeben ist. In Windows wird eine jar-Datei durch Doppelklick auf den Dateinamen im Windows-Explorer gestartet. In Linux bzw. in einer Windows-Eingabeaufforderung verwendet man die Kommandozeile

```
java -jar <jar-Dateiname>
```

9.2 Pakete

9.2.1 Laden von Paketen

Betrachten wir eine einfache Java-Klasse, in der eine komplexe Zahl eingelesen wird:

```
public class Mathtest {

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Realteil: ");
    double re = sc.nextDouble();
    System.out.print("Imaginaerteil: ");
    double im = sc.nextDouble();
    Complex c = Complex.valueOf(re, im);
    System.out.println(c);  }
}
```

Diese Programm ist so nicht lauffähig. Beim Start (und in Eclipse) wird für die beiden Klassen *Scanner* und *Complex* die Fehlermeldung

```
... cannot be resolved to a type
```

ausgegeben. Das bedeutet, dass die beiden Klassen nicht gefunden werden können. Rufen wir uns zunächst in Erinnerung: Die beiden Klassen liegen in Bibliotheken. Die Bibliotheken müssen Java bekannt sein. *Scanner* liegt in der Java-Standardbibliothek, die immer bekannt ist. *Complex* liegt in der Bibliothek *JScience*, die entsprechend dem vorigen Kapitel eingebunden werden muss.

Innerhalb der Bibliotheken liegen die Klassen in *Paketen*. Scanner liegt in `java.util`, Complex liegt in `org.jscience.mathematics.number`. Die vollständigen Klassennamen lauten demgem-a–ss-:

```
java.util.Scanner
org.jscience.mathematics.number.Complex
```

Wenn eine Klasse aus einem Paket genutzt werden soll, gibt es 3 Möglichkeiten:

- Sie wird mit dem vollständigen Klassennamen angegeben. In unserem Beispiel wäre das

  ```
  public class Mathtest {
    public static void main(String[] args) {
      java.util.Scanner sc = new java.util.Scanner(System.in);
  System.out.print("Realteil: ");
  double re = sc.nextDouble();
  System.out.print("Imaginaerteil: ");
  double im = sc.nextDouble();
  org.jscience.mathematics.number.Complex c =
        org.jscience.mathematics.number.Complex.valueOf(re, im);
  System.out.println(c);
    }
  }
  ```

  Das führt natürlich zu sehr unübersichtlichen Klassennamen.

- Die Klasse wird durch eine *import*-Anweisung importiert. Die import-Anweisung steht vor der ersten Klassen-Definition. Beispiel:

  ```
  import org.jscience.mathematics.number.Complex;
  import java.util.Scanner;

  public class Mathtest {
    public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Realteil: ");
  ```

```
        double re = sc.nextDouble();
        System.out.print("Imaginaerteil: ");
        double im = sc.nextDouble();
        Complex c = Complex.valueOf(re, im);
        System.out.println(c);
    }
}
```

Unter Eclipse kann man die import-Anweisungen automatisch mit *SHIFT-STRG-o* erzeugen.

- Das komplette Paket wird importiert. Dazu sind die import-Anweisungen aus dem letzten Beispiel durch

  `import org.jscience.mathematics.number.*;`

  `import java.util.*;`

  zu ersetzen.

Die letzte Variante wird *import on demand* genannt. Viele *import on demands* bewirken, dass sich die Zeit für den Compiler-Durchlauf etwas erhöht, haben aber keine weiteren Auswirkungen. Der Grund ist: Die import-Anweisung sagt dem Java-Compiler, wo er die benötigten Klassen findet. In unserem letzten Beispiel würde der Compiler eine Klasse `XYZ` in zwei Schritten suchen:

- Er durchsucht alle Bibliotheken nach den Paketen

  `Default-Paket (Paket ohne Namen)`

  `java.lang (Automatisch eingebunden)`

  `org.jscience.mathematics.number`

  `java.util`

- Sind die Pakete gefunden, werden sie nach der Klasse `XYZ` durchsucht.

Wird statt *import on demand* die spezifische Klasse importiert:

```
import org.jscience.mathematics.number.Complex;
import java.util.Scanner;
```

dann wird für die Klasse `XYZ` nur noch das Default-Paket und `java.lang` durchsucht, was natürlich schneller geht. Die class-Datei wird dadurch aber nicht kleiner und das Programm, wenn es einmal compiliert ist, nicht schneller.

9.2.2 Erstellen eigener Pakete

Unsere bisherigen Klassen wurden bereits, ohne dass wir es beabsichtigt haben, einem Paket zugeordnet, nämlich dem sogenannten *Default-Paket*. Für das Default-Paket gilt:

1. Die Klassen können direkt, ohne import-Anweisung verwendet werden. Der vollständige Klassenname ist gleich dem eigentlichen Klassennamen.

2. Die Klassen müssen direkt in einem Bibliotheksverzeichnis stehen, nicht in einem Unterverzeichnis. Das haben wir erfüllt, da wir in Eclipse die Java-Dateien direkt in das Projektverzeichnis geschrieben haben.

Nun wollen wir die Klassen in Pakete legen. Zum Beispiel wollen wir eine Klasse *Pakettest* angelegen, die im Paket *mypack* stehen soll. Zunächst legen wir versuchsweise die Klasse *Pakettest* in Eclipse an und geben in der Zeile *Paket/Package* den Namen *mypack* an. Eclipse erzeugt die Datei

```
package mypack;

public class Pakettest {

}
```

und legt sie im Unterverzeichnis *mypack* des Projektverzeichnisses an. Damit sind auch schon die zwei Bedingungen genannt, die erfüllt sein müssen. Erstens muss in der ersten Zeile des Programms eine *package*-Anweisung stehen. Zweitens muss die *class*-Datei in einem Ordner der Form

```
Bibliothekspfad\Paketpfad
```

liegen. Wenn also der Bibliothekspfad `C:\lib` heißt, muss die Datei im Verzeichnis `C:\lib\mypack` stehen.

Zum Starten der Datei aus der Kommandozeile ist das Kommando

```
java mypack.Packtest
```

zu benutzen. Es können auch tiefer verschachtelte Ordnerpfade angegeben werden. Ersetzt man im vorigen Beispiel die package-Anweisung durch

```
package myapp.mypack;
```

dann erwartet Java die class-Datei im Ordner `C:\lib\myapp\mypack` und der Startbefehl für Java lautet

```
java myapp.mypack.Packtest
```

Ist die Bibliothek eine jar-Datei, dann entspricht der Paketname dem Pfad der class-Datei innerhalb der jar-Datei. Beispiel: Die Klasse `Complex` in JScience liegt im Paket `org.jscience.mathematics.number`. Das heißt, dass in der Datei `jscience.jar` eine Datei

```
org\jscience\mathematics\numbers\Complex.class
```

vorhanden ist. Auch die Dateien der Java-Klassenbibliothek kann man auf diese Weise identifizieren. Die Klasse `String` im Paket `java.lang` befindet sich im Java-Installationsverzeichnis, Unterordner `lib`, Bibliotheks-Datei `rt.jar` und dort im Unterverzeichnis `java\lang` in der Datei `String.class`. Die Java-Sourcen finden sich übrigens in der direkt im Java-Installationsverzeichnis gelegenen Bibliotheks-Datei `src.zip`.

9.2.3 Eindeutigkeit von Paketen

Ziel der Java-Entwickler ist es, dass die vollständigen Klassennamen aller auf der Welt erstellten Klassen eindeutig sind. Das ist deswegen sinnvoll, weil prinzipiell alle Klassen miteinander kombiniert werden können. Daher gibt es die Vorgabe, sich bei der Namensgebung eines Pakets an die eigene Internet-Adresse anzulehnen. Genauer gesagt, sollen die Paketnamen mit dem eigenen Domain-Namen beginnen, wobei allerdings die Reihenfolge der Namensbestandteile herumgedreht ist. Beispiele:

1. JScience hat den Domain-Namen `jscience.org`. Alle JScience-Pakete beginnen deshalb mit `org.jscience`, wie auch unser Beispiel:

   `org.jscience.mathematics.number`

2. Eclipse-Pakete beginnen mit `org.eclipse`, z.B.

   `org.eclipse.jface.dialogs`

3. Bei uns im Haus erstellte Java-Pakete müssten nach diesem Schema sinnvollerweise mit

   `de.rwth_aachen.itc`

   beginnen.[3]

9.2.4 Pakete und Sichtbarkeitsgrenzen

Bisher kannten wir nur eine Sichtbarkeitsgrenze, nämlich die Klasse. Methoden und Attribute, die als *private* deklariert sind, können nur in der eigenen Klasse gesehen werden, als *public* deklarierte Methode und Attribute auch in anderen Klassen. Java kennt noch zwei weitere Sichtbarkeitsgrenzen: Die abgeleiteten Klassen und die Pakete. Dafür gibt es die Zugangsbeschränkungen *protected* und *default* (das ist, wenn man gar nichts davorschreibt). Die Zugangsbeschränkungen sind kurz gefasst:

public : Alle Klassen haben Zugriff.

protected : Alle Klassen des eigenen Pakets und alle abgeleiteten Klassen (auch in anderen Paketen) haben Zugriff.

default : Nur Klassen des eigenen Pakets haben Zugriff.

private : Nur die eigene Klasse hat Zugriff.

Zweierlei ist zu beachten:

1. *protected* ist, anders als es die Namenswahl suggeriert, freizügiger als *default*.

[3] Der korrekte Domain-Name ist `itc.rwth-aachen.de`. Ein Bindestrich ist jedoch in Java-Paketnamen nicht zugelassen. Nach Java-Konvention ersetzt man ihn durch einen Unterstrich.

2. Eine Zugriffsbeschränkung „Die eigene Klasse und alle davon abgeleiteten“ gibt es nicht. Das ist Java-typisch. In C# sieht es z.B. ganz anders aus:[4]

Java	C#	Zugriff von
public	public	überall
protected	protected internal	Eigenes Package (Assembly) und abgeleitete Klassen
default	internal	Eigenes Package (Assembly)
-	protected	Eigene und abgeleitete Klassen
private	default, private	Nur eigene Klasse

9.3 Dateien

Eine weitere größere Programmeinheit ist die Datei (Sun sagt dazu auch *compilation unit*). Dazu gibt es einige sehr Java-spezifische Regeln:

1. Eine Datei darf mehrere Klassen enthalten.

2. Nur eine Klasse davon darf *public* sein. Das heißt, nur eine davon darf mit

   ```
   public class ...
   ```

   deklariert werden. Diese Klasse muss in der Datei zuoberst stehen.

3. Die anderen Klassen müssen mit

   ```
   class ...
   ```

   deklariert werden. Sie dürfen nicht außerhalb des Pakets benutzt werden. Weiterhin können sie nicht als Start-Klasse dienen.

Diese Regelungen haben zur Folge, dass Java-Projekte die Neigung haben, sich auf viele winzige Dateien aufzusplittern. Das ist zwar von den Java-Entwicklern so beabsichtigt, kann aber durchaus lästig sein. Ein Trick, mehrere Java-Klassen in einer Datei zusammenzufassen, sind die statischen inneren Klassen, die im folgenden Kapitel erklärt werden.[5]

9.3.1 Statische innere Klassen

Diese Kapitel zeigt, wie statische innere Klassen eigesetzt werden, um mehere Klassen in einer Datei zusammenzufassen und welche sonstigen Vorteile sich daraus entwickeln.

Dazu betrachten wir ein Beispiel:

[4] Die C#-Sichtbarkeitsgrenze *Assembly* entspricht eher einer Bibliothek als einem Paket.

[5] Es gibt insgesamt 4 Arten von inneren Klassen. In dieser Vorlesung werden nur die statischen und die anonymen inneren Klassen (Kapitel 6) angesprochen.

```
public class Sternkatalog {

  public static class Stern {
    public String name;
    public double hell;
    public double entf;
    public String bild;
  }
  public static class Katalog {
    public ArrayList<Stern> katalog = new ArrayList<Stern>();
  }
}
```

Die „äußere“ Klasse `Sternkatalog` enthält nichts, außer zwei inneren Klassen `Stern` und `Katalog`. Die inneren Klassen sind mit den Schlüsselworten `public static` deklariert. Sie unterscheiden sich im Verhalten nicht von einer normalen Klasse. Der Hauptunterschied liegt in der Namensgebung. Die Klasse *Katalog* aus diesem Beispiel würde man mit

```
Sternkatalog.Katalog testKatalog = new Sternkatalog.Katalog();
```

erzeugen, was auch außerhalb des Pakets möglich ist. Das macht einerseits deutlich, dass die Klasse `Katalog` nur im Zusammenhang mit `Sternkatalog` sinnvoll ist. Andererseits verhindert es Namensüberschneidungen. Man kann statischen inneren Klassen Namen wie „Element“ oder „Debug“ geben, ohne befürchten zu müssen, dass es bereits andere Klassen dieses Namens gibt. Die äußere Klasse hat nur den Zweck, als *Namensraum* zu dienen. Namensräume werden im nächsten Kapitel erläutert.

9.4 Über Java hinaus: Namensräume

Abstrahieren wir an dieser Stelle etwas und lösen uns von Java. Der vollständige Name einer Klasse setzt sich zusammen aus zwei Bestandteilen: dem **Namensraum** (Namespace) und dem eigentlichen Klassennamen. Die Form dabei ist gewöhnlich:

```
Namensraum.Klassenname
```

wobei der Namensraum durch beliebig viele Punkte gegliedert sein darf. In Java können wir

1. Pakete und
2. Klassen (für statische innere Klassen)

als Namensräume betrachten. Alle Klassen innerhalb desselben Namensraums können sich gegenseitig ansprechen, ohne dass die lange (vollständige) Version des Klassennamens benutzt werden muss. Klassen aus fremden Namensräumen brauchen die lange Version, was natürlich sehr umständlich wäre, wenn man

Klassen nicht aus Namensräumen *importieren* könnte. *Importieren* heißt: Eine Klasse in den eigenen Namensraum mit hineinnehmen, so dass der Klassenname auch in der kurzen Form benutzt werden kann. Dieses Konzept findet sich in allen objektorientierten Programmiersprachen. Unterschiede bestehen allerdings dahin, *was* als Namensraum betrachtet werden kann.

- In C# kann man sich mit der Anweisung `namespace` recht frei Namensräume zusammenstellen. In Visual Studio wird automatisch ein Namensraum mit dem Projektnamen vorgegeben.
- In Python ist eine Datei ein Namensraum mit dem entsprechenden Dateinamen.

9.4.1 Statische innere Java-Klassen und Namensräume

In Java ist die Verwendung von äußeren Klassen als Namensräume in einem Punkt eingeschränkt. Soll eine äußere Klasse als Namensraum importiert werden, darf sie nicht im Default-Paket liegen. In anderen Paketen klappt es. Beispiel:

```
package testpaket;

public class Aussen {
    public static class Innen1 {
        //..
    }
}
```

Im import-Befehl kann jetzt der Name der äußeren Klasse verwendet werden.

```
import testpaket.Aussen.*;

public static void Start(String[] args) {
    Innen1 = new Innen1();
}
```

Chapter 10

Lesbarkeit eines Programms

Bezogen auf die gesamte Lebenszeit eines Software-Produkts geht wesentlich mehr Programmieraufwand in die Wartung als in die Programmierung.[1] Hinzu kommt, dass nur selten ein Produkt vom ursprünglichen Programmierer gewartet wird. Darum ist es wichtig, dass ein Programm nicht nur (möglichst) korrekt, sondern auch **lesbar** und **verständlich** geschrieben ist. Dabei kann man grob zwischen zwei Bereichen unterscheiden:

- Der Bereich der Konzeption. Dazu gehört unter anderem die richtige Auswahl der Datenstrukturen, der Algorithmen und der Entwurfsmuster (mit Namensangabe im Kommentar). Eine gute Konzeption bedeutet auch, dass in Programmiersprachen, die objektorientierte Programmierung anbieten, konsequent objektorientiert programmiert wird. Diese Themen werden in den Vorlesungen „Algorithmen und Datenstrukturen“ und „Software Engineering“ behandelt.
- Der Bereich der optischen Lesbarkeit. Dazu gehören z.B. Einrückung, Namenskonventionen, Klammerung oder Kommentare. Hierzu gibt es Programmierrichtlinien, mit denen wir uns im folgenden Kapitel beschäftigen werden.

10.1 Programmierrichtlinien

Programmierrichtlinien sollen die Lesbarkeit von Programmcode erhöhen. Traditionell sind diese Richtlinien firmenspezifisch. Eine gewisse Bedeutung haben z.B. die Firmenrichtlinien von Microsoft oder die GNU Coding Standards. Für Java gibt es Programmierrichtlinien von Sun, die von den meisten Java-Programmierern (nicht allen) befolgt werden. Die Richtlinien sind in den 20-seitigen *Java Code Conventions* zusammengefasst, die man unter der Adresse

https://www.oracle.com/java/technologies/javase/codeconventions-contents.html

im Internet findet. Im folgenden Kapitel werden einige interessante Teile der Richtlinie vorgestellt, wobei die Richtlinien über Kommentare in einem eigenen Kapitel zusammengefasst werden.

[1] Die *Java Code Conventions* von Oracle sprechen von 80% Wartung.

Umgang mit Programmierrichtlinien

Es können zwei verschiedene Situationen auftreten:

- Die Programmierrichtlinien sind von der Firma oder vom Projektgeber verbindlich vorgeschrieben. Dann ist die Situation klar.
- Es gibt im konkreten Projekt keine verbindlichen Programmierrichtlinien. Dann sollte man dennoch eine verbindliche Festlegung treffen und diese auch dokumentieren. Dabei ist es gut, sich an eine weitverbreitete Programmierrichtlinie zu halten, in Java also in erster Linie an die *Java Code Conventions* von Sun.

  Man sollte aber immer den *Zweck* von Programmierrichtlinien im Hinterkopf haben, nämlich die Erhöhung der Lesbarkeit von Programmen. Gut begründete Verstöße sind also erlaubt, wenn sie der Erhöhung der Lesbarkeit dienen.

10.1.1 Ausgewählte Richtlinien aus den Java Code Conventions

Hier einige ausgewählte und gekürzte Richtlinien aus den Java Code Conventions. Die Auswahl soll einen Eindruck vermitteln, was der Inhalt der Java Code Cenventions ist. Sie ersetzt nicht das Studium der Konventionen selbst.

- 6.1 Benutze für jede Variablendeklaration nur eine Zeile. Das erleichtert die Kommentierung.
- 6.3 Initalisiere lokale Variablen gleich bei der Deklaration.
- 7.1 Jede Zeile sollte nicht mehr als einen Befehl enthalten.
- 7.2 Benutze geschweifte Klammern auch bei einzeiligen if- oder for-Blöcken.
- 7.8 Kommentiere jedes *fall through* bei switch-Anweisungen. Jede switch-Anweisung sollte einen default-Fall haben. Auch der letzte Fall einer switch-Anweisung sollte mit `break` enden.
- 9 Namenskonventionen
  - Benutze Substantive für Klassennamen. Schreibe Klassennamen groß und benutze Binnenversalien (*UpperCamelCase*). Vermeide Abkürzungen, solange sie nicht gängig sind.
  - Interfacenamen gleichen Klassennamen.
  - Benutze Verben für Methodennamen. Methodennamen werden klein geschrieben (*lowerCamelCase*).
  - Variablennamen werden ebenfalls klein geschrieben. Sie sollten sprechend sein. Benutze Variablennamen mit einem Buchstaben ausschließlich für solche Variablen, die nur in einem kurzen Codestück gültig sind (*throwaway variables*).
  - Konstanten werden komplett groß geschrieben. Einzelne Worte werden durch einen Unterstrich getrennt.

- 10.1 Setze die Zugangsbeschränkung für Attribute nicht ohne guten Grund auf `public`. Ein guter Grund ist, wenn die Klasse eine simple Zusammenfassung von Daten ohne spezielles Verhalten ist.
- 10.2 Benutze zum Aufruf statischer Methoden immer die Form `Klassenname.Methodenname`.
- 10.3 Codiere numerische Konstanten nie direkt. Ausnahmen können die Wert 1, 0 und -1 sein, z.B. in einer for-Schleife.
- 10.5.1 Verwende in Rechenausdrücken Klammern großzügig. Gehe davon aus, dass andere Programmierer die Operator-Präzedenzen nicht so gut kennen wie du.

  ```
  if (a == b && c == d) // AVOID!
  ```

  ```
  if ((a == b) && (c == d)) // RIGHT
  ```

10.1.2 Unterstützung in Eclipse

Um Programmierrichtlinien einzuhalten, kann man Hilfsprogramme benutzen, die *Style Checker* oder *Code Reviewer* genannt werden. Eclipse hat solche Hilfsfunktionen bereits eingebaut. Die vier Hilfsfunktionen sind:

- *STRG-I* oder *Source → Correct Indentation* korrigiert die Einrückungen des Codes.
- Der weitaus mächtigere *Formatter* (*SHIFT-STRG-F* oder *Source → Format*) formatiert den Code nach einstellbaren Regeln.
- *Source → Clean Up...* korrigiert den Code nach einstellbaren Regeln und bildet eine Ergänzung zum Formatter.
- Einstellbare *Errors und Warnings* unterbinden das Compilieren von unsauberem Code oder geben eine Warnung aus.

Die Werkzeuge *Formatter*, *Clean Up* und *Errors/Warnings* werden in je einem Unterabschnitt kurz vorgestellt.

Formatter

Menüpunkt: *Source → Format*.
Die Formatierung umfasst z.B. die Zeileneinrückung, die Position der geschweiften Klammern oder die Leerzeichen und Leerzeilen im Code. Die Regeln zur Formatierung sind unter *Window → Preferences → Java → Code Style → Formatter*einstellbar. Wenn man sich dort zum Test ein neues Profil anlegt, gewinnt man einen guten Überblick über die Mächtigkeit des Formatters.

Clean Up

Menüpunkt: *Source → Clean Up...*.
Das „Clean Up“ korrigiert den Code selbst. Beispiele dafür sind:

- Einzeilige Blöcke (if, else, for, ...) werden mit geschweiften Klammern versehen.
- Alle Zugriffe auf Attribute werden mit `this` versehen.
- Annotations, wie `@Override` werden an den passenden Stellen hinzugefügt (zu Annotationen siehe den nächsten Abschnitt).
- Nicht benutzte import-Anweisungen werden gelöscht.

Die genauen Regeln sind unter *Window* → *Preferences* → *Java* → *Code Style* → *Clean Up* einstellbar. Auch hier lohnt es sich, sich einmal alle Optionen anzusehen.

Errors / Warnings

Errors und Warnings werden in Eclipse auf dem Balken links neben dem Source-Code angezeigt. Warnings erscheinen in gelb und verhindern, anders als die roten Errors, nicht das Compilieren des Programms. Syntaktische Fehler verursachen natürlich immer einen Error. Bei „weichere“ Fehlerarten, insbesondere bei Verst-o–ss-en gegen die Java Code Conventions, ist das Verhalten jedoch in Eclipse einstellbar. Der entsprechende Menüpunkt ist *Window* → *Preferences* → *Java* → *Compiler* → *Errors/Warnings*. Auch hier lohnt es sich, einen Blick auf die zahlreichen (über 50) Optionen zu werfen. Man kann sich hier alle lästigen Warnings vom Hals schaffen oder aber sich durch Verschärfen der Restriktionen einen sauberen Programmierstil angewöhnen.

10.1.3 Formatierung des Quelltexts

Auch hier gibt es zahlreiche Regeln in den Java Code-Konventionen, die hier jedoch nicht wiedergegeben werden sollen. Eclipse versucht, den Quelltext möglichst automatisch in der richtigen Weise zu formatieren. Die Regeln dazu kann man unter *Window* → *Preferences* → *Java* → *Code Style* → *Formatter* einsehen und auch gegebenenfalls ändern. Mit der Tastenfolge *STRG-SHIFT-F* veranlasst man Eclipse, die gesamte Datei entsprechend der eingestellten Regeln neu zu formattieren.

10.2 Dokumentationskommentare

Dokumentationskommentare können mit dem Befehl *javadoc* in html-Text umgewandelt werden. Ein Beispiel für die daraus entstehende Dokumentation ist die Java-API, die komplett aus Dokumentationskommentaren besteht. Dokumentationskommentare haben folgendes Format:

```
/**

Dokumentationskommentar

*/
```

Die Java-Programmierrichtlinien empfehlen das folgende Format, das Eclipse auch automatisch nach der Eingabe von `/**` und <`return`> setzt:

```
/**
 *
 * Dokumentationskommentar
 *
 */
```

10.2.1 Einteilung der Dokumentationskommentare

Dokumentationskommentare tauchen vorzugsweise an drei Orten auf:

- vor Klassendefinitionen (Klassen-Javadoc),
- vor der Definition von Attributen (Attribut-Javadoc)
- und vor der Definition von Methoden und Konstruktoren (Methoden-Javadoc)

10.2.2 Javadoc von Klassen

Die typische Javadoc einer Klasse hat das Aussehen:

```
/**
 *
 * Genaue Beschreibung der Klasse, gegebenenfalls mit
 * Beispielen.
 *
 *
 * @version 1.5, November 2007
 * @author  Herbert Mustermann
 */
public class Testklasse {
...
}
```

`@version` und `@author` sind spezielle Tags, die Javadoc nutzt, um die für Javadoc-Dokumente typische Seitenstruktur aufzubauen.

10.2.3 Javadoc von Attributen

Bei Attributen ist es üblich, eine einfache Javadoc-Zeile ohne Tags vor die Definition des Atrtibuts zu setzen. Beispiel:

```
/** Beschreibung des Attributs maxLen */
public double maxLen;
```

10.2.4 Javadoc von Methoden und Konstruktoren

Hier finden die meisten Tags Verwendung. Die wichtigsten sind:

- `@param` für die Beschreibung eines Übergabeparameters. Es folgt der Name und die Beschreibung des Parameters. Beispiel:

  ```
  @param breite Breite des Rechtecks
  ```

- `@return` für die Beschreibung des Rückgabewerts. Beispiel:

  ```
  @return Flaeche des Rechtecks
  ```

- `@throws` für die Beschreibung der möglichen Exceptions. Es folgt der Name und die Beschreibung der möglichen Exception. Beispiel:

  ```
  @throws ArithmeticException falls Laenge oder Breite kleiner 0.
  ```

Nachfolgend ein ausführliches zusammenhängendes Beispiel. Eine Methode

```
public static String wandleZiffer(String zahl, int systemAlt,
                                  String systemNeu)
```

die `zahl` vom Zahlensystem `systemAlt` in das Zahlensystem `systemNeu` wandelt, wird kommentiert.

```
/**
 *
 *Wandelt eine Zahl von einem Zahlensystem in ein anderes.
 *Alle Zahlensysteme zwischen dem 2er und dem 20er-System
 *sind erlaubt. Ziffernwerte ab 10 werden durch Buchstaben
 *dargestellt, beim 20er System gibt es also die Ziffern
 *0123456789ABCDEFGHIJ.
 *
 *@param zahl Die zu wandelnde Zahl als String. Der String
 *darf nur die Zeichen enthalten, die aufgrund des
 *Zahlensystems systemAlt erlaubt sind.
 *
 *@param systemAlt Das Ausgangs-Zahlensystem. Erlaubt sind
 *die Werte 2 bis 20.
 *
 *@param systemNeu Das neue Zahlensystem. Erlaubt sind
 *die Werte 2 bis 20.
 *
 *@return Die Zahl im neuen Zahlensystem in String-Darstellung.
 *
 *@throws NumberFormatException falls zahl falsche Zeichen
 *enthaelt.
 *
 *@throws ArithmeticException falls systemAlt oder systemNeu
 *nicht zwischen 2 und 20 liegen.
 *
```

```
 */
public static String wandleZiffer(String zahl, int systemAlt,
                                  int systemNeu) {
...
}
```

Der erste Satz des Kommentars wird im API-Dokument als Kurzbeschreibung verwendet. Der komplette erste Teil (vor den Tags) steht in der ausführlichen Beschreibung.

10.2.5 Weitere Tags und Formatierungsmöglichkeiten

Weitere mögliche Tags, die sowohl bei Klassen als auch bei Attributen und Methoden auftauchen dürfen sind:

- `@see`: Verweis auf eine andere Stelle in der Dokumentation. Beispiele:

  `@see "Hinweis, z.B. auf Buch"`

  `@see package.class#member`

  `@see <a href=ÜRL#value">label</a>`

  Erzeugt eine See Also-Sektion mit dem entsprechenden Link.

- {`@link`}: Verweis auf eine andere Stelle in der Dokumentation. Im Gegensatz zu `@see` wird der Link direkt in den Text gesetzt und keine *See Also*-Sektion erzeugt.

Andere Tags werden z.B. in Wikipedia erklärt (Stichwort *javadoc*). Neben den Javadoc-Tags dürfen auch beliebige HTML-Tags verwendet werden. Ausnahmen sind <H1>...<H6> und <HR>.

Umlaute in der Javadoc

Da alle Zeichen direkt in HTML übernommen werden, müssen Umlaute ausgeschrieben oder als HTML-Tag geschrieben werden, also z.B. *ae* oder *ä* statt ä. Emtwicklungsumgebungen wie Eclipse setzen die Umlaute beim Erstellen der Javadoc automatisch um.

10.2.6 Erzeugung und Kontrolle der Javadoc

Auf der Kommandozeile können die Java-Dateien des aktuellen Verzeichnisses mit

```
javadoc *.java
```

in eine Javadoc umgesetzt werden. Die verschiedenen Optionen des Javadoc-Befehls sollen hier nicht erklärt werden. In Eclipse kann die gesamte Javadoc des Projekts mit *Project* → *Generate Javadoc* erzeugt werden. Es gibt in Eclipse noch eine weitere Möglichkeit, Dokumentationskommentare zu sehen. Wenn man im unteren Fenster (wo normalerweise die Konsole sichtbar ist) den Reiter *Javadoc* anklickt, erscheint immer dann, wenn sich der Cursor in einem Dokumentationskommentar befindet, dort der zugehörige formatierte Text.

10.3 Verwendung von Kommentaren

In Kommentaren können prinizipiell beliebige Texte untergebracht werden. Trotzdem haben sich Regeln ausgebildet, wann Kommentare sinnvoll eingesetzt werden.

10.3.1 Notwendigkeit von Kommentaren

Darüber, wie viele Kommentare in einem Programm angebracht sind, bestehen stark unterschiedliche Meinungen. Bei der Assembler-Programmierung kann es durchaus sinnvoll sein, *jede Programmzeile einzeln* zu kommentieren, wie das nachfolgende Beispiel zeigt:

```
INC 2  2         Increase buffer pointer
JEC 3  1
J3P    1B        Repeat ten times
OUT    -20,2     (PRINTER)
J2P    DONE      Have we printed both lines?
ENT2   0         Set buffer pointer for 2nd buffer
```

Gelegentlich trifft man diese Regel auch in Hochsprachen an, durchgesetzt hat sich jedoch eine andere Kommentierungsweise:

- Kommentiere nur, was zum Verständnis des Programms nötig ist. Die Kenntnis der Java-Bibliothek sollte dabei vorausgesetzt werden.
- Kommentiere den Zweck und das Ziel eines Code-Bereiches.
- „Tricks“ und schwierige Code-Stellen müssen kommentiert werden.

Es gibt also ein „zu wenig“ und ein „zu viel“ an Kommentaren, was auch die folgenden Zitate verdeutlichen:

Von der Möglichkeit, Kommentare in Programme einzufügen, sollte reger Gebrauch gemacht werden.[2]

The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer. (Java Code Conventions)

Hier noch ein kurzes Zitat aus dem englischen Wikipedia: „Good comments ... clarify intent.“.

10.3.2 Einsatzgebiete von Kommentaren

Anfangskommentare

Vor der ersten Java-Zeile steht ein Anfangskommentar. Der Anfangskommentar ist kein (javadoc-) Dokumentationskommentar, sondern ein „normaler“ Blockkommentar. In den Java-Programmierrichtlinien findet man:

[2] *Krekel, Trier: Die Programmiersprache PASCAL.*

All source files should begin with a c-style comment that lists the programmer(s), the date, a copyright notice, and also a brief description of the purpose of the program. For example:

```
/*
* Classname
*
* Version info
*
* Copyright notice
*/
```

Oft steht hier auch die sogenannte *History*, d.h. die Aufzählung der Änderungen seit der 1. Programmversion. Diese Liste ist oft ziemlich lang. In der Regel wird sie automatisch von Versionsverwaltungssystemen (CVS, Subversion) erstellt. Dieser Kommentar ist nicht identisch mit der Klassen-Javadoc. Zwischen Anfangskommentar und Klassen-Javadoc stehen die `package`- und die `import`-Anweisungen.

Gliederung des Quelltexts

Kommentare gliedern einen Quelltext in Abschnitte. Sie helfen, sich in einem längeren Code zurechtzufinden. Vor dem Kommentar sollte eine (oder mehrere) Leerzeilen stehen.

Erläuterung einer einzelnen Zeile

Kommentare können eine schwierige Programmzeile erläutern, damit andere oder der Autor selbst diese später leichter verstehen.

Hinweis auf zu erledigende Arbeit

Es gibt spezielle Kommentare für verbesserungswürdige oder fehlerhafte Programmteile. Die Java-Programmierrichtlinien sagen:

Use `XXX` in a comment to flag something that is bogus but works. Use `FIXME` to flag something that is bogus and broken.

Außerdem gibt es noch `TODO` als Platzhalter für fehlende Codestücke. Eclipse färbt die Kommentare

```
//XXX
//FIXME
//TODO
```

in einer speziellen Farbe (blaugrau) ein und setzt eine spezielle Markierung an den rechten Rand.

Auskommentierung von Code

Mit `/*` und `*/` können Codesücke (vorübergehend) ungültig gemacht werden. Besonders häufig werden Zeilen, die dem Debugging dienen, auskommentiert.

10.3.3 Fehler bei der Kommentierung

Widersprüche von Code und Kommentar

Eine Gefahr bei der nachträglichen Änderung eines Programmes ist, dass vergessen wird, auch den Kommentar entsprechend zu ändern. In diesem Fall bleibt ein Kommentar zurück, der im Widerspruch zum Programmcode steht und verwirrt, statt zu unterstützen. Daher gilt die wichtige Regel:
Beim Ändern von Programmcode muss stets auch der Kommentar geändert werden.
In den Java Programmierrichlinien wird diese Regel andersherum gesehen:
In general, avoid any comments that are likely to get out of date as the code evolves.

Schimpfworte im Kommentar

Muss das gesagt werden? Unflätige Kommentare über die Programmiersprache, das Betriebssystem, die Entwicklungsumgebung, die Projektbedingungen, die Arbeitskollegen oder den Chef sind unprofessionell (und dennoch immer wieder zu finden).

Appendix A

Formatierung von Ausgaben

Mit den Methoden `System.out.printf` und `String.format` kann man Daten formatieren und ausgeben. Formatieren kann z.B. folgendes bedeuten:

- Daten linksbündig oder rechtsbündig auf eine bestimmte Mindesbreite ausrichten.
- Dezimalbrüche mit einer bestimmten Anzahl von Nachkommastellen ausgeben.
- Ganze Zahlen durch führende Nullen auf eine definierte Stellenanzahl bringen.
- Zahlen in Hexadezimalzahlen wandeln.

A.1 Syntax

```
//Gibt formatiert auf dem Bildschirm aus
System.out.printf(formatstring, wert1, wert2, ...);

//Schreibt formatierten Text in einen String
String s = String.format(formatstring, wert1, wert2, ...);
```

Es können beliebig viele Wert in einem Rutsch formatiert werden. Wieviele Werte auf welche Weise formatiert werden, hängt vom Formatstring ab.

A.2 Struktur des Formatstrings

Die Regeln für den Formatstring stammen aus der Sprache C, in der es einen ähnlichen Befehl (printf) gibt. Die Regeln wurden außer in C++ und Java auch in den wichtigsten Skriptsprachen, wie Perl, Python und Ruby übernommen. Sprachen mit anderen Regeln sind C#, Pascal und Fortran.
Im einfachsten Fall besteht den Formatstring aus einem beliebigen Text, der 1:1 ausgegeben wird.[1] Weitere Parameter gibt es nicht. Beispiel:

[1] Die Ausgabe erfolgt ohne Zeilenvorschub, entspricht also eher `print` als `println`. Zum Zeilenvorschub in printf-Befehlen siehe das Kapitel über Escape-Sequenzen.

```
System.out.printf("Hello world");
```

In diesen Text kann man *Umwandlungsbefehle* einbetten. Ein Umwandlungsbefehl beginnt immer mit einem %-Zeichen und endet mit einem speziellen Umwandlungsbuchstaben. Beispiel:

```
int eu = 2;
int ce = 10;
System.out.printf("Endpreis: %d Euro %d Cent", eu, ce);

//Ausgabe: 2 Euro 10 Cent
```

Der erste Umwandlungsbefehl (`%d`) holt sich den auf den Formatstring folgenden Parameter (`eu`), formatiert ihn (in diesem Fall als Ganzzahl) und setzt ihn an genau diese Stelle ein. Der zweite Umwandlungsbefehl macht das gleiche mit dem nächsten Parameter (`ce`).
Der Umwandlungsbuchstabe muss dem Typ des dazugehörigen Parameters entsprechen (ansonsten gibt es einen Laufzeitfehler). Mögliche Umwandlungsbuchstaben sind:

Symbol	Mögliche Typen	Beispiel
%b	boolean	true oder false
%c	char	a
%d	byte, short, int, long	200
%f	float, double	45.460
%e	float, double	4.546e+01 (Exponentialdarst.)
%s	String	Java ist toll
%x	byte, short, int, long	ffd2 (Hexadezimal)

Zwischen dem %-Zeichen und dem Umwandlungsbuchstaben können noch verschiedene weitere Zeichen eingefügt werden. Damit können viele Effekte erreicht werden. Wir werden uns der Übersichtlichkeit halber auf drei wichtige beschränken.

A.2.1 Rechts- und linksbündiges Formatieren

Zwischen %-Zeichen und Umwandlungsbuchstabe wird eine Zahl eingefügt. Java interpretiert den Betrag der Zahl als Mindestbreite. Ist die Zahl positiv, wird rechtsbündig formatiert, bei negativen Zahlen linksbündig.
Beispiele (Leerzeichen werden als ␣ wiedergegeben):

printf-Befehl	Ausgabe
`.printf("%5d", 123)`	␣␣123
`.printf("%-8d", 123)`	123␣␣␣␣␣
`.printf("%10s", "Hallo")`	␣␣␣␣␣Hallo
`.printf("%-10s", "Hallo")`	Hallo␣␣␣␣␣

A.2.2 Dezimalbrüche mit Nachkommastellen

Bei %f und %e kann im Anschluß an die Mindestbreite noch die Anzahl der Nachkommastellen angegeben werden. Es wird gerundet. Mindestbreite und Nachkommastellen werden durch einen Punkt getrennt. Es kann wahlweise auch eine der beiden Angaben weggelassen werden.
Beispiele (Leerzeichen werden als ␣ wiedergegeben):

printf-Befehl	Ausgabe	Kommentar
`.printf("%.2f", 34.565)`	`34.57`	2 NKS (Nachkommastellen)
`.printf("%.5f", 34.565)`	`34.56500`	5 NKS
`.printf("%7.2f", 34.565)`	`␣␣34.57`	Mindestbreite 7, 2 NKS
`.printf("%7f", 34.565)`	`34.565000`	Mindestbreite 7, 6 NKS (default)
`.printf("%.3e", 34.565)`	`3.457e01`	Exponentialform, 3 NKS

A.2.3 Definierte Stellenanzahl für ganze Zahlen

Sollen ganze Zahlen ziffernweise bearbeitet werden, ist es sinnvoll, sie vorher auf eine definierte Stellenanzahl zu bringen. Dazu gibt es zwei weitere Formatierungsregeln:

- Steht vor der Mindestbreite eine 0 (nur sinnvoll bei rechtsbündiger Formatierung), dann wird nicht mit Leerzeichen, sondern mit Nullen aufgefüllt.

- Steht vor der Mindestbreite ein +, dann beginnen positive Zahlen mit einem +.

Beispiele (Leerzeichen werden als ␣ wiedergegeben):

printf-Befehl	Ausgabe
`.printf("%03d", 5)`	`005`
`.printf("%+03d", 5)`	`+05`
`.printf("%+03d", -5)`	`-05`
`.printf("%+3d", 5)`	`␣+5`
`.printf("%-+3d", 5)`	`+5␣`
`.printf("%+-3d", 5)`	`+5␣`

A.2.4 Formatieren ohne Bildschirmausgabe

Oft soll der formatierte Text in einem String gespeichert statt auf dem Bildschirm ausgegeben werden. Dazu verwendet man den Befehl `String.format`, der wie `printf` funktioniert. Die Zeile

```
String x = String.format("%03d",5);
```

speichert den String `005` in der Variablen `x`.

A.3 Andere Formatierungsklassen

Generell bietet Java zwei Möglichkeiten an, Zeichenketten zu formatieren: Die Klasse `java.util.Formatter` und die Klasse `java.text.Format`. Beide Klassen haben nicht nur ähnliche Namen, sondern auch ähnliche Funktionalität, allerdings ist die Syntax für den Formatierungsstring ganz unterschiedlich:

- Die Klasse `Formatter` wird intern von `printf` benutzt, kann aber auch direkt verwendet werden. Die Regeln für den Formatierungsstring entsprechen denen von `printf`.
- Die Klasse `Format` und davon abgeleitete Klassen, wie z.B. `DecimalFormat` bieten eine andere, Java-spezifische Formatierungsmöglichkeit.

Ist eine bestimmte Formatierung in einem System nicht möglich, kann man eventuell auf das andere ausweichen. Der umständlichere, aber objektorientert sauberere Weg ist `Format`.

Appendix B

Escape-Sequenzen

Innerhalb von Strings kann man mit den sogenannten *Escape-Sequenzen* Zeichen darstellen, die entweder über die Tastatur nicht zu erreichen sind oder in Strings nicht direkt verwendet werden dürfen.[1] Eine Escape-Sequenz beginnt immer mit einem *Backslash* (\). Die wichtigsten Escape-Sequenzen sind:

Sequenz	Bedeutung
`\n`	Neue Zeile (line feed)
`\b`	Backspace
`\t`	Tab
`\"`	Doppeltes Anführungszeichen (")
`\'`	Einfaches Anführungszeichen (')
`\\`	Backslash (\)
`\u00b1`	Unicode-Zeichen (hier B1(hex): ±)

Die Unicode-Zeichen bis 127 entsprechen den ASCII-Zeichen. Diese sind im nächsten Kapitel in einer Tabelle zusammengefasst.

[1] Ein Beispiel für den zweiten Fall ist das doppelte Anführungszeichen, das als String-Ende interpretiert wird.

Appendix C

ASCII-Zeichen

Einige ASCII-Codes[1] kommen so häufig vor, dass man sie auswendig kennen sollte. Es sind dies:

Zeichen	ASCII-Code
Leerzeichen	32
Die Ziffer 0	48
A	65
a	97

Die anderen Buchstaben und Ziffern ergeben sich aus dem Ziffernwert bzw. der Position im Alphabet. Die komplette ASCII-Tabelle sieht wie folgt aus:

0	NUL	16	DLE	32		48	0	64	@	80	P	96	‘	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	’	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

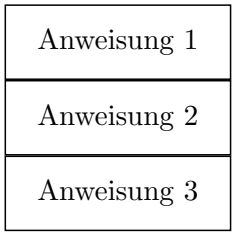
[1] *ASCII* steht für *American Standard Code for Information Interchange.*

Appendix D

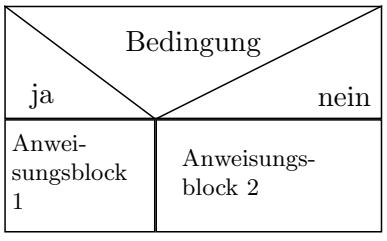
Struktogramme und Flussdiagramme

D.1 Elemente von Struktogrammen

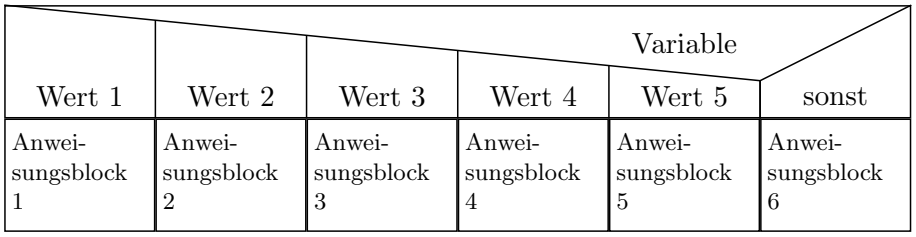
D.1.0.1 Linearer Ablauf (Sequenz)



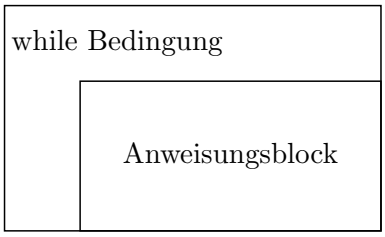
D.1.0.2 Verzweigung (if-then-else)



D.1.0.3 Fallauswahl (switch-case)

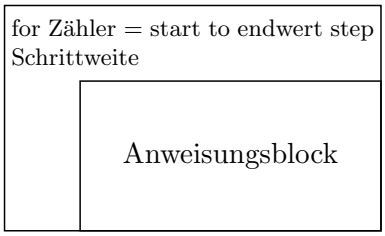


D.1.0.4 Kopfgesteuerte Schleife (while)

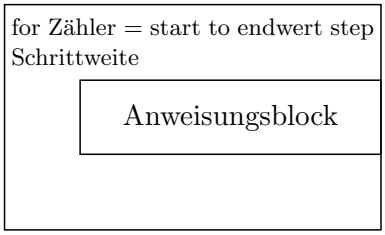


D.1.0.5 Fußgesteuerte Schleife (do-while)

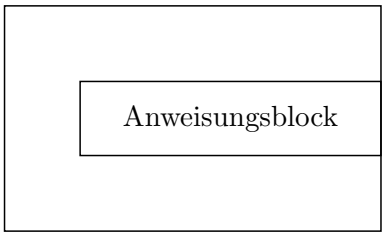
D.1.0.6 Zählergesteuerte Schleife (for)



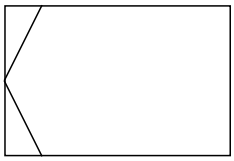
Häufig findet man bei der for-Schleife statt der DIN-Norm die folgende Variante:



D.1.1 Endlosschleife



D.1.2 Aussprung (break)



D.1.3 Aufruf eines Unterprogramms

D.2 Elemente von Flussdiagrammen

Oval: Start oder Ende

Rechteck: Operation

Rechteck mit doppelten Linien: Unterprogramm

Raute: Verzweigung