Java lernen mit BoSym – Board of Symbols

Stephan Euler
TH Mittelhessen
Version 0.65b – Herbst 2021

8. Dezember 2021

Änderungen

Versionen

0.65b Dezember 21 Kleinere Änderungen bei Methoden etc.

0.65a November 21 Rückgabe bei `compareTo` korrigiert

0.65 Oktober 21 Unterschied der Formen *Kreis* und *Oval*, BoSL-Befehle aktualisiert

0.64 Juli 21 Lambda-Ausdücke für ActionListener

0.63 Juni 21 Assertions

0.62 Juni 21 Überarbeitung bei Datenstrukturen

0.61 April 21 Kleinere Änderungen bei Klassen und Vererbung

0.6 März 21 Anhang mit Infos zur Klasse `Dialogs`

0.55b Dez. 20 Umbenennung Zustand in Tabelle zum Spiel *Dame*

0.55a Dez. 20 Kleinere Änderung bei Beispiel `float` zu Rundungsfehlern

0.55 Dez. 20 Umbenennung in BoSym, neuer Abschnitt zu Autoboxing, diese Übersicht zu Änderungen

Inhaltsverzeichnis

Kapitel 1

Vorwort

1.0.1 Hinweise zum Lesen

Das Skript ist gedacht als Einführung in Java für Programmieranfänger. Daher liegt der Schwerpunkt auf den wichtigsten Sprachelementen und einfachen, leicht nachvollziehbaren Beispielen. Daneben enthält es allerdings auch Abschnitte mit spezielleren Themen. Diese Abschnitte dienen eher als Referenz und können beim ersten Lesen gut übersprungen werden. Als Markierung dient das △-Zeichen. Enthält eine Überschrift dieses Zeichen, so dient dieser Abschnitt mehr der Vollständigkeit als dem Verständnis beim Einstieg. Mehrere △-Zeichen symbolisieren einen noch stärkeren Referenz-Charakter.

Aus Gründen der besseren Lesbarkeit wird in diesem Skript lediglich die männliche Form verwendet. Dies hat keinen Einfluss auf die inhaltliche Wertigkeit, sondern dient als neutrale Bezeichnung für Angehörige des weiblichen und männlichen Geschlechtes.

1.0.2 Dank

Mein Dank gilt Doris Brand für die sorgfältige Korrektur der ersten 7 Kapitel.

Stephan Euler

Kapitel 2

Einführung

Frage 2.1. Wozu brauchen wir dieses BoSym?

Java verfügt über eine umfangreiche Bibliothek von Klassen zur Grafikprogrammierung. Der Einstieg ist nicht wirklich schwierig, erfordert aber doch solide Grundkenntnisse. Andererseits ist die Ausgabe in einem Textfenster für viele nicht besonders motivierend. Hier soll BoSym helfen. Mit seiner Hilfe kann man leicht grafische Ausgaben erzeugen.

2.1 Voraussetzungen

Die Anwendung ist in Java geschrieben. Zur Ausführung benötigt man die Java-Laufzeitumgebung (*Java Runtime Environment*, kurz JRE), die auf den allermeisten Rechnern bereits vorhanden ist. Java ist weitgehend unabhängig vom Betriebssystem, so dass es hier keine Probleme geben sollte. Um die selbst geschriebenen Java-Programme auszuführen, ist außerdem eine Entwicklungsumgebung (*Java Development Kit*, JDK) nötig. Dabei gibt es verschiedene Möglichkeiten, wobei allerdings auch das jeweilige Betriebssystem eine Rolle spielt. Informationen zu diesen eher technischen Fragen sind im Moodle-Kurs und auf der Seite zusammen gestellt. Das Projekt selbst ist auf der Plattform GitHub[1] zugänglich.

2.2 Erste Schritte

Die Java-Anwendung befindet sich als Archiv zusammen gepackt in der Datei `jserver.jar`. In den meisten Fällen startet ein Doppelklick auf diese Datei die Anwendung. Alternativ kann man den Befehl

```
java -jar jserver.jar
```

[1] `https://github.com/stephaneuler/board-of-symbols`

in der Konsole (Eingabeaufforderung) eingeben. Wenn alles richtig eingerichtet ist, sollte ein Fenster wie in Bild 2.1 erscheinen. Wir sehen ein 10×10 Felder großes Spielfeld. Jedes einzelne Feld ist mit einem Kreis bedeckt und in jedem Kreis steht eine Zahl. Dies ist die fortlaufende Nummer, über die wir das Feld ansprechen können. Sollten die Nummern nicht erscheinen, so kann man sie über das Menü *Formen* ↪ *Nummerierung Ein/Aus* erscheinen lassen. Die Nummerierung beginnt links unten mit 0 und läuft dann nach rechts und oben bis zu 99.

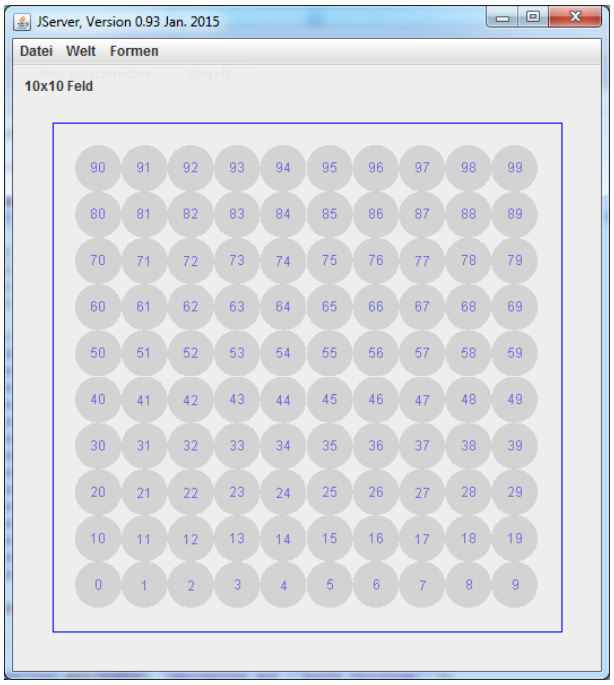


Abbildung 2.1: JServer

Frage 2.2. Warum zählen wir nicht ab 1?

Im täglichen Leben beginnen wir mit der 1. Außer der im Kölner Karneval besungenen *Kaygasse Numero 0* kenne ich zumindest keine Straße, deren Hausnummern mit 0 beginnt. Aber in den Programmiersprachen der C-Familie ist die Zählung ab 0 üblich (*Zero-based numbering*). Dieser Vorgabe folgt die Nummerierung der Felder. Ursprünglich gab es durchaus gute technische Gründe für die 0-Nummerierung. Beim Herunterzählen konnte der Vergleich auf 0 durch einen direkten Prozessorbefehl ausgeführt werden: *jump on zero* oder ähnlich. Hintergründe dazu findet man z. B. im Artikel von Edsger W. Dijkstra [Dij82]. Mittlerweile spielt dies keine Rolle mehr, aber die 0 hat ihre Startposition behauptet.

Über die jeweilige Nummer kann man ein Symbol ansprechen und verändern. Wir beginnen mit einem einfachen Einfärben. Dazu wählen wir zunächst den

Menüpunkt *Welt* ↪ *Fenster für Code Eingabe* und sollten ein Fenster ähnlich wie 2.2 sehen. Das Fenster besteht aus drei Bereichen:

- oben: Steuerelemente
- Mitte: Eingabe der Anweisungen
- unten: Meldungen beim Ausführen

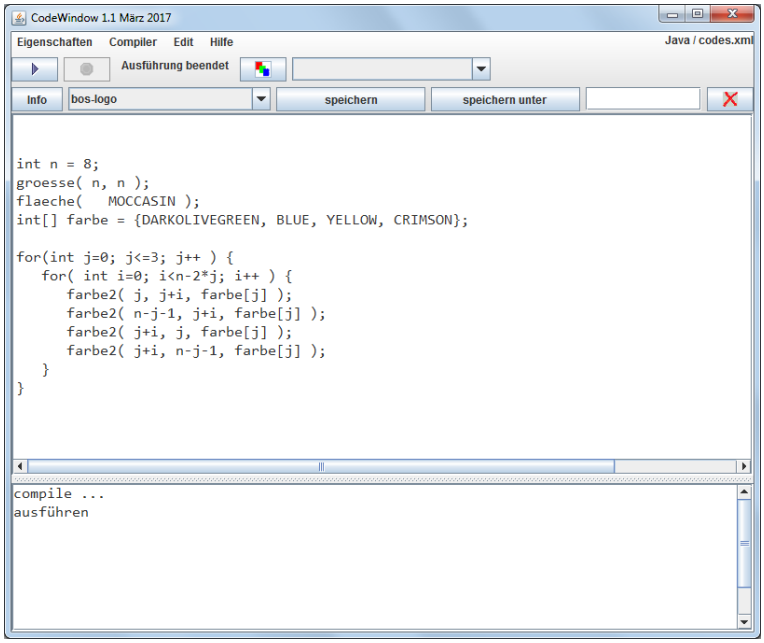


Abbildung 2.2: Eingabe für Code

Ins Eingabefenster schreiben wir unsere erste Anweisung:

```
farbe( 12, 0xff );
```

Es handelt sich dabei um einen kleinen Ausschnitt aus einem Java-Programm. Wir bezeichnen dies auch als Programmcode. BoSym unterstützt neben Java auch andere Programmiersprachen wie C. Daher ist es wichtig, die korrekte Programmiersprache einzustellen. Dies erfolgt über das Menü *Compiler*. Der ausgewählte Compiler wird rechts oben angezeigt.

In unserer ersten Anweisung ist `farbe` der Name einer Methode. Methoden erkennt man an den runden Klammern um die Argumente. Das Konzept kennen wir von mathematischen Funktionen. In anderen Programmiersprachen spricht man ebenfalls von Funktionen.

Als Beispiel steht in der Mathematik $sin(\pi/2)$ für die Berechnung der Sinus-Funktion. Die Funktion bekommt einen Wert – $\pi/2$ – und berechnet daraus ein Ergebnis. Wir wissen dabei nicht, wie die Berechnung ausgeführt wird. Aber wir

können beispielsweise am Taschenrechner die Funktion ausführen und erhalten das gesuchte Ergebnis. Diesem Grundprinzip – wir übergeben Werte an eine Funktion, diese führt Berechnungen aus und gibt ein Resultat zurück – folgen auch die Methoden in Java. Wir können Methoden als Bausteine verwenden, ohne dass wir uns Gedanken über deren innere Abläufe machen müssen. Methoden in Java sind aber nicht das gleiche wie Funktionen in der Mathematik. Es gibt grundlegende Unterschiede, um die wir uns aber bei unserer Programmierung vorerst nicht kümmern müssen.

Zurück zu unserer Eingabe. Wir rufen eine Methode auf und übergeben ihr zwei Werte als Argumente. Die beiden Argumente werden durch ein Komma getrennt. In diesem Fall sind wir nur an der ausgeführten Aktion interessiert. Eventuell gibt `farbe` auch ein Ergebnis zurück, aber wir ignorieren es einfach. Abgeschlossen wird die Anweisung durch das Semikolon `;`. Die beiden Argumente geben die Position und die Farbe an. Das erste ist klar, wir wollen das Symbol mit der Nummer 12 färben. Beim zweiten Argument – dem Farbwert – ist es etwas komplizierter.

Wir verwenden den RGB-Farbraum, bei dem eine Farbe durch das additive Mischen der drei Grundfarben Rot, Grün und Blau gebildet wird. Für jede der drei Grundfarben wird der Anteil als Zahl zwischen 0 und 255 angegeben. Die Obergrenze 255 ist gerade $2^8 - 1$, so dass wir sie als Binärzahl mit 8 Einsen schreiben würden. Dementsprechend sind für jede Grundfarben 8 Stellen (Bits) reserviert. Der Gesamtwert ergibt sich dann durch Hintereinanderschreiben der Binärzahlen.

Anstelle der recht umständlichen Binärdarstellung verwendet man eher die Darstellung im 16-er System als Hexadezimalzahl. Dort steht jede Ziffer (0 bis 9 und dann A, B, C, D, E, und F, groß oder klein geschrieben) für 4 Bits. Der Anteil jeder Grundfarbe wird dann mit zwei Ziffern angegeben. In Java erkennen wir Hexadezimalzahlen am Anfang `0x`. In unserem Beispiel war der Farbwert `0xff`. Das letzte Byte ist gefüllt und demnach haben wir einen maximalen Anteil Blau und keine Anteile in den beiden anderen Grundfarben. Dementsprechend sollte die Darstellung wie in Bild 2.3 erscheinen. Durch die eingeschaltete Nummerierung ist die Farbe etwas blass, so dass man die Schrift noch lesen kann. Die Darstellung als Hexadezimalzahl ist nach etwas Eingewöhnung übersichtlich, wir hätten aber auch mit

```
farbe( 12, 255 );
```

den Wert als Dezimalzahl angeben können. Bei neueren Versionen von Java kann man sogar die Binärzahl

```
farbe( 12, 0b11111111 );
```

direkt schreiben.

Frage 2.3. Muss ich jetzt das RGB-Modell verstehen und Bits und Bytes herum schubsen?

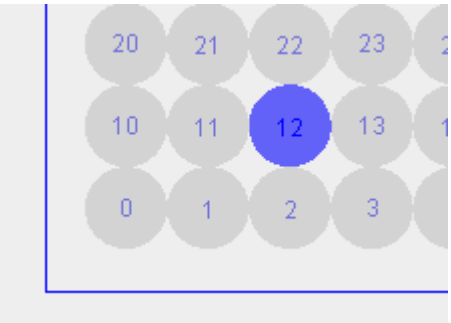


Abbildung 2.3: Erstes Beispiel mit Farbe

Nein, BoSym bietet zwei Hilfen zur Farbeingabe. Zunächst gibt es einen Farbwähler wie man ihn aus anderen Anwendungen kennt. Dort kann man entweder die Anteile der Grundfarben über die Schieberegler einstellen oder per Klick in den Farbbereich eine Farbe direkt auswählen. Der zugehörige RGB-Wert wird dann in einem Feld angezeigt, von wo wir ihn mittels Copy-Paste holen und einfügen können (`0x` davor nicht vergessen). Weiterhin gibt es eine Auswahl-Menü von vordefinierten Farben. Wählt man eine der Farben über das Menü aus, wird der Name direkt an der aktuellen Zeigerposition eingefügt. Wir hätten unser Beispiel also auch als

```
farbe( 12, BLUE );
```

schreiben können.

Frage 2.4. Was passiert wenn ich mich vertippe?

Der eingegebene Text wird zunächst zu einer vollständigen Java-Klasse ergänzt und in einer Datei (`ATest.java`) gespeichert. Diese Datei wird von einem speziellen Programm analysiert und – vereinfacht gesprochen – in ein ausführbares Programm übersetzt. Bei Java wird dieser Code von einer virtuellen Maschine (VM) ausgeführt. Unseren – von Menschen für Menschen geschriebenen – Text bezeichnet man daher als Quelltext (Quellcode, *source code*). Das Ergebnis ist Bytecode – Code, den der Prozessor direkt verstehen und ausführen kann.

Den Vorgang nennt man Kompilierung, den Übersetzer Compiler. Falls bei der Analyse Fehler gefunden werden, bricht der Compiler den Vorgang ab und meldet die gefundenen Fehler. Zum Beispiel bekommen wir bei einem vergessenen Semikolon am Ende der Anweisung die Meldung

```
compile ...
ATest.java:147: error: ';' expected
farbe( 12, BLUE )
                 ^
1 error
compile gescheitert
```

Die Meldungen sind mehr oder weniger hilfreich. Kommen viele Meldungen, so sollte man sich zunächst um den ersten Fehler kümmern. Häufig stammen die weiteren Meldungen von Folgefehlern, die wie magisch verschwinden, sobald der erste Fehler beseitigt ist.

Frage 2.5. 147 ist wohl die Nummer der Zeile mit dem Fehler. Das passt aber gar nicht zum Programmcode.

Die Zeilennummer bezieht sich auf die generierte Klasse, in die der Code-Schnipsel eingebettet wurde. Im Menü *Compiler* gibt es einen Befehl, um sich diesen Code mit Zeilennummern anzeigen zu lassen. Erst wenn das Programm aus Sicht des Compilers fehlerfrei ist, wird ein ausführbares Programm erzeugt. Dieses wird in unserer Umgebung sofort ausgeführt. Der Compiler weiß aber nichts über die BoSym-Anwendung. Fehler, die die Logik von BoSym betreffen, kann er daher nicht finden. So ist aus Sicht des Compilers

```
farbe( 100000, BLUE );
```

vollkommen in Ordnung. Der Compiler hat die Information, dass `farbe` zwei Zahlen als Argumente benötigt. Die Bedeutung der Zahlen und mögliche Einschränkungen im Wertebereich kennt er nicht. Dass es wohl nicht so viele Felder geben wird, muss die Anwendung selbst behandeln. In diesem Beispiel erscheint die Fehlermeldung

```
ERROR - 100000 out of Range
```

Dabei handelt es sich aber nicht um eine Compiler-Meldung, sondern der Fehler wird erst während der Ausführung gemeldet.

Frage 2.6. Okay, die Farben sind damit klar. Was kann ich noch ändern?

Als nächstes ändern wir die Symbole. Dazu gibt es eine Funktion `form(i, s)`. Das erste Argument ist wieder die Feldnummer, das zweite gibt den gewünschten Symbolnamen (eigentlich eher Kurznamen) an. Der Name ist jetzt anders als die Farbe keine Zahl, sondern ein kurzer Text. Texte werden in doppelte Anführungszeichen geschrieben. Betrachten wir die Erweiterung

```
farbe(12, BLUE);
form( 12, "s" );
form( 2, "d" );
form( 11, "*" );
```

mit dem Ergebnis 2.4. Hier werden auf drei Feldern die Kreise durch andere Symbole ersetzt, ein Quadrat, eine Raute und ein Stern. Folgende Symbole sind möglich:

Typ	Kürzel	Details
Kreis	`c`	*circle*, Standard-Symbol beim Start
Oval	`o`	*oval*
Quadrat	`s`	*square*
Raute	`d`	*diamond*
Stern	`*`	
Plus	`+`	
Linie	`|`	vertikal
Linie	`-`	horizontal
Linie	`/`	schräg
Linie	`\`	schräg (siehe Hinweis im Text)
Säule	`b`	*bar*, senkrechte Säule ab der Unterkante des Feldes
Block	`B`	Rechteck mit fester Höhe
Rahmen	`F`	*frame*, ein Rahmen um das Symbol-Feld
Dreieck	`tld`	*triangle left down*, Ecke links unten
Dreieck	`trd`	*triangle right down*, Ecke rechts unten
Dreieck	`tlu`	*triangle left up*, Ecke links oben
Dreieck	`tru`	*triangle right up*, Ecke rechts oben
Würfel	`d1`, `d2`, ...	*dice*, Würfel mit 1 bis 6 Punkten
zufällig	`r`	*random*, Symbol wird zufällig gewählt
leer	`none`	Feld ist leer

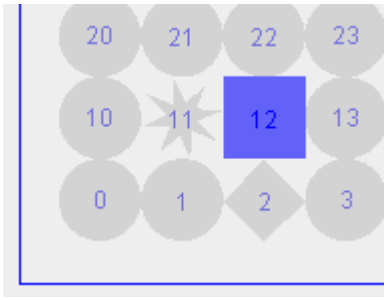


Abbildung 2.4: Wechsel der Symbole

Der Schrägstrich \ (*Backslash*, Rückstrich) hat in Java eine spezielle Bedeutung. Er leitet normalerweise ein Sonderzeichen wie z. B. `\n` für einen Zeilenumbruch ein. Um eine Schrägstrich selbst zu erhalten, muss er geschützt als `\\` eingeben werden. Zwei Kürzel haben eine besondere Wirkung. Bei `r` wird ein zufälliges Symbol ausgewählt, während bei `none` ein eventuell vorhandenes Symbol entfernt wird und das Feld leer bleibt. Insbesondere bei diesen beiden Fällen möchte man manchmal alle Symbole gleichzeitig ändern. Dies kann man jederzeit über das Menü *Formen* erreichen. Alternativ steht die Methode `formen()` zur Verfügung. Beim Aufruf gibt man lediglich das Symbolkürzel an. Als Beispiel füllt

```
formen( "r" );
```

das gesamte Spielfeld zufällig mit Symbolen.

Frage 2.7. Was unterscheidet die Formen `Kreis` und `Oval`?

Kreise werden auch bei einer Änderung der Fenstergröße immer als Kreise dargestellt. Demgegenüber wird die Form *Oval* an die Dimension des jeweiligen Symbolfeldes angepasst und als Ellipse gezeichnet. Bild **??** zeigt dieses Verhalten. Das Symbol (2,1) wurde auf *Oval* gesetzt und wird dementsprechend höher gezeichnet. Bei quadratischer Zeichenfläche sind die beiden Symbole identisch.

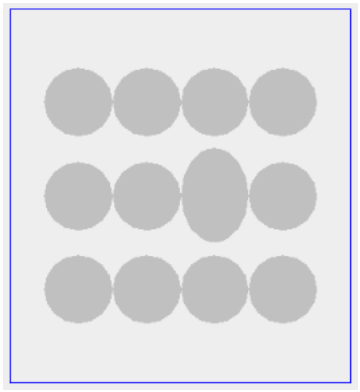
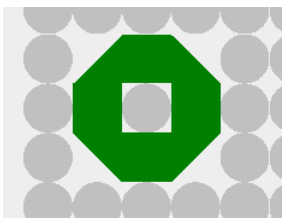


Abbildung 2.5: Vergleich der Formen Kreis und Oval

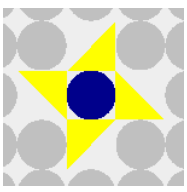
Nachdem wir jetzt Farbe und Symbol jedes Feldes ändern können, ist die Zeit für einige Übungen gekommen.

Übung 2.2.1. Einfache Muster
Schreiben Sie Anweisungen, die die folgenden Muster erzeugen:

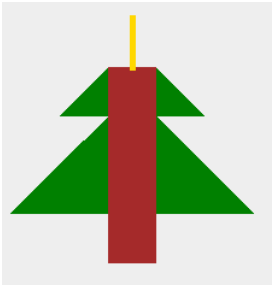
1. Stadion



2. Windrad



3. Baum



Frage 2.8. Kann ich mehrere Symbole auf einem Feld kombinieren, z. B. zwei Dreiecke mit unterschiedlichen Farben?

Nein, man kann sich das wie bei *Schach* oder *Mensch ärgere dich nicht* vorstellen. Auf jedem Feld ist nur Platz für ein Symbol. Setzt man ein neues Symbol, so wird das alte vom Brett genommen. Allerdings wäre es schade, wenn jedes Feld nur jeweils eine Farbe hätte. Daher gibt es die Möglichkeit, zumindest eine zweite Farbe als Hintergrund anzugeben. Das Vorgehen ist das gleiche wie bei der Hauptfarbe, die entsprechende Methode heißt `hintergrund`. Eine kleines Beispiel dazu: der Code-Abschnitt

```
form( 11, "tld" );
farbe(11, BLUE);
hintergrund(11, YELLOW);
form( 13, "*" );
farbe(13, BLUE);
hintergrund(13, YELLOW);
```

erzeugt das Muster in Bild 2.6. Wie das erste Feld zeigt, kann man auf diesem Weg zwei nebeneinander liegende Dreiecke nachbilden.

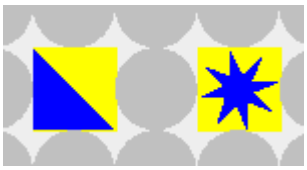


Abbildung 2.6: Zwei Felder mit gelbem Hintergrund

Frage 2.9. Wie kann ich den Code abspeichern?

Die Code-Schnipsel werden in einer Datei gespeichert (sofern keine eigene Datei gewählt: `codes.xml`). Als Format wird XML (*Extensible Markup Language*) verwendet. Dabei werden neben dem eigentlichen Code noch einige Informationen wie das Erstellungs- und Änderungsdatum oder der Name des Autors oder der Autorin (kann man im Menü Eigenschaften eintragen) abgelegt. Beim Speichern

vergibt man einen Namen für einen Code-Schnipsel, über den man ihn später auch wieder laden kann. Zum Abschluss dieses Kapitels noch ein kurzer Blick auf einige Hilfsmethoden. Die folgenden Methoden

Name	Funktionsweise
`groesse(int x, int y)`	neue Größe für das Brett
`flaeche(int f)`	Hintergrundfarbe
`rahmen(int f)`	Rahmenfarbe
`farben(int f)`	Farbe aller Symbole
`loeschen()`	löscht (fast) alle Farb-Einstellungen

sind manchmal nützlich, um Größe und Aussehen einzustellen. Das `int` vor den Argumenten legt fest, dass es sich um ganze Zahlen handeln muss – dazu mehr im nächsten Kapitel. Ein Aufruf der Methode `loeschen` zu Beginn entfernt eventuell vorhandene Einstellung aus vorhergehenden Ausführungen. Bei einer Änderung der Brettgröße werden ebenfalls alle Symbole zurückgesetzt.

2.3 Übungen

Übung 2.3.1. Buchstabe
Schreiben Sie Anweisungen für eine schöne Darstellung des Anfangsbuchstabens Ihres Namens.

Kapitel 3

Variablen und ganze Zahlen

Bisher haben wir Feldindex und Farbe immer direkt als Zahl eingegeben. Das wird schnell mühsam und ist auch schwer zu ändern oder zu erweitern. In diesem und dem folgenden Kapitel werden wir sehen, wie auch aufwändigere Muster durch entsprechende Möglichkeiten der Sprache Java einfach generiert werden können.

3.1 Variablen

Beginnen wir mit dem einfachen Beispiel

```
farbe( 11, 255 );
farbe( 12, 255 );
farbe( 13, 255 );
farbe( 14, 255 );
```

in dem vier aufeinander folgende Felder blau gefärbt werden. Das funktioniert, ist aber nicht besonders elegant. Wollen wir das Muster verschieben oder die Farbe ändern, so müssen wir alle Zeilen ändern. Besser wäre es, die Informationen an einer Stelle zu bündeln, so dass wir uns bei Änderungen auch nur um eine Stelle kümmern müssen. Beginnen wir mit dem Index und führen eine so genannte Variable ein:

```
int i = 11;
```

In dieser kleinen Zeile stecken einige Neuigkeiten. Vor dem =-Zeichen steht ein Datentyp – `int` – und der Name `i` unserer ersten Variablen. Stellen wir uns das vor wie einen Platz in einem Lagerregal oder ein Schließfach am Bahnhof. Wir wollen in dieses Fach etwas ablegen und später dort auch wieder abholen. Vielleicht wollen wir später auch etwas anderes in dieses Fach legen. Damit wir unser Fach später wieder finden, geben wir ihm einen Namen. In Java und verwandten Programmiersprachen müssen wir außerdem festlegen, welche Art von Daten in dieses Fach gelegt werden soll. In unserem anschaulichen Beispiel Schließfach können wir auch zwischen verschiedenen Größen auswählen. Je nach dem ob wir

nur eine kleine Einkaufstüte oder einen sperrigen Koffer abstellen wollen, müssen wir ein Fach entsprechender Größe wählen.

Java unterscheidet mehrere Typen von Daten. Unser `int` steht für Integer, der englischen Bezeichnung für ganze Zahlen. Wir legen also fest, dass unser `i` nur negative oder positive ganze Zahlen einschließlich der 0 enthalten kann.

Allgemein gilt in Java, dass der Datentyp einer Variablen fest ist und sich zur Laufzeit nicht mehr ändert. Es handelt sich um eine explizite und statische Typisierung. Viele neuere Programmiersprachen – insbesondere interpretierte Sprachen wie PHP oder Javascript – verwenden demgegenüber eine implizite und dynamische Typisierung. Zur Laufzeit wird der Datentyp bei einer Zuweisung passend festgelegt. Dabei kann sich der Datentyp einer Variablen durchaus während der Laufzeit ändern. Beide Ansätze haben Vor- und Nachteile.

Das =-Zeichen ist keineswegs ein mathematisches Gleichheitszeichen, sondern der so genannte Zuweisungsoperator. Der Wert auf der rechten Seite wird der Variablen auf der linken Seite zugewiesen. In unserem Fall steht dort nur eine Zahl. Im allgemeinen kann dort aber auch ein Ausdruck stehen, Beispiele dazu folgen gleich. Wichtig ist nur, dass der Ausdruck auf der rechten Seite der Zuweisung zum Typ der Variablen auf der linken Seite passt. Insgesamt haben wir mit der kleinen Zeile also folgendes erreicht:

- Es gibt jetzt eine Variable mit dem Namen `i`.
- Die Variable ist vom Typ `int`.
- Sie wird mit dem Wert 11 belegt.

Die ersten beiden Punkte bezeichnen wir als Definition der Variablen, den dritten als Initialisierung. Man kann auch Variablen definieren, und ihnen erst später Werte zuweisen. Allerdings meldet der Compiler, falls man den Wert einer nicht initialisierten Variablen verwenden möchte. Der Code

```
int i;

farbe( i, BLUE );
```

führt zur Fehlermeldung

```
ATest.java:147: error: variable i might not have been initialized
farbe( i, BLUE );
       ^
```

Umgekehrt geht es gar nicht. In Java muss jede Variable vor ihrer Verwendung definiert werden. Der Compiler überprüft dies und meldet Verstöße in der Form

```
ATest.java:147: error: cannot find symbol
farbe( j, RED );
```

```
      ^
  symbol:   variable j
  location: class ATest
1 error
```

Nach diesen Vorbereitungen können wir jetzt endlich unsere Variable einsetzen und unser Beispiel ändern zu

```
int i = 11;

farbe( i, 255 );
farbe( i + 1, 255 );
farbe( i + 2, 255 );
```

Beim ersten Aufruf von `farbe` verwenden wir direkt unsere Variable, bei den beiden folgenden addieren wir jeweils die Verschiebung dazu. Die absolute Position steht jetzt nur noch an einer Stelle. Unser Mini-Muster kann also durch eine einzige Änderung verschoben werden.

Frage 3.1. Besonders variabel ist `i` aber in diesem Fall nicht.

Das stimmt, in dieser Form nutzen wir `i` wie eine Konstante. Wir können den Code-Abschnitt allerdings wie folgt umschreiben:

```
int i = 11;

farbe( i, 255 );
i = i + 1;
farbe( i, 255 );
i = i + 1;
farbe( i, 255 );
```

Jetzt ändert `i` tatsächlich den Inhalt. Nach jedem Aufruf von `farbe` wird in der Zuweisung

- der Wert in `i` geholt
- eine `1` dazu addiert
- das Ergebnis dann wiederum nach `i` geschrieben.

In diesem Fall hat die Variable `i` in der Anweisung `i = i + 1;` zwei verschiedene Bedeutungen. Auf der rechten Seite der Anweisung ist der Inhalt der Speicherzelle gemeint. Dieser Wert wird zu 1 addiert. Das Ergebnis wird dann in die mit `i` bezeichnete Speicherzelle geschrieben. Man spricht auch von R-Wert und L-Wert (rvalue, lvalue), wenn man die Bedeutung auf der rechten bzw. linken Seite meint. Mit den englischen Begriffen kann man sich unter den Namen auch *read value* und *location value* vorstellen. Eine Konstante kann nur als R-Wert benutzt werden.

Eine Anweisung in der Art $7 = 3 + 4$; ist nicht sinnvoll und auch gar nicht erlaubt, der Compiler meldet

```
error: lvalue required as left operand of assignment
```

Um konsequent zu sein, führen wir eine zweite Variable für die Farbe ein:

```
int i = 11;
int linienFarbe = 255;

farbe( i, linienFarbe );
i = i + 1;
farbe( i, linienFarbe );
i = i + 1;
farbe( i, linienFarbe );
```

In diesem Fall ändert sich der Wert der Linienfarbe nicht. Trotzdem lohnt es sich, den Zahlenwert in eine Variable zu speichern und damit über ihren Namen ansprechen zu können. Der Code wird lesbarer und Änderungen werden einfacher. Um explizit auszudrücken, dass die Variable ihren Wert nicht ändern wird, kann man das Schlüsselwort `final` angeben:

```
final int linienFarbe = 255;
```

Frage 3.2. Ist `linienFarbe` dann vergleichbar mit den Farbnamen wie `BLUE`?

Ja, die Farbnamen sind auch final-Variablen, allerdings noch etwas allgemeiner – dazu später mehr. Die Wiederholung der gleichen Zeilen ist noch unschön. Im nächsten Kapitel werden wir auch dies vermeiden lernen.

Typinferenz mit var

Mit Version 10 wurde 2018 in Java eine vereinfachte Schreibweise zur Typ-Festlegung mittels `var` eingeführt. Ziel war, Code durch Vermeiden von Wiederholungen übersichtlicher zu gestalten. Betrachten wir als Beispiel eine Anweisung in der Form

```
Point2DInteger p = new Point2DInteger(3, 5);
```

(Details dazu später). Hier kann man das doppelte Schreiben von `Point2DInteger` vermeiden und stattdessen

```
var p = new Point2DInteger(3, 5);
```

schreiben. Der Compiler übernimmt dann automatisch den Typ des Ausdrucks auf der rechten Seite für die Deklaration. In unseren Anwendungen können wir beispielsweise mit

```
var i = 10;
```

eine int-Variable `i` anlegen. Dabei muss die Deklaration und Initialisierung in einer gemeinsamen Anweisung stehen. Die neue Schreibweise ändert aber nichts am Wesen der Typisierung – `i` ist ein `int` und bleibt dies auch für die gesamte Lebenszeit. Damit der Typ erkannt werden kann, muss natürlich ein eindeutig zuordenbarer Ausdruck zugewiesen werden. Eine Deklaration ohne Zuweisung

```
var i;
```

hingegen führt daher Fehlermeldung

```
Cannot use 'var' on variable without initializer
```

3.1.1 Ausgabe mit print, println und printf

An dieser Stelle ist ein Einschub zur Ausgabe sinnvoll. Man kann mit der Methode `println` in der Form

```
System.out.println( 4 * 12 );
```

Werte ausgeben lassen. In unserer Anwendung erscheint die Ausgabe im unteren Fenster. Dabei steht die Endung `ln` für einen Zeilenumbruch am Ende der Ausgabe. Mit der Variante `print` können mehrere Ausgaben in die gleiche Zeile geschrieben werden. Der Aufruf `println()` kann für einen Zeilenumbruch ohne weitere Ausgabe verwendet werden.

Der Ausdruck im Argument wird ausgewertet und dann als Text ausgegeben. Dabei kann man auch numerische Ausdrücke und Zeichenfolgen mischen. So ergibt

```
System.out.println( "wert: " + 4 * 12 );
```

die Ausgabe

```
wert: 48
```

Dabei wird zuerst die Multiplikation durchgeführt. Dann wird das Ergebnis in eine Zeichenfolge gewandelt und an die erste angehängt. Das Zeichen `+` steht in diesem Fall für das Aneinanderhängen der Zeichenketten (Konkatenation).

Frage 3.3. Was passiert, wenn man in einem Ausdruck `+` zum Aneinanderhängen von Zeichenketten und zur Addition von Zahlen mischt?

Beide Operationen haben gleichen Rang. Daher werden entsprechende Ausdrücke von links nach rechts ausgewertet. Bei

```
System.out.println("3 + 5 = " + 3 + 5 );
```

wird dementsprechend zunächst die 3 in eine Zeichenkette gewandelt und angehängt und anschließend die 5. Die Ausgabe ist demnach

```
3 + 5 = 35
```

Sollen die beiden Zahlen addiert werden, so kann man die Reihenfolge der Auswertung durch Klammern oder Vertauschen ändern. Die beiden Varianten

```
System.out.println("3 + 5 = " + (3 + 5 ) );
System.out.println( 3 + 5 + " = 3 + 5 ");
```

ergeben das wahrscheinlich gewünschte Ergebnis. Bei `print` hat man keinen Einfluss auf die Umwandlung in eine Zeichenkette. Als Alternative wurde aus der Programmiersprache C die Methode `printf` (**print f**ormatted) übernommen. Die allgemeine Form von `printf` ist:

```
printf("Kontrollzeichenkette", Argument1, Argument2, ... )
```

Die durch Anführungszeichen begrenzte Zeichenkette (*Format String*) kann Text (Kommentar) enthalten, der direkt so ausgegeben wird. Weiterhin werden durch `%`-Zeichen Formatbeschreiber eingeleitet, mit denen festgelegt wird, wie die folgenden Argumente auszugeben sind. Im einfachen Fall folgt auf das `%` ein Buchstabe, der das Format angibt. Im Beispiel

```
System.out.printf("3 + 5 = %d\n ", 3 + 5 );
```

wird mit `%d` angegeben, dass ein Integerwert im Argument folgen wird. Die Bezeichnung `\n` steht für eine neue Zeile (*new line*). Ohne diese Angabe würde nach der Ausgabe kein Zeilenumbruch erfolgen. Details der Formatierung können durch zusätzliche Angaben festgelegt werden. Als Beispiel bewirkt `%3d`, dass bei der Ausgabe mindestens 3 Zeichen ausgegeben werden. So werden bei

```
System.out.printf("3␣+␣5␣␣=␣%3d\n",␣3␣+␣5␣);
System.out.printf("3␣*␣15␣=␣%3d\n",␣3␣*␣15␣);
```

in der Ausgabe

```
3␣+␣5␣␣=␣␣␣8
3␣*␣15␣=␣␣45
```

zusätzliche Leerzeichen eingefügt (Zur Verdeutlichung diesmal mit dem Symbol ␣ dargestellt). Bei `%d` wird die Zahl im Dezimalsystem dargestellt. Alternativ kann man mit `%x` auf hexadezimale (16er-System) und mit `%o` auf oktale (8er System) Ausgabe wechseln. Als Beispiel liefert

```
int a = 123456;
System.out.printf("%o (okt) %d (dez) %x (hex)\n", a, a, a );
```

die Ausgabe

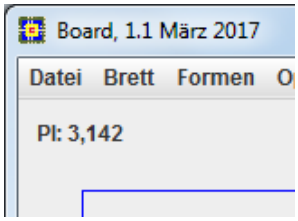
```
361100␣(okt)␣123456␣(dez)␣1e240␣(hex)
```

Intern analysiert `printf` die Zeichenkette mit den Formatbeschreibern. Dabei wird Anzahl und Typ der weiteren Argumente bestimmt. Daher ist es wichtig, dass die übergebenen Argumente auch genau mit dieser Spezifikation übereinstimmen. Eventuelle Differenzen werden vom Compiler nicht beanstandet. Erst

während des Programmablaufs kommt es zu Fehlern. Die Formatierungsmöglichkeiten lassen sich auch außerhalb von `printf` verwenden. Im Beispiel

```
statusText( "PI: " + String.format( "%.3f", Math.PI ));
```

wird die Ausgabe der Zahl π durch die Angabe `.3` auf 3 Nachkommastellen beschränkt. Der zusammengesetzte Text erscheint in der Statuszeile als



Dabei wird die aktuelle Region und Sprache des Standorts berücksichtigt. Im Beispiel wird daher wie in Deutschland üblich ein Komma als Dezimaltrennzeichen verwendet. Alternativ kann man den Text auch in die Formatierung aufnehmen:

```
statusText( String.format( "PI: %.3f", Math.PI ));
```

3.1.2 Variablennamen △

Der Name einer Variablen kann in Java nach folgenden Regeln gebildet werden:

- Der Name beginnt mit einem Buchstaben.
- Anschließend folgt eine beliebige Folge von Buchstaben und Ziffern.
- Der Unterstrich `_` (*underscore*) kann wie ein Buchstabe eingesetzt werden.
- Groß- und Kleinschreibung wird unterschieden.
- In Java reservierte Wörter (Schlüsselwörter) sind verboten (also nicht `int long;`).
- Java-Konvention: Namen von Variablen beginnen mit einem Kleinbuchstaben.

Neben den festen Vorgaben sollte man folgende allgemeine Hinweise beachten:

- Namen sollten passend und weitgehend selbsterklärend (aussagekräftig) sein (nicht `int eineintvariable;`).
- Namen von Integer-Variablen fangen oft mit Buchstaben von `i` bis `n` an.
- Für kurzlebige Variablen können kurze Bezeichnungen wie `i`, `j`, `k` verwendet werden.
- `l` (*klein L*) ist wegen der Verwechslungsgefahr mit der Ziffer `1` kein guter Variablenname.

- Konsequent sein: wenn sich die Bedeutung einer Variable im Programm ändert, auch ihren Namen anpassen.
- Die Struktur innerhalb eines Namens wird mit Groß- / Kleinschreibung markiert:
  - `anzahlStudenten`
  - `linienFarbe`

  Aufgrund der Ähnlichkeit mit Kamelhöckern spricht man von *CamelCase* („KamelSchrift“), genauer gesagt bei Beginn mit einem Kleinbuchstaben *lowerCamelCase*. In älteren Programmen findet man häufiger den Unterstrich als Trennzeichen (`brett_groesse`, *snake case*).
- Einheitliche Sprache (Englisch, Deutsch oder ...).
- Konsistenz! Ein einmal eingeführter Stil für Namen, Einrückungen, etc. sollte beibehalten werden.

Bei großen Software-Projekten werden oft Namenskonventionen verbindlich vorgegeben.

Frage 3.4. Darf man auch Umlaute in Variablennamen verwenden?

Im Prinzip ja. Java erlaubt Umlaute und ähnliche Sonderzeichen sowie Währungszeichen. Aber selbst erfahrene Java-Programmierer kennen oft nicht die genauen Regeln für die Bildung von Variablennamen und halten sich an die oben angegebenen Grundregeln. Das Problem liegt in der unterschiedlichen internen Darstellung der Sonderzeichen. Dafür gibt es mehrere Standards und der Wechsel zwischen verschiedenen Standards ist oft problematisch. Solange man nur auf einem Rechner und in einer Umgebung entwickelt, funktioniert alles gut. Aber beim Wechsel zu einem anderen Betriebssystem oder einer anderen Entwicklungsumgebung kann es passieren, dass die Sonderzeichen nicht mehr richtig dargestellt werden. Auch beim internationalen Austausch können sprachspezifische Sonderzeichen verwirren. Häufig stellt sich schon das Problem, wie man solche Zeichen überhaupt eingeben kann. Daher ist es sehr üblich, sich auf die Buchstaben `A` bis `Z` zu beschränken. Ein Sonderfall sind Namen in automatisch generiertem Code, wo die Lesbarkeit keine Rolle spielt. Dort findet man eher spezielle Formen wie z. B. `$1`.

3.2 Datentyp Integer △

Der Grundtyp `int` ist 4 Byte groß und enthält Zahlen in 2er-Komplement-Darstellung. Der Wertebereich geht von -2147483648 (-2^{31}) bis +2147483647 ($2^{31}-1$).

Konstanten für `int` können sowohl als Dezimalwerte als auch in den Zahlensystemen zur Basis 2, 8 und 16 angegeben werden. Für Oktalzahlen fügt man eine führende `0` ein (Beispiel `0156`), bei Hexadezimalzahlen schreibt man 0x... (Beispiel `0xFF`). Binärzahlen beginnen mit `0b`. Das Vorzeichen wird durch - oder ein optionales + gekennzeichnet. Zur besseren Lesbarkeit kann man die Zahlen mit _-Zeichen unterteilen (Beispiel `2_483_999`). Neben dem Basistyp `int` gibt es weitere Typen:

Typ	Speicherbedarf	Wertebereich
byte	1 Byte	-128 ... 127
short	2 Byte	-32768 ... 32767
int	4 Byte	-2147483648 ... 2147483647
long	8 Byte	-9223372036854775808 ... 9223372036854775807

Dabei stellt sich die Frage: was passiert bei Bereichsüberschreitungen? Betrachten wir dazu ein Beispiel:

```
int i = 2147483647; // groesste int Zahl
System.out.println("i: " + i);
System.out.println("i + 1: " + (i + 1) );
```

Die Ausgabe lautet:

```
i: 2147483647
i + 1: -2147483648
```

Indem wir eine `1` zu der größten positiven Zahl addiert haben, sind wir zu der kleinsten negativen Zahl gelangt. Java verwendet das Zweierkomplement zur binären Darstellung von ganzen Zahlen. Dabei kann man sich die Zahlen, wie in Bild 3.1 dargestellt, im Kreis angeordnet vorstellen. Gegenüber der Null stoßen die kleinste und die größte Zahl aneinander. Unter- oder Überschreitung führt dann jeweils in den anderen Bereich.

Frage 3.5. Kann man solche Überläufe beim Rechnen abfangen?

In der Klasse `Math` gibt es Methoden zum Rechnen mit Prüfung auf Überlauf. So führt der Code

```
int n = Math.addExact(2147483647, 1 );
```

zu einem Laufzeitfehler:

```
java.lang.ArithmeticException: integer overflow
```

Frage 3.6. Kann man tatsächlich durch die kleineren Datentypen Speicher sparen? Lohnt es sich also beispielsweise `short` statt `int` zu nehmen, wenn die Werte immer nur im kleineren Bereich liegen?

Das hängt von der Plattform ab, auf der die Anwendung läuft. In der Regel wird der Typ `int` verwendet – auch wenn beispielsweise ein Zähler in einer for-Schleife offensichtlich nur kleine Werte annimmt.

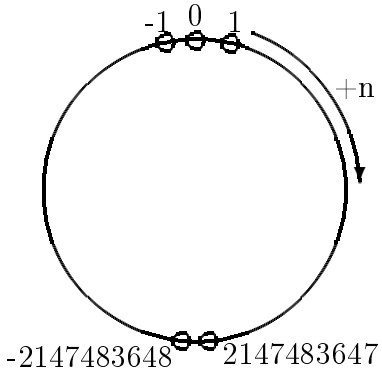


Abbildung 3.1: Anordnung von Integerzahlen

3.2.1 Rechnen mit Integerwerten

Java unterstützt die Grundrechenarten mit den Operatoren `+`, `-`, `*` und `/`. Daneben gibt es noch den Modulo-Operator `%`, der den Rest bei der Division ergibt. Für die Operatoren gilt die übliche Hierarchie. Bei gleichberechtigten Operatoren wird der Ausdruck von links nach rechts abgearbeitet. Bei der Division muss man berücksichtigen, dass das Ergebnis wieder ein Integer-Wert ist und damit ein eventueller Rest verloren geht. Dadurch spielt bei komplexeren Ausdrücken unter Umständen die Reihenfolge eine Rolle. Bei ungeschickter Reihenfolge kann es passieren, dass Zwischenergebnisse nur mit eingeschränkter Genauigkeit berechnet werden. In Folge des damit verbundenen Fehlers führt die Auswertung nicht zu dem gewünschten Ergebnis.

Ausdruck	Ergebnis
`32 / 5 * 5`	`30`
`32 * 5 / 5`	`32`
`21 % 6`	`3  (21 - 3 * 6 )`
`28 % 7`	`0`
`21 / 6`	`3`
`23 / 6`	`3`

Man kann die Reihenfolge durch Klammern `()` verändern. Bei der Auswertung werden zunächst die Ausdrücke in Klammern berechnet. Klammern können geschachtelt werden. Die Berechnung beginnt dann bei der innersten Klammer. Dann wird die Klammer durch das Resultat ersetzt und die Berechnung fortgesetzt.

Übung 3.2.1. Welchen Wert ergeben die folgenden Ausdrücke?

`2 * 5 + 6 * 2`	
`2 * ( 5 + 6 * 2)`	
`2 * ( ( 5 + 6 ) * 2)`	

3.2.2 Inkrement- und Dekrement-Operator

In Java gibt es zwei spezielle Operatoren `++` und `--`, die eine Variable um 1 erhöhen oder vermindern. Diese Operatoren verändern den Operanden und erfordern daher einen lvalue. Sie können beispielsweise nicht auf Konstanten angewandt werden (`++5` ist nicht erlaubt). Der Ausdruck mit dem Operator ist selbst wieder ein rvalue (`++i = ...;` geht nicht). Ungewöhnlich ist, dass die Operatoren vor (präfix) oder nach (postfix) dem Operanden stehen können. Im ersten Fall wird die Variable verändert, bevor sie weiter verwendet wird. Im zweiten Fall wird erst der Wert benutzt, dann verändert.

Mit `n = 7` wird durch `i = ++n;` die Variable `i` auf `8` gesetzt, bei `i = n++;` auf `7`. In beiden Fällen hat `n` anschließend den Wert `8`. Man kann die Operatoren anwenden, ohne die Variable weiter zu benutzen, d. h. `++n;` oder `n--;` sind mögliche Anweisungen und gebräuchliche Abkürzungen für `n = n + 1;` bzw. `n += 1;` (siehe nächster Abschnitt) oder `n = n - 1;`. In diesem Fall spielt die Unterscheidung zwischen präfix und postfix keine Rolle. Ansonsten muss man bei komplizierteren Ausdrücken mit unbeabsichtigten Nebeneffekten rechnen. In jedem Fall leidet die Lesbarkeit des Programms. Gefährlich sind Querbezüge in der Art

```
i = 4;
i = i++ * 5;
```

In solchen Fällen ist es besser, eine Zeile mehr zu schreiben und damit klar darzustellen, was gemeint ist. Selbst wenn das Programm tatsächlich das ausführt, was die Programmiererin oder der Programmierer wollte, ist die kompakte Schreibweise schwerer zu verstehen. Es ist auch nicht zu erwarten, dass das entstehende Programm schneller läuft. In aller Regel wird der Compiler selbst durch Optimierung den effizientesten Code generieren.

3.2.3 Vereinfachte Zuweisung

Häufig hat man Zuweisungen in der Art

```
aktienImDepot = aktienImDepot + kauf;
```

d. h. die Variable auf der linken Seite wird auch als erster Operand auf der rechten Seite benutzt:

$$\text{expr1} = \text{expr1 op expr2}$$

Java bietet dafür bei den meisten Operatoren mit zwei Argumenten die kompakte Schreibweise

expr1 op= expr2

(ohne Leerzeichen zwischen op und =) an. Die wahrscheinlich am häufigsten in der Praxis auftretende Form ist die Verbindung mit der Addition oder Subtraktion: `i += 5;` Hauptvorteil ist die bessere Übersichtlichkeit. Insbesondere wenn der Ausdruck auf der rechten Seite komplex wird, ist die Bedeutung in dieser Schreibweise sofort ersichtlich. Die Schreibweise entspricht der Intention: `i` soll um 5 erhöht werden. Die Berechnung des Ausdrucks auf der rechten Seite hat dabei Vorrang. Das Beispiel

```
i *= j + 5;
```

wird als

```
i = i * (j + 5);
```

ausgewertet.

3.2.4 Bit-Operatoren △

Tief im Inneren des Rechners werden alle Daten binär dargestellt. Jede Speicherzelle enthält eine Reihe von Nullen und Einsen, den einzelnen Bits. Eine Bedeutung erhalten diese Bits erst durch den Datentyp. Damit legen wir fest, ob das Bitmuster eine Zahl, ein Zeichen oder was auch immer ist. Gelegentlich ist es notwendig oder zumindest hilfreich, das Bitmuster direkt zu bearbeiten. Dazu stellt Java eine Reihe von Bit-Operatoren bereit.

Ein Bit-Operator betrachtet demnach den Operanden als Folge von Bits. Diese Bitfolge kann manipuliert werden, indem beispielsweise alle Werte um eine gegebene Anzahl von Positionen verschoben werden. Bei der Verknüpfung zweier Operanden werden alle Bits Position für Position miteinander bearbeitet. Die Bit-Operatoren kann man auf ganzzahlige Daten anwenden. Im Einzelnen bietet Java folgende Bit-Operatoren:

Operator	Funktion	Anwendung
~	bitweises NICHT	~ausdruck
&	bitweises UND	ausdruck1 & ausdruck2
|	bitweises ODER	ausdruck1 | ausdruck2
^	bitweises EXOR	ausdruck1 ^ ausdruck2
<<	schieben nach links (shift)	ausdruck1 << ausdruck2
>>	schieben nach rechts (shift)	ausdruck1 >> ausdruck2

Das bitweise NICHT invertiert jedes Bit des Operanden. Die drei Operationen UND, ODER und EXOR verknüpfen entsprechend die einzelnen Bits. Im folgenden Beispiel werden zwei Byte (Datentyp `char`) mit UND verknüpft:

	Binär								Hex.	Dez.
	0	0	1	1	0	0	1	1	33	51
&	1	0	1	0	0	1	1	0	A6	166
	0	0	1	0	0	0	1	0	22	34

UND wird oft benutzt, um Bits auszuschneiden, während mit ODER Bits gezielt gesetzt werden können. Dazu nutzt man folgenden Beziehungen

$$\begin{aligned} b \text{ UND } 1 &= b \\ b \text{ UND } 0 &= 0 \\ b \text{ ODER } 1 &= 1 \\ b \text{ ODER } 0 &= b \end{aligned}$$

Wendet man die *UND Maske* auf einen Wert an, so bleiben demnach die Werte an Stellen mit 1 in der Maske erhalten, an den anderen Stellen werden sie 0. Umgekehrt führt bei der *ODER Maske* eine 1 in der Maske zu einer 1 im Ergebnis, während bei einer 0 der Wert übernommen wird. In Java lässt sich das in der Art

```
i = i & 0xF; // löscht alle außer den letzten vier Bits
i = i | 0xF0; // setzt vier Bits
```

schreiben. Die Shift-Operatoren verschieben den linken Ausdruck um die im rechten Ausdruck angegebene Anzahl von Stellen. Dabei wird beim Schieben nach links stets mit 0 aufgefüllt. Schiebt man nach rechts, so werden bei positiven Zahlen oder dem Datentypen `unsigned` in gleicher Weise die höherwertigen Stellen auf 0 gesetzt. Das Verhalten bei negativen Werten – d. h. Füllen mit 0 oder 1 – ist implementationsabhängig. Einige Beispiele werden im folgenden Code-Abschnitt gezeigt:

```
int i=15; // Bitmuster 0000 1111

System.out.printf("i= %4d (dez) %4o (oct) %4x (hex)\n", i, i, i);
// Schieben nach links, mit 0 auffuellen
i = i << 1; // Bitmuster 0001 1110
System.out.printf("i= %4d (dez) %4o (oct) %4x (hex)\n", i, i, i);

// Schieben nach rechts
i = i >> 2; // Bitmuster 0000 0111
System.out.printf("i= %4d (dez) %4o (oct) %4x (hex)\n", i, i, i);

// ODER mit 0111 0000
i = i | 0x70; // Bitmuster 0111 0111
System.out.printf("i= %4d (dez) %4o (oct) %4x (hex)\n", i, i, i);

// UND mit 0011 1111
```

```
i = i & 0x3f; // Bitmuster 0011 0111
System.out.printf("i= %4d (dez) %4o (oct) %4x (hex)\n", i, i, i);

// Umkehren aller Stellen
i = ~i; // Bitmuster 1100 1000
System.out.printf("i= %4d (dez) %4o (oct) %4x (hex)\n", i, i, i);
```

ergibt:

```
i=   15 (dez)   17 (oct)    f (hex)
i=   30 (dez)   36 (oct)   1e (hex)
i=    7 (dez)    7 (oct)    7 (hex)
i=  119 (dez)  167 (oct)   77 (hex)
i=   55 (dez)   67 (oct)   37 (hex)
i=  200 (dez)  310 (oct)   c8 (hex)
```

Die beiden Schrägstriche `//` leiten in Java einen Kommentar ein. Der Compiler ignoriert alles ab diesen Zeichen bis zum Zeilenende. Auch mehrzeilige Kommentare sind möglich. Sie werden mit `/*` eingeleitet und mit `*/` wieder beendet. Das folgende Beispiel dazu

```
/* ********************
   * erstes Beispiel *
   * s.e. August 2015 *
   ******************** */
```

könnte man gut als Kopf an den Anfang eines Programms schreiben.

3.3 Übungen

Übung 3.3.1. Welche Namen sind zulässige Bezeichner?

beta	ws01/02	___test___	α
ws2001_okt_08	ss2001-07-08	3eck	monDay
Uebung1.1	Uebung1_1	Uebung1A	Uebung1 A
$5	_8	ä	preisIn€

Übung 3.3.2. Integer Quiz
Die Variablen `i` und `j` sind vom Typ `int`. Welche Werte haben sie nach folgenden Anweisungen:

`i = 32 / 6 + 7 % 2;`	`i =`
`i = 7654; i = i / 100 * 10`	`i =`
`i = 6; j = 4; j = ++i * j;`	`i =  j =`
`i = 6; j = 4; j = i++ * j;`	`i =  j =`
`i = 5; j = 7; i -= j;`	`i =`
`i = 2; j = 3; i *= j + 1;`	`i =`

Übung 3.3.3. RGB
Wie beschrieben, entsteht eine Farbe im RGB-Farbraum durch Mischung der drei Anteile Rot, Grün und Blau. Jeder Anteil liegt zwischen 0 und 255. Als Farbwert f errechnet sich dann

$$f = r * 256^2 + g * 256 + b$$

Diese Farbwerte können in `int`-Variablen gespeichert werden.

- Definieren Sie drei Variablen für die einzelnen Anteile `r`, `g` und `b`. Wählen Sie sinnvolle Namen für die Variablen.

- Fragen Sie die einzelnen Werte in der Art

  ```
  int r = Dialogs.askInteger("Rot-Anteil", 0, 256);
  ```

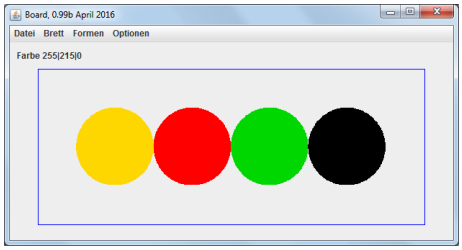
  ab[1]. Geben Sie zur Sicherheit diese Werte mit `println` aus.

- Nach der Eingabe soll jede einzelne Farbe in BoSym angezeigt werden.

- Lassen Sie den resultierenden Farbwert `f` berechnen.

- Geben Sie mit `printf` den Zahlenwert in `f` einmal als Dezimalzahl und einmal als Hexadezimalzahl (Formatbeschreiber `%x`) aus.

- Schließlich soll die Farbe in BoSym dargestellt werden. Verwenden Sie diesmal auch die Methode `statusText`, um eine sinnvolle Überschrift einzubauen. Ein Vorschlag:

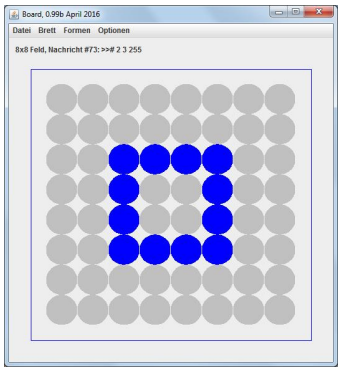


- Freiwillig: *Lassen Sie zusätzlich noch den zur Frabe gehörenden Grauwert darstellen.*

Testen Sie Ihren Code mit verschiedenen Farben.

[1] `Dialogs` ist eine Hilfsklasse in BoSym, Details siehe A.2

Übung 3.3.4. PlusPlus



Ein Rahmen soll gezeichnet werden. Vorgegeben ist der Anfang

```
int x=2, y=2;


farbe2( x, y, BLUE );
farbe2( ++x, y, BLUE );
```

Ergänzen Sie die weiteren Aufrufe von `farbe2`. Zusätzlich ist gefordert, dass dabei nur die Operatoren `++` und `--` benutzt werden.

Kapitel 4

Abläufe

In unserem letzten Beispiel hatten wir mit den Zeilen

```
int i = 11;
int linienFarbe = 255;

farbe( i, linienFarbe );
i = i + 1;
farbe( i, linienFarbe );
i = i + 1;
farbe( i, linienFarbe );
```

drei Kreise hintereinander gezeichnet. Das funktioniert, ist aber etwas umständlich. Wollten wir z. B. nicht nur drei sondern 30 Kreise malen, wäre die Wiederholung der gleichen Zeilen mühsam und unübersichtlich. Glücklicherweise bietet Java Konstruktionen, um mehrfache Wiederholungen kompakt zu formulieren. In diesem Kapitel werden wir die verschiedenen Möglichkeiten kennenlernen und dabei auch logische Ausdrücke als Basis der Entscheidung betrachten.

4.1 Zählschleife

Um eine Anweisung – oder auch mehrere Anweisungen – mit einem Zähler mehrfach zu wiederholen, benutzt man eine so genannte Zählschleife. In Java ist dies die for-Schleife. Unser Beispiel lässt sich dann als

```
int i;
int linienFarbe = 255;

for ( i=11; i<14; i=i+1 ) {
   farbe( i, linienFarbe );
}
```

schreiben. Die Konstruktion besteht aus dem Kopf mit den Informationen zur Ausführung sowie dem Schleifenrumpf (auch als Schleifenkörper bezeichnet) mit den zu wiederholenden Anweisungen. Der Schleifenrumpf kann eine einzige Anweisung oder ein Block in geschweiften Klammern sein. Es ist aber eine gute Idee, auch bei einer einzelnen Anweisung die Klammern zu schreiben. Dadurch wird die Struktur deutlicher erkennbar. Nun zum Kopf: er besteht aus dem Schlüsselwort `for` und dahinter in runden Klammern drei durch `;`-Zeichen getrennte Ausdrücke. Diese drei Ausdrücke haben folgende Bedeutung:

Initialisierung	`i=11`	wird vor dem ersten Durchlauf ausgeführt
Abfrage	`i<14`	wird vor jedem Durchlauf geprüft Abbruch, falls nicht erfüllt
Fortsetzung	`i=i+1`	wird am Ende jedes Durchlaufs ausgeführt

In unserem Beispiel wird `i` zunächst auf den Startwert 11 gesetzt. Dann wird – wie vor jedem weiteren Durchgang – geprüft, ob `i` noch kleiner als 14 ist. Mehr zu Vergleichen folgt später in diesem Kapitel. Falls ja, wird die Anweisung ausgeführt. Danach wird `i` um 1 erhöht und es geht wieder zurück zur Abfrage. Falls nein, wird die Schleife abgebrochen und das Programm geht mit der ersten Anweisung hinter dem Schleifenrumpf weiter.

Frage 4.1. Welchen Wert hat `i` nach der Schleife?

Die Variable behält einfach den letzten Wert. In unserem Fall wäre dies 14.

Frage 4.2. Prüft der Compiler, ob die Bedingungen in der Schleifen-Definition sinnvoll sind?

Nein, der Compiler kontrolliert nur die formale Struktur. Moderne Compiler erkennen allerdings bei der Optimierung Schleifen, die nie ausgeführt werden, und lassen sie dann einfach weg. Aber dies gilt nicht als Fehler. So akzeptiert der Compiler sowohl

```
for ( i=21; i<14; i=i+1 ) {
```

als auch

```
for ( i=0; i<14; i=i ) {
```

Im ersten Fall passiert gar nichts. Falls die Bedingung am Anfang nicht erfüllt ist, wird der Rumpf überhaupt nicht ausgeführt. Man spricht daher auch von einer abweisenden Schleife. Im zweiten Fall bleibt das Programm endlos in der Schleife.

Frage 4.3. Darf ich die Zählervariable innerhalb des Schleifenrumpfs ändern?

Ja, die Zählervariable hat keinen besonderen Schutz. Allerdings muss man aufpassen, dass der Code gut lesbar und verständlich bleibt. Die Beispiele verwenden die Standardform einer Zähl-Schleife. Allerdings können in den drei Komponenten von `for` beliebige Anweisungen stehen. Man könnte das Beispiel auch als

```
int i;
int linienFarbe = 255;

for ( i=11; i<14; farbe( i++, linienFarbe ) );
```

schreiben. Derartige Konstruktionen sind aber schwerer zu lesen und sollten daher nicht genutzt werden. Einzelne Komponenten können leer bleiben. Eine leere Kontrollabfrage gilt als immer wahr. Die Schreibweise `for ( ;; )` ist eine gängige Konstruktion für Endlosschleifen.

4.1.1 Verschachtelte Schleifen

Schleifen können ineinander verschachtelt werden. Bei jedem Durchlauf der äußeren Schleife wird die innere Schleife einmal komplett ausgeführt. Als Beispiel laufen in

```
for (int x = 2; x < 6; x++) {
  for (int y = 2; y < 6; y++) {
    farbe2(x, y, BLUE);
  }
}
```

die beiden Schleifen jeweils von 2 bis 5. Die Variablen `x` und `y` nehmen dabei nacheinander alle möglichen Kombinationen dieser Werte an:

$$(x, y) = (2,2), (2,3), (2,4), (2,5), (3,2), (3,3), \ldots, (5,5)$$

Im Ergebnis wird ein 4×4-Quadrat gezeichnet (Bild 4.1). Mit der Methode `farbe2` kann man Symbole über zwei Koordinaten ansprechen. Das System orientiert sich an der üblichen Notation bei rechteckigen Brettspielen. Die erste Koordinate bezeichnet die Spalte und die zweite die Zeile, wobei die Zählung von links nach rechts und von unten nach oben läuft. Allerdings verwendet man bei Brettspielen meistens Buchstaben für die Spalten. So bezeichnet A1 das Feld links unten und H8 das Feld rechts oben. Außerdem spricht man von waagerechten Reihen und vertikalen Linien. Wir bleiben bei der numerischen Bezeichnung und starten wieder bei 0. Damit wird A1 zu (0,0) und H8 zu (7,7).

Je nach Bedarf kann man im Menü *Optionen* zwischen linearer und zweidimensionaler Nummerierung umschalten. Bild 4.2 zeigt beispielhaft die Darstellung. Dabei wurde mit

```
farbe2( 3, 2, BLUE);
```

das Symbol an der Stelle $x = 3$ und $y = 2$ blau gefärbt. Zur Färbung der Symbole stehen damit die folgenden drei Methoden zur Auswahl:

`farben`	Plural	alle Symbole gleichzeitig
`farbe`	Singular	eine Koordinate, ein Symbol
`farbe2`	Singular, 2	zwei Koordinaten, ein Symbol

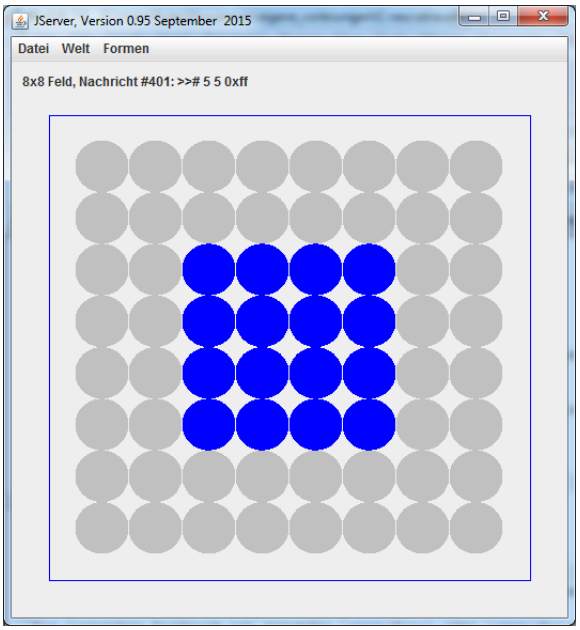


Abbildung 4.1: Quadrat durch verschachtelte for-Schleifen

Dieser Namenskonvention folgen auch die anderen BoSym-Methoden. Die jeweilige Plural-Variante wirkt auf alle Symbole gleichzeitig während die Singular-Varianten nur ein einzelnes Symbol ansprechen. Allerdings sind nicht für alle Methoden alle drei Varianten implementiert.

Zurück zu den verschachtelten Schleifen: die Schleifen können natürlich unterschiedlich lang sein. So ergibt

```
for ( int x=2; x<4; x++ ) {
   for ( int y=2; y<8; y++ ) {
     farbe2( x, y, BLUE );
   }
}
```

ein Rechteck. Es ist auch durchaus möglich, sich in der inneren Schleife auf den Zähler der äußeren Schleife zu beziehen. So durchlaufen in

```
for ( int x=0; x<8; x++ ) {
   for ( int y=0; y<=x; y++ ) {
     farbe2( x, y, BLUE );
   }
}
```

die Variablen die Wertepaare

$$(x, y) = (0,0), (1,0), (1,1), (2,0), (2,1), (2,2), \ldots, (7,7)$$

und ein Dreieck wird gezeichnet (Bild 4.3)

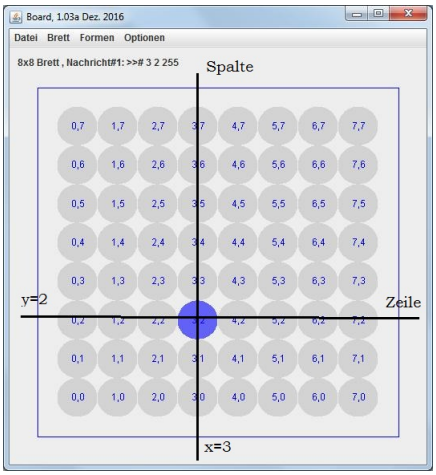


Abbildung 4.2: Quadrat mit xy-Koordinaten

Übung 4.1.1. Doppelte Schleife
Was bewirkt folgende Konstruktion:

```
int x, y;

for ( x=0, y=0; x<8; x++, y++ ) {
    farbe2( x, y, BLUE );
}
```

4.2 Logische Ausdrücke

Die Entscheidung über einen weiteren Schleifendurchlauf wurde mit dem Ausdruck `i<14` getroffen. Das ist ein einfaches Beispiel für einen logischen Ausdruck. Ein logischer Ausdruck ist entweder *wahr* oder *falsch*. In Java gibt für logische Werte den Typ `boolean`[1]. Variablen vom Typ `boolean` haben entweder den Wert `true` oder `false`. Man kann den logischen Wert als Konstante oder als Ergebnis eines Ausdrucks angeben:

```
int i = 6;
boolean b1 = i < 5;
boolean b2 = true;
System.out.println("b1 = " + b1);
System.out.println("b2 = " + b2);
```

ergibt

[1] benannt nach George Boole, engl. Mathematiker 1815-1864. Er gilt als Begründer der mathematischen Logik.

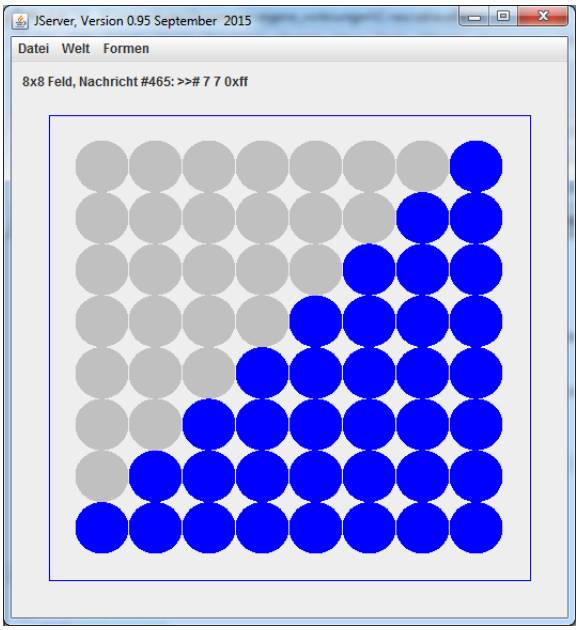


Abbildung 4.3: Dreieck durch verschachtelte for-Schleifen

```
b1 = false
b2 = true
```

Tabelle 4.1 enthält eine vollständige Liste der Vergleichsoperatoren. Damit können Ausdrücke miteinander verglichen werden. Mehrere mit diesen Vergleichsoperatoren gebildete logische Ausdrücke können durch logische Operatoren verknüpft werden. Die logischen Operatoren sind in Tabelle 4.2 zusammengestellt. So bezeichnet `&&` die Und-Verknüpfung zweier Ausdrücke und `||` die Oder-Verknüpfung. Beispiele sind

```
n > 5 && n < 10
i == 2 || i == 4 || i == 6
```

Es werden wieder die gleichen Zeichen verwendet wie bei den entsprechenden Bit-Operatoren. Eine Verwechslungsgefahr besteht nicht, da Java anhand des Typs der Operanden erkennt, ob eine logische oder bitweise Verknüpfung gemeint ist. Weiterhin kann durch `!` (NICHT) ein Ausdruck negiert werden. Damit ist es möglich, die Rechenregeln der Booleschen Algebra anzuwenden, um beispielsweise Ausdrücke umzuformen.

Übung 4.2.1. Logische Operatoren.
Wie kann man in Java die folgende Bedingungen abfragen? Getestet werden soll, ob

Tabelle 4.1: Vergleichsoperatoren in Java

`==`	gleich
`!=`	nicht gleich
`>`	größer
`>=`	größer gleich
`<`	kleiner
`<=`	kleiner gleich

Tabelle 4.2: Logische Operatoren in Java

`!`	Nicht
`&`	Und
`|`	Oder
`^`	exklusives Oder
`&&`	Und mit Short-Circuit-Evaluation
`||`	Oder mit Short-Circuit-Evaluation

	if( )
eine Variable `i` ein ganzzahliges Vielfaches von 5 ist	
eine Variable `x` im Bereich $[10, 20]$ liegt (einschließlich der Grenzen)	
eine Variable `i` kleiner als -10 oder größer als +10 ist	
mindestens eine der Variablen `x` oder `y` einen positiven Wert (> 0) enthält	
ein Punkt mit den Koordinaten (x,y) innerhalb eines Kreises mit dem Mittelpunkt (0,0) und dem Radius 2 liegt	

Eine Besonderheit in Java ist die Unterscheidung zwischen Operatoren mit und ohne so genannter *Short-Circuit-Evaluation*. Bei der Short-Circuit-Evaluation nutzt man die Eigenschaften der Operatoren zu einer eventuell verkürzten Auswertung aus. Bei einer Und-Verknüpfung müssen beide Operanden wahr sein, damit der Ausdruck selbst auch wahr ist. Stellt man nun fest, dass der erste Operand den Wert *falsch* hat, so steht bereits fest, dass auch der gesamte Ausdruck den Wert *falsch* haben wird. Auf die Auswertung des zweiten Ausdrucks kann dann verzichtet werden. Dadurch kann Rechenzeit gespart werden. Kompliziert wird das Verhalten, wenn bei der Auswertung des zweiten Ausdrucks Nebenwirkungen auftreten. Betrachten wir das Beispiel

```
int n = 4;
System.out.println( n > 5  &&  n++ < 10 );
```

```
System.out.println( "n = " + n);
System.out.println( n > 5  &  n++ < 10 );
System.out.println( "n = " + n);
```

Die Bedingung `n > 5` ist nicht erfüllt. Mit Short-Circuit-Evaluation wird der zweite Ausdruck nicht ausgewertet. Dementsprechend behält die Variable `n` ihren alten Wert. Ohne Short-Circuit-Evaluation wird auch der zweite Ausdruck berechnet und der Wert von `n` erhöht. Das Programm liefert demnach die Ausgabe

```
false
n = 4
false
n = 5
```

Man kann dieses Verhalten gezielt einsetzen und damit eine Art verkürzte Abfrage realisieren. Diesen Stil findet man in anderen Programmiersprachen wie C oder Perl recht häufig. Allerdings sollte man bei der Verwendung vorsichtig sein und sich auf wenige, klar verständliche Fälle beschränken.

4.3 if-Abfrage

Die einfache Fallunterscheidung ist durch die `if-else`-Anweisung realisiert. Die allgemeine Form ist:

```
if( ausdruck ) {
    anweisung1
} else {
    anweisung2
}
```

In der runden Klammer steht ein logischer Ausdruck. Abhängig von seinem Wert wird bei Resultat *wahr* der erste Block ausgeführt, ansonsten der zweite. Der durch `else` eingeleitete Block ist optional und kann entfallen. Betrachten wir als Beispiel eine Schleife, die ein 8×8 Brett füllt. Abhängig vom Zähler werden die Felder gefüllt.

```
for ( int i=0; i<64; i++ ) {
   if( i < 10 ) {
       farbe(i, WHITE);
   } else {
       farbe(i, BLACK);
   }
}
```

Das Result zeigt Bild 4.4. Die ersten 10 Felder sind weiß, alle anderen schwarz. Wenn ein Block nur aus einer einzigen Anweisung besteht, kann man die geschweiften Klammern weglassen. Das Beispiel kann auch als

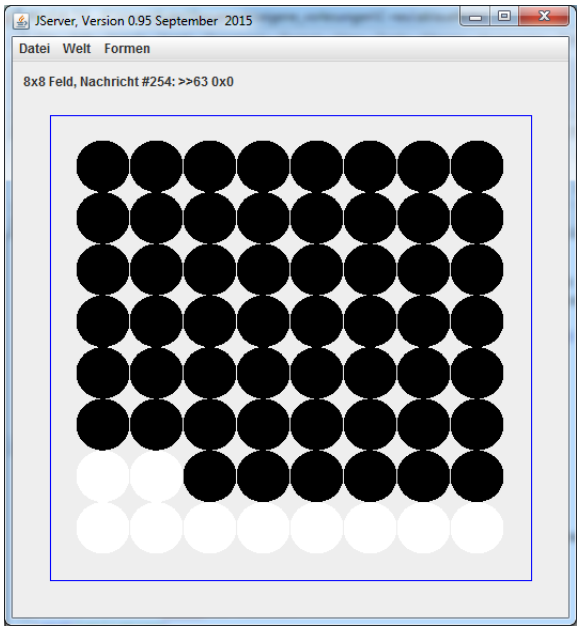


Abbildung 4.4: Ergebnis bei Abfrage `if( i < 10 )`

```
if( i < 10 ) farbe(i, WHITE);
else farbe(i, BLACK);
```

geschrieben werden. Die Form ohne Klammern sollte allerdings nur für einfache und übersichtliche Fälle benutzt werden. Die Blöcke können auch leer sein. So kann man die Berechnung des Absolutbetrags als

```
if( x < 0 ) {
    x = -x;
} else {
}
```

schreiben. Ein leerer Else-Block kann weggelassen werden. Aus dem Beispiel wird dann

```
if( x < 0 ) {
    x = -x;
}
```

4.3.1 Else-If

Häufig möchte man mehr als zwei Fälle unterscheiden. If-Abfragen lassen sich beliebig ineinander verschachteln. Wir können unser Beispiel erweitern und bei 20 einen weiteren Farbwechsel einbauen:

```
for ( int i=0; i<64; i++ ) {
```

```
   if( i < 10 ) {
      farbe(i, WHITE);
   } else {
      if( i < 20 ) {
         farbe(i, BLACK);
      } else {
         farbe(i, BLUE);
      }
   }
}
```

Allerdings wird die Gesamtstruktur durch die zunehmende Einrücktiefe schnell etwas unübersichtlich. Daher gibt es für die Fallunterscheidung eine spezielle Form mit `else if`:

```
if( ausdruck1 ) {
    anweisung1
} else if( ausdruck2 ) {
    anweisung2
} else if( ausdruck3 ) {
    anweisung3
} else {
    anweisung4
}
```

In dieser Form werden mehrere Überprüfungen geschachtelt nacheinander ausgeführt. Sobald eine Bedingung erfüllt ist, wird der zugehörige Block ausgeführt und die ganze Kette verlassen (selbst wenn weitere Bedingungen erfüllt wären). Der letzte (optionale) `else`-Block behandelt dann alle verbleibenden Fälle, die von keiner Bedingung abgedeckt sind. Wir erweitern unser Beispiel auf vier Fälle:

```
if( i < 10 ) {
    farbe(i, WHITE);
} else if( i < 20 ) {
    farbe(i, BLACK);
} else if( i < 30 ){
    farbe(i, BLUE);
} else {
    farbe(i, GREEN);
}
```

Wie Bild 4.5 zeigt, werden die ersten drei Zehnerblöcke auf die angegebenen Farben gesetzt, alle restlichen auf Grün.

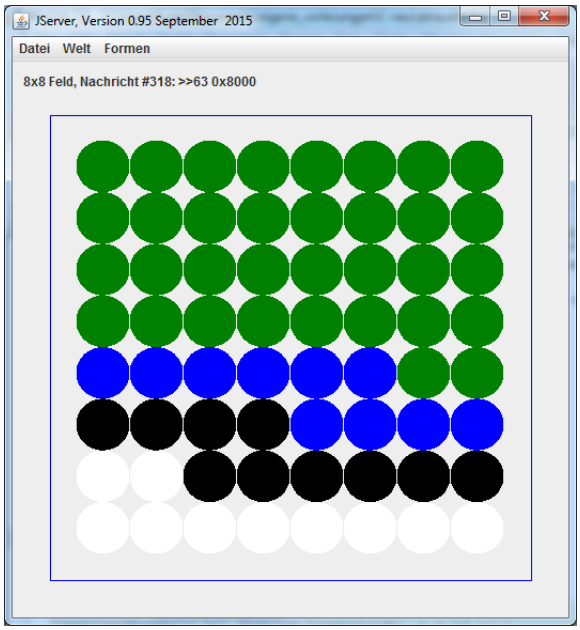


Abbildung 4.5: Ergebnis bei Else-If Abfrage

4.3.2 Fragezeichen-Operator △△

Die Auswahl zwischen zwei Alternativen kann kompakt mit dem ?-Operator geschrieben werden. Die allgemeine Syntax ist

ausdruck ? alternative_ja : alternative_nein

Ist der Ausdruck wahr, so wird die erste Alternative, ansonsten die zweite ausgeführt. Als Beispiel ist

```
p = ( x > 0 ) ? x : -x;
```

eine kompakte Alternative zu

```
if( x > 0 ) {
    p = x;
} else {
    p = -x;
}
```

um den Absolutbetrag von `x` berechnen. Die Verwendung des ?-Operators ist Geschmackssache. Man kann damit kompakten und eleganten Code schreiben, aber zumindest für Anfänger ist eine ausführliche Formulierung über if-else übersichtlicher.

4.4 Mehr zu Schleifen

Neben der for-Schleife gibt es in Java noch zwei weitere Schleifentypen. Die wohl einfachste Schleife wird mit `while` eingeleitet:

```
while( ausdruck ) {
    anweisung
}
```

Der Ausdruck in der `while` Anweisung wird zunächst ausgewertet. Ergibt er *wahr*, so werden die Anweisungen im Block ausgeführt (vorausgehende Bedingungsprüfung). Anschließend wird die Bedingung wieder überprüft und der Block gegebenenfalls erneut ausgeführt. Ein Beispiel dazu

```
i = 0;
while( i < 10 ) {
    farbe( i, RED );
    ++i;
}
```

Die Schleife wird durchlaufen, bis der Wert von `i` größer als 10 wird. Bei der dritten, relativ selten benutzte Form wird die Bedingung erst nach der Ausführung des Blocks geprüft. (nachfolgende Bedingungsprüfung). Die allgemeine Form ist

```
do {
    anweisungen
} while( ausdruck );
```

In diesem Fall wird die Schleife immer mindestens einmal durchlaufen (nicht abweisende Schleife). Erst am Ende der ersten Schleife wird die Bedingung zum ersten Mal überprüft. Dieser Art von Abfrage ist dann sinnvoll, wenn die Bedingung erst nach der Ausführung ausgewertet werden kann. Man kann diese Form verwenden, um die Korrektheit von Eingaben zu prüfen. In diesem Fall muss die Schleife einmal durchlaufen werden, um überhaupt einen Wert zu haben. Ein Beispiel:

```
int z = 0;

do {
    z = Dialogs.askInteger( "Positive Zahl" );
} while( z <= 0 );
```

In Tabelle 4.3 sind die drei Grundtypen von Schleifen zusammen mit der Java-Syntax zusammen gestellt. Grundsätzlich kann man noch die Wiederholung ohne Prüfung als eigenen Typ betrachten. Derartige Schleifen werden endlos wiederholt. Einsatzgebiet sind Prozesse, die ständig auf Ereignisse warten. Für Endlosschleifen gibt es in Java keine gesonderte Konstruktion. Üblich sind die Formen `for (;;)` und `while( true )`.

Tabelle 4.3: Übersicht Schleifen

Typ		C-Syntax
Wiederholung mit vorausgehender Prüfung (Kopfprüfung)	abweisend	`while(){}`
Wiederholung mit nachfolgender Prüfung (Fußprüfung)	nicht abweisend	`do{}while()`
Zählschleife	abweisend	`for (;;){}`

4.4.1 Vorzeitiges Verlassen von Schleifen △

In manchen Fällen ist es erforderlich, eine Schleife entweder ganz zu verlassen oder zumindest den aktuellen Durchgang abzubrechen. In Java gibt es dazu die Anweisungen `break` und `continue`. Ein `break` beendet die Schleife komplett, der Programmablauf wird mit der ersten Anweisung hinter der Schleife fortgesetzt. Demgegenüber bleibt bei einem `continue` das Programm in der Schleife, nur der aktuelle Durchgang wird nicht zu Ende geführt. Statt dessen wird bei einer `while`- oder `do`-Schleife die Abbruchbedingung ausgewertet. Bei einer `for`-Schleife wird das Programm mit der Ausführung des dritten Ausdrucks fortgesetzt.

Beispiel 4.1. break-Anweisung

```
int i, summe = 0;
for ( i=0; ; i++ ) {
    if( ( summe += i ) > 1000 ) break;
}
System.out.println("Summe " + summe + " bei i=" + i + " erreicht");
```

In diesem Programm wird in der Abfrage ausgenutzt, dass auch eine Zuweisung einen Ausdruck darstellt. Der Wert der Zuweisung ist genau der zugewiesene Wert. Beispielsweise hat die Zuweisung

```
i = 4 * 5;
```

den Wert (rvalue) 20.

Weiterhin können die Befehle `break` und `continue` mit einer Marke versehen werden. Der Name der Zielmarke oder Sprungmarke (engl. *label*) wird nach den Regeln für Variablen gebildet. Sie muss wiederum eine der umgebenden Kontrollstrukturen markieren. Dazu wird der Name gefolgt von einem Doppelpunkt vor die entsprechende Schleife gesetzt. Als Beispiel kann man schreiben:

```
f1: for ( ...) {
    for ( ...) {
        for ( ...) {
            if( katastrophe ) break f1;
...
```

In diesem Fall beendet das `break` die mit `f1` markierte umgebende Schleife. Allerdings enthält Java wesentlich leistungsfähigere Konzepte zur Fehlerbehandlung. Die Bedeutung von gelabelten `break`- und `continue`-Anweisungen ist daher nicht sehr groß.

4.5 Fallunterscheidung mit der switch-Anweisung

Wenn man eine Auswahl aus vielen einander sich gegenseitig ausschließenden Alternativen treffen will, werden `if ... else if`-Strukturen recht unübersichtlich. Als Vereinfachung stellt Java die Möglichkeit der `switch`-Anweisung (engl. für Schalter) zur Verfügung. Sie hat die Form

```
switch( ausdruck ) {
    case konst1:
        anweisung1
    case konst2:
        anweisung2
    default:
        anweisung3
}
```

In der Klammer steht ein Ausdruck, der einen ganzzahligen Wert liefert[2]. Dieser wird dann mit den Konstanten an den `case` (engl. Fall) Marken verglichen. Bei Gleichheit wird das Programm an dieser Stelle fortgesetzt. Etwas gewöhnungsbedürftig ist die Tatsache, dass die Ausführung nicht automatisch durch das nächste `case` beendet wird. Soll – wie es in der Regel der Fall ist – die Bearbeitung der `switch`-Anweisung nach einer Übereinstimmung verlassen werden, muss dies explizit durch ein `break` festgelegt werden. Das optionale `default` „sammelt“ alle Fälle, in denen keine Übereinstimmung gefunden wurde. Im folgenden Beispiel wird für eine Zahl zwischen 1 und 7 der zugehörige Wochentag ausgegeben.

Beispiel 4.2. Ausgabe des Wochentages.

```
int wochentag = Dialogs.askInteger("Wochentag");
switch( wochentag ) {
    case 1:
        System.out.println("Montag" );
        break;
    case 2:
        System.out.println("Dienstag" );
        break;
    case 3:
        System.out.println("Mittwoch" );
```

[2]Seit Version Java SE 7 kann man switch-Anweisungen auch mit Zeichenketten verwenden.

```
        break;
    case 4:
        System.out.println("Donnerstag" );
        break;
    case 5:
        System.out.println("Freitag" );
        break;
    case 6:
        System.out.println("Samstag" );
        break;
    case 7:
        System.out.println("Sonntag" );
        break;
    default:
        System.out.println("Kein gültiger Wochentag");
}
```

Die Reihenfolge der `case`-Marken spielt – solange sie mit einem `break` beendet werden – keine Rolle. Möchte man mehrere Fälle zusammenfassen, kann man die entsprechenden `case`-Marken direkt aufeinander folgen lassen. Man könnte etwa im obigen Beispiel schreiben

```
    case 6: case 7:
        System.out.println("Wochenende" );
        break;
```

Mit Version 14 wurden weitere Möglichkeiten bei switch-Anweisungen eingeführt. Damit ist es unter anderem möglich, mehrere Fälle kompakt in einem `case`-Block zu sammeln. Als etwas ausführlicheres Beispiel betrachten wir die Generierung eines Spielplans. Dabei sollen die einzelnen Felder je nach Feldtyp unterschiedlich gestaltet werden. Listing 4.1 zeigt eine entsprechende Lösung. Dabei sind drei Feldtypen vorgesehen. Zur besseren Lesbarkeit sind für diese Typen Konstanten mit passenden Namen definiert. Die Zahlenwerte selbst haben keine Bedeutung und dienen nur zur Identifikation[3]. Mit zwei Schleifen werden alle Felder durchlaufen. Bei jedem Feld wird zufällig einer der drei Typen ausgewählt. Die Methode `Math.random()` liefert eine Zufallszahl zwischen 0 und 1. Durch Multiplikation mit 3 und Umwandlung in eine ganze Zahl erhält man den Zufallswert 0, 1, oder 2. In der `switch`-Anweisung wird der entsprechende Fall ausgewählt. Dort wird dann die passende Formatierung durchgeführt. Bild 4.6 zeigt ein Resultat.

[3]Java bietet für solche Fälle einen Aufzählungstyp `enum` an. Das erlaubt insgesamt besseren Code. Die Switch-Anweisung bleibt aber gleich.

Listing 4.1: Switch-Anweisung zum Füllen eines 10×10-Feldes

```
final int FELD = 0;
final int WALD = 1;
final int WASSER = 2;

int N = 10;
groesse( N, N );

for ( int x=0; x<N; x++ ) {
   for ( int y=0; y<N; y++ ) {
      int typ = (int)(Math.random()*3);
      switch( typ ) {
      case FELD:
         form2( x, y, "s" );
         farbe2( x, y, LAWNGREEN );
         break;
      case WALD:
         form2( x, y, "c" );
         hintergrund2( x, y, LAWNGREEN );
         farbe2( x, y, DARKGREEN );
         break;
      case WASSER:
         form2( x, y, "s" );
         farbe2( x, y, BLUE );
         break;
      }
   }
}
```

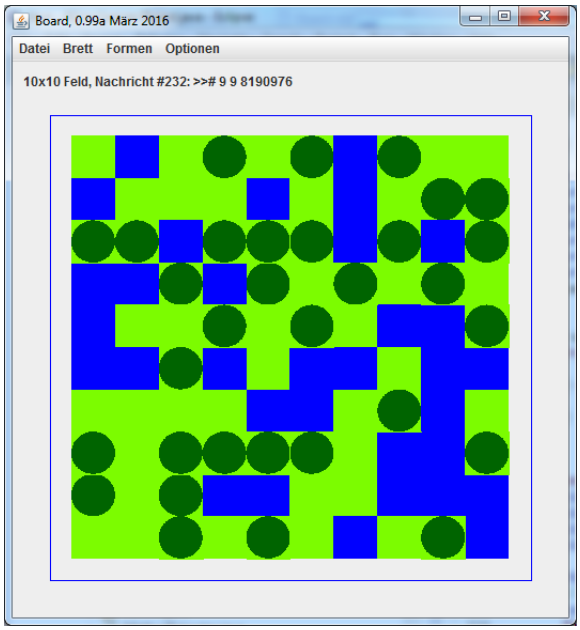


Abbildung 4.6: Zufalls-Landschaft

4.6 Rangfolge der Operatoren

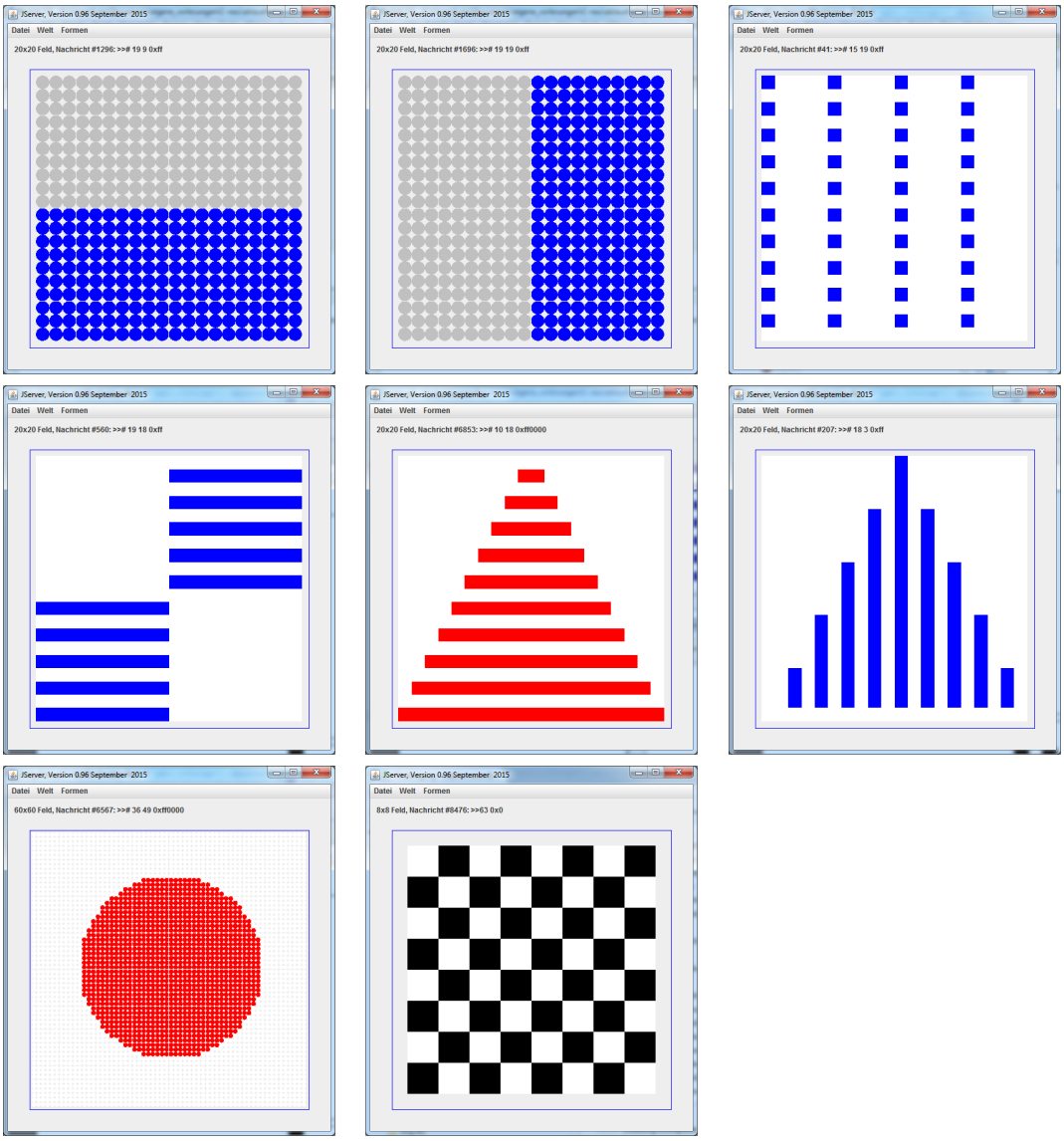
In Integer-Ausdrücken können arithmetische und logische Operatoren, Vergleichsoperatoren, u. s. w. auftreten. Die Reihenfolge der bisher behandelten Operatoren ist in Tabelle 4.4 zusammengestellt. Je weiter oben ein Operator in der Tabelle steht, desto höher ist sein Rang oder man sagt, er bindet stärker. Am stärksten binden die Operatoren, die nur einen Operanden haben. Gleichberechtigte Operatoren werden von links nach rechts abgearbeitet. Schließlich kann man mit runden Klammern angeben, dass ein Teilausdruck zunächst als Ganzes bearbeitet wird.

4.7 Übungen

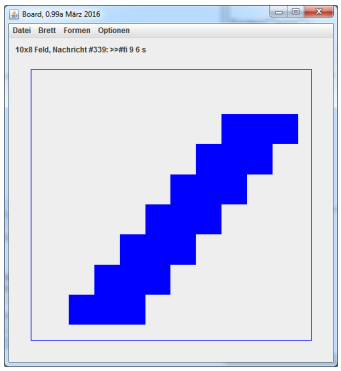
Übung 4.7.1. Muster
Schreiben Sie Anweisungen, um folgende Muster zu erzeugen:

Tabelle 4.4: Reihenfolge der Operatoren

`++, --`	Inkrement, Dekrement
`~`	bitweise NICHT
`!`	logisches NICHT
`+, -`	Vorzeichen
`*, /, %`	multiplikative Operatoren
`+, -`	Addition und Subtraktion
`<<, >>`	Bit shift
`<, <=, >=, >`	Vergleichsoperatoren
`==, !=`	Gleichheit, Ungleichheit
`&`	bitweises UND
`^`	bitweises XOR
`|`	bitweises ODER
`&&`	logisches UND
`||`	logisches ODER



Übung 4.7.2. Treppe



Wie kann man das Treppen-Muster mit zwei ineinander geschachtelten for-Schleifen generieren?

Übung 4.7.3. Fibonacci-Zahlen
Die Fibonacci[4]-Zahlen sind durch die Startbedingung $i_1 = 1$, $i_2 = 1$ und die Rekursion $i_n = i_{n-2} + i_{n-1}$ definiert. Damit gilt

$$\begin{aligned} i_3 &= i_1 + i_2 = 1 + 1 = 2 \\ i_4 &= i_2 + i_3 = 1 + 2 = 3 \\ i_5 &= i_3 + i_4 = 2 + 3 = 5 \\ i_6 &= i_4 + i_5 = 3 + 5 = 8 \end{aligned}$$

und so weiter. Geben Sie alle Fibonacci-Zahlen kleiner als 30000 aus. Welchem Wert nähert sich der Quotient i_{n+1}/i_n mit wachsendem n an?

Übung 4.7.4. Sie betreiben einen Internet-Handel. Bei jedem Kunden zählen Sie die Anzahl der Einkäufe. Abhängig von dieser Zahl gelten Kunden als

Neuling	bis 5 Einkäufe
Kunde	bis 50 Einkäufe
Stammkunde	bis 500 Einkäufe
Gold Kunde	ab 501 Einkäufe

Wie sieht eine `if else if` Abfrage aus, um den Status eines konkreten Kunden in Abhängigkeit von der Anzahl der Käufe auszugeben?

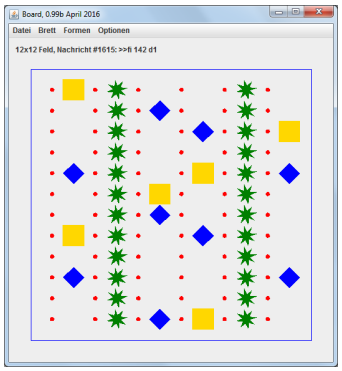
Übung 4.7.5. if-else

[4]Leonardo Pisano genannt Fibonacci, italienischer Mathematiker, ca. 1170-1250

Schreiben Sie nach dem Beispiel im Skript (Abschnitt 4.3.1) eine Schleife mit if-else Abfrage um Vielfache zu markieren. Vorschlag:

- Vielfache von 2: rote Punkte (Form `d1`)
- Vielfache von 3: grüne Sterne
- Vielfache von 5: blaue Rauten
- Vielfache von 7: gelbe Quadrate

Dabei gilt jeweils der erste Faktor. Zum Beispiel wird 6 als Vielfaches von 2 markiert und nicht mehr als Vielfaches von 3.



Kapitel 5

Gleitkommazahlen

5.1 Gleitkomma-Darstellung

Bisher hatten wir Zahlen in der Integerdarstellung betrachtet. Diese Darstellung von ganzen Zahlen und das Rechnen damit ist exakt solange man

- im darstellbaren Zahlenbereich bleibt
- keine Nachkommstellen betrachtet (z. B. nach Division).

In vielen praktischen Anwendungen benötigt man aber eine flexiblere Repräsentation von Zahlen. Oft ist das Rechnen mit Nachkommastellen notwendig oder zumindest natürlich. Viele Angaben enthalten Nachkommastellen (Zinssatz 3,5%, 4,7 Liter auf 100 km). Die erste Erweiterung ist die Einführung von Nachkommastellen. Bei der Festkommadarstellung gibt man die Anzahl der Vor- und Nachkommastellen fest vor. Als Nachteil bleibt dabei die eingeschränkte Dynamik des Zahlenbereichs. Daher wird diese Darstellung nur in wenigen Spezialanwendungen verwendet.

Auch im Alltag und noch mehr in der Technik haben wir das Problem der unterschiedlichen Bereiche. Bei Längen beispielsweise kann je nach Anwendung eine Angabe in mm (Schrauben im Baumarkt) oder in km (Urlaubsreise) sinnvoll sein. Der Trick dabei ist, dass die Länge mit einer Anzahl von signifikanten Stellen und der Größenordnung angegeben wird: 3,5 mm oder 650 km. Vollständige Genauigkeit 650.245.789 mm ist weder sinnvoll noch notwendig. Allgemein schreibt man eine Größe z als Produkt der Mantisse M und einer ganzzahligen Potenz p von 10:

$$z = M \cdot 10^p$$

etwa 3,5 10^{-3} m. Für Maßangaben gibt es Namen bzw. Vorsilben für die entsprechenden Potenzen von 10 (Kilo, Giga, Mega, Nano, etc). Zum Rechnen muss man Zahlen auf eine einheitliche Darstellung normieren:

$$3{,}5\text{mm} + 650\text{km} = 0{,}003.5\text{m} + 650.000\text{m} = 650.000{,}003.5\text{m}$$

Die Beispiele zeigen, wie man den Exponenten durch Verschieben des Kommas in der Mantisse verändert. Es gibt für eine gegebene Zahl (unendlich) viele gleichwertige Darstellungen:

$$123 = 12,3 \cdot 10 = 1,23 \cdot 10^2 = 1230 \cdot 10^{-1} = \ldots$$

Eine einheitliche Darstellung erreicht man mit der Vereinbarung, dass die Mantisse nur genau eine Vorkommastelle hat. Damit hat man eine eindeutige Abbildung zwischen der Zahl z und ihrer Darstellung durch Mantisse m und Exponent p:

$$z \Leftrightarrow (m, p)$$

Die Position des Kommas wird je nach Bedarf verschoben, man nennt daher dieses Format Gleitkommadarstellung (floating point) oder auch halblogarithmische Darstellung. Für die Verwendung in Computern geht man zum Dualsystem über, so dass p dann der Exponent zur Basis 2 ist. In der normalisierten Darstellung wird das Komma so gesetzt, dass nur eine Vorkommastelle bleibt. Die Mantisse hat dann im Dualsystem – außer bei der Null – immer die Form

$$m = 1, \ldots$$

Da die führende 1 bei jeder Zahl steht, braucht man sie in der Realisierung nicht abzuspeichern und gewinnt dadurch eine weitere Stelle. In der Praxis sind für m und p nur endlich viele Bits verfügbar. Die Größe von m bestimmt die Genauigkeit der Zahlendarstellung und die Größe von p legt den insgesamt abgedeckten Zahlenbereich fest. Aufgrund der endlichen Größe von m und p kann es zu folgenden Fehlern in der Repräsentation kommen:

- Die Zahl ist zu groß oder zu klein ($z \geq 2^{pmax+1},\ z \leq -2^{pmax+1}$).
- Die Zahl ist betragsmäßig zu klein ($|z| < 2^{-pmax}$ bei symmetrischem Zahlenbereich des Exponenten).
- Die Mantisse ist nicht groß genug, um die erforderliche Anzahl von Stellen zu repräsentieren (Rundungsfehler).

Übung 5.1.1. Gleitkomma-Darstellung
Betrachten wir folgende Zahlendarstellung im Dezimalsystem: $\pm x.xxx \cdot 10^{\pm ee}$. Geben Sie dazu folgende Werte an:

- Die kleinste Zahl
- Die größte Zahl
- Die betragsmäßig kleinste Zahl $\neq 0$
- Den Abstand zwischen den beiden größten Zahlen

- Den Abstand zwischen den beiden betragsmäßig kleinsten Zahlen

Insbesondere die Rundungsfehler bedingen, dass eine reelle Zahl im allgemeinen nicht genau dargestellt werden kann. Berechnet man etwa 1/3, so ist das Resultat 0,33333... prinzipiell nicht exakt darstellbar. Für die Repräsentierung einer Gleitkommazahl benötigt man im Detail:

- Die Mantisse
- Das Vorzeichen der Mantisse
- Den Exponent
- Das Vorzeichen des Exponents

Weiterhin muss festgelegt werden, wie die insgesamt verfügbaren Bits auf Mantisse und Exponent aufgeteilt werden. Lange Zeit waren die Details der Implementierung von Gleitkommazahlen herstellerabhängig. Da dies zu Problemen beim Datenaustausch und auch bei der Kompatibilität von Programmen führen kann, wurde von Normierungsgremien des IEEE (Institute for Electrical and Electronics Engineers, www.ieee.org) ein Standard verabschiedet. Dieser Standard mit der Bezeichnung IEEE 754[1] definiert 3 Formate:

short real	32 Bit	einfache Genauigkeit
long real	64 Bit	doppelte Genauigkeit
temporary real	80 Bit	erweiterte Genauigkeit

Als Beispiel betrachten wir das Format short real näher:

Bit 31		Bit 0
Vz	Charakteristik c	Mantisse m
1 Bit	8 Bit	23 Bit

mit

Vz	Vorzeichen der Mantisse (0 positiv, 1 negativ)
Charakteristik	Exponent + 127
Mantisse	Nachkommastellen

Im Gegensatz zu der bei Integer üblichen Darstellung mit dem 2er-Komplement wird die Mantisse mit Betrag und getrenntem Vorzeichen abgelegt (Vorzeichen-Betrag-Darstellung). In der Charakteristik wird der um 127 verschobene Exponent (biased exponent) eingetragen. Die Zahl $-37,125_{10}$ soll beispielhaft in das Format short real gebracht werden.

[1] genauer: IEEE Standard for Binary Floating-Point Arithmetic for microprocessor systems (ANSI/IEEE Std 754-1985)

1. Konvertierung in Binärdarstellung: $100101,001$ $(32 + 4 + 1 + 1/8)$
2. Normalisierung: $1,00101001 \cdot 2^5$
3. Mantisse: 00101001
4. Charakteristik: $5 + 127 = 132_{10} = 10000100_2$
5. Vorzeichen: 1 für negative Zahl

Bit 31		Bit 0
1	1000010 0	0010100 10000000 00000000
1 Bit	8 Bit	23 Bit

Die Werte 0 und 255 sind reserviert, um den Wert Null sowie einige Spezialfälle darstellen zu können. Die Null ist ein Sonderfall, da für sie keine normalisierte Darstellung mit einer Eins vor dem Komma möglich ist. Daher wurde vereinbart, dass die Null durch ein Löschen aller Bit in Charakteristik und Mantisse dargestellt wird. Im Detail gilt:

	Vz	Charakteristik	Mantisse
Nicht normalisiert	$\pm$	0	$\neq 0$
Null	$\pm$	0	0
Unendlich (Inf)	$\pm$	255	0
Keine Zahl (NaN)	$\pm$	255	$\neq 0$

Mit den beiden Werten Inf und NaN können Fehlerfälle abgefangen werden. Bei Bereichsüberschreitung wird das Result auf Inf gesetzt, während NaN durch „unerlaubte“ Operationen wie Division von 0 durch 0 oder Wurzel aus einer negativen Zahl entsteht. Damit besteht einerseits die Möglichkeit, solche Fälle zu erkennen. So steht beispielsweise in C die Funktion `_isnan` zur Verfügung, um auf NaN zu testen. Andererseits kann man mit den Werten auch weiter rechnen. Der Standard spezifiziert das Ergebnis von Operationen wie Inf + 10 = Inf. In manchen Algorithmen kann man damit alle Abfragen auf Sonderfälle vermeiden, die sonst den linearen Programmablauf stören würden. Eine ausführliche Darstellung der Thematik enthält der Artikel von Goldberg [Gol91].

Bei den beiden größeren Typen long real und temporary real wird sowohl die Genauigkeit erhöht als auch der Wertebereich erweitert. Rechnen mit Gleitkommazahlen bedeutet einen wesentlich größeren Aufwand als das Rechnen mit Integerzahlen. Speziell bei der Addition müssen zunächst die Mantissen und Exponenten verschoben werden, bevor die Mantissen addiert werden können. Das Ergebnis muss anschließend gegebenenfalls wieder in die Normalform gebracht werden. Die Angabe der möglichen Gleitkommaoperationen pro Sekunde (FLOPS: *floating point operations per second*) ist eine wichtige Kenngröße für die Leistungsfähigkeit eines Computers.

Übung 5.1.2. Gleitkommadarstellung
Betrachten Sie das folgende einfache Format für Gleitkommazahlen:

- Bit 15: Vorzeichen des Exponenten (0 für positiv)
- Bit 14: Vorzeichen der Mantisse (0 für positiv)
- Bit 6-13 Betrag der Mantisse
- Bit 0-5 Betrag des Exponenten

Welchen Wert haben die folgenden Bitmuster:

0	0	0110 0000	00 0011
1	0	1010 0000	00 0100

Übung 5.1.3. Wertebereich im Standard IEEE 754
Welches ist jeweils die

- kleinste
- größte
- betragsmäßig kleinste

darstellbare Zahl im short real Format?

Übung 5.1.4. Konvertierung in den Standard IEEE 754
Wie wird die Zahl $14,625_{10}$ als Gleitkommazahl im Format real short dargestellt? Geben Sie das Resultat in Binär- und Hexadezimaldarstellung an.

Übung 5.1.5. Addition und Multiplikation
Berechnen Sie Summe und Produkt für die beiden Zahlen $7,5 \cdot 10^4$ und $6,34 \cdot 10^2$. Geben Sie das Ergebnis jeweils in normierter Form mit einer Vorkommastelle an. Welche einzelnen Rechenschritte sind erforderlich?

Übung 5.1.6. Darstellung der Zahl 0
In IEEE 754 gibt es zwei Bitmuster für die Zahl 0, einmal mit positivem und einmal mit negativem Vorzeichen. Diese Werte kann man als +0 und -0 interpretieren. Wo kann man diese Unterscheidung sinnvoll einsetzen? Welche Probleme können sich aus der doppelten Darstellung ergeben?

5.1.1 Vergleich der Zahlenformate

Abschließend sind die wesentlichen Unterschiede in den Zahlendarstellungen in Tabelle 5.1 zusammengestellt. Abhängig von der Anwendung sind die einzelnen Kriterien unterschiedlich zu gewichten. Beispielsweise spielt bei einer low-cost-Anwendung der Preis eine entscheidende Rolle, so dass auf eine eigene Einheit für Gleitkommaoperationen verzichtet wird. Dann ist es oft notwendig Berechnungen, für die Gleitkommazahlen besser geeignet wären, aus Performanzgründen trotzdem mit Integerzahlen durchzuführen.

Tabelle 5.1: Vergleich zwischen Integer- und Gleitkommazahlen

	Integer	Gleitkomma
Nachkommastellen	Nein	Ja
Genauigkeit	Ergebnisse sind exakt	Rundungsfehler
Bereich	eingeschränkt	groß
Überlauf	wird nicht gemeldet	Inf
Rechenaufwand	niedrig	groß
Speicherbedarf	8-32 Bit	32-80 Bit

5.2 Gleitkommazahlen in Java

Java unterstützt die beiden Gleitkomma-Typen `float` und `double` gemäß IEEE 754. Zwischen Gleitkomma-Werten gelten die Grundrechenarten mit den Operatoren +, -, * und / sowie dem Modulo-Operator `%` mit den schon behandelten Vorrangsregeln. Auch die Vergleichsoperatoren sind für Gleitkomma-Werte gültig. Allerdings sind die Bit-Operatoren für Gleitkomma-Werte nicht definiert.

Die Genauigkeit beträgt etwa 7 Stellen bei `float` und 15 Stellen bei `double`. Bei der Eingabe von Konstanten benutzt man den Punkt anstelle des Kommas. Optional kann man einen Zehner-Exponenten angeben. Einige Beispiele sind

```
0.456
-177.999
99.988e17
-1e-10
.1e-10
```

Die Werte werden intern normalisiert, bei der Eingabe ist man frei in der Anzahl der Stellen vor dem Dezimalpunkt. Der Dezimalpunkt direkt vor dem Exponenten kann weggelassen werden. Standardmäßig interpretiert der Compiler diese Konstanten als `double`. Durch Anhängen der Endung `f` (`F`) werden sie zu `float`-Werten.

5.2.1 Rechnen mit Gleitkommazahlen

Der große Vorzug von Gleitkommazahlen ist der große Wertebereich mit gleichbleibender relativer Genauigkeit. Im Gegensatz zu Integerwerten stellt für übliche Anwendungen der Wertebereich kein Problem dar. Über- oder Unterschreitungen sind eher selten. Allerdings muss man stets mit Rundungseffekten rechnen. Problematisch sind Vergleiche in der Art

```
if( x == 5 )
```

Wenn `x` das Ergebnis einer Rechnung ist, kann es durchaus sein, dass `x` aufgrund von Rundungsfehlern den Wert `4.99999` hat. Die Abfrage würde dann nicht er-

füllt sein. Als Ausweg kann man in einem solchen Fall den Betrag des Abstandes vom Vergleichswert bestimmen und die Prüfung in der Form

```
if( Math.abs( x - 5 ) < epsilon )
```

schreiben. Dabei gibt `epsilon` die gewünschte oder geforderte Genauigkeit an. Mit der gleichen Vorsicht muss man for-Schleifen mit Gleitkommawerten betrachten. In dem Beispiel

```
for ( x=0.; x<=1.; x+=0.01 )
```

ist nicht gewährleistet, dass der Wert `1` exakt getroffen wird, so dass eventuell (z.B. bei `x=1.0000001`) die Schleife eine Iteration zu früh abgebrochen wird.

Grundsätzlich ist die Abdeckung innerhalb des Wertbereichs – im Gegensatz zu den Integerzahlen – nicht vollständig. Zwischen benachbarten Gleitkommazahlen ist eine Lücke, und die Breite der Lücke hängt von der Größe der Zahlen ab. Dies kann zu Effekten führen, die der Mathematik widersprechen. Das folgende Fragment Java-Code dient zur Veranschaulichung dieses Verhaltens. In diesem Programm wird der Wert 1 zu einer Zahl addiert, wobei die Zahl schrittweise um den Faktor 10 erhöht wird.

```
double zahl = 1.;
double zahlplus1;

do {
  zahl *= 10.;
  zahlplus1 = zahl + 1.;
  System.out.printf("%10g %10g %5g \n", zahl, zahlplus1, zahlplus1 - zahl);
} while (zahlplus1 > zahl);
```

Die Ausführung liefert die Ausgabe:

```
   10,0000 11,0000 1
   100,000 101,000 1
   1000,00 1001,00 1
   10000,0 10001,0 1
    100000 100001 1
1,00000e+06 1,00000e+06 1
1,00000e+07 1,00000e+07 1
1,00000e+08 1,00000e+08 1
1,00000e+09 1,00000e+09 1
1,00000e+10 1,00000e+10 1
1,00000e+11 1,00000e+11 1
1,00000e+12 1,00000e+12 1
1,00000e+13 1,00000e+13 1
1,00000e+14 1,00000e+14 1
1,00000e+15 1,00000e+15 1
```

```
1,00000e+16 1,00000e+16 0
```

Zunächst zeigt das Programm das erwartete Verhalten und berechnet die Differenz zu 1. Aber wenn der Wert 10^{16} erreicht ist, führt die Addition nicht mehr zu einer anderen Zahl und die Differenz zwischen 10^{16} und $10^{16}+1$ liefert nur noch den Wert 0.

Frage 5.1. Was passiert, wenn man einen größeren Wert addiert?

Die Schleife läuft dann weiter bis auch der größere Wert kleiner wird als der halbe Abstand zur nächsten Zahl. Addiert man beispielsweise 10000 endet die Ausgabe mit

```
1,00000e+16 1,00000e+16 10000,0
1,00000e+17 1,00000e+17 10000,0
1,00000e+18 1,00000e+18 9984,00
1,00000e+19 1,00000e+19 10240,0
1,00000e+20 1,00000e+20 16384,0
1,00000e+21 1,00000e+21 0,00000
```

Die Rundungsfehler werden dabei bereits ab 10^{18} sichtbar.

Übung 5.2.1. Abstand zwischen Gleitkommazahlen
Die Methode `Math.ulp( double d)` gibt den Abstand zwischen `d` und der nächsten Gleitkommazahl zurück. Die Abkürzung *ulp* steht dabei für *unit in the last place*. Erweitern Sie das Beispiel, so dass auch der Abstand vom jeweiligen Wert von `zahl` zur nächsten Gleitkommazahl ausgegeben wird.

Übung 5.2.2. Reihenfolge der Auswertung eines Ausdrucks
Sei $x = 10^{30}$, $y = -10^{30}$ und $z = 1$. Welches Resultat ergibt sich bei dem Rechnen in Gleitkomma-Arithmetik für die beiden Ausdrücke

- $(x + y) + z$
- $x + (y + z)$

5.2.2 Mathematische Funktionen

Eine ganze Reihe von mathematischen Funktionen sind in Java standardmäßig verfügbar. Die Funktionen sind als Methoden einer Klasse `Math` aufrufbar. Wir werden in einem späteren Kapitel auf die Verwendung von Methoden genauer eingehen. An dieser Stelle verwenden wir Methoden als Implementierung von mathematischen Funktionen. Die Syntax in Java für den Aufruf einer Methode ist

```
Klasse.methode( Parameterliste )
```

Die Methode erhält eine Anzahl von Parameter, berechnet aus diesen Werten ein Ergebnis und gibt dieses Ergebnis zurück. Das Beispiel

```
y = Math.sin( t );
```

berechnet den Sinus des Wertes in der Variablen `t` und speichert das Resultat in `y`. Unter anderem gibt es:

- trigonometrische Funktionen: `sin(x)`, `cos(x)`, `tan(x)`, Argument jeweils im Bogenmaß
- inverse trigonometrische Funktionen: `asin(x)`, `acos(x)`, `atan(x)`
- Potenzen und Logarithmen: `exp(x)`, `log(x)`, `sqrt(x)`, `pow(x,y)`
- Runden:
  - `ceil(x)` – kleinste ganze Zahl größer `x`
  - `floor(x)` – größte ganze Zahl kleiner `x`
- Betrag: `abs(x)`
- Zufallszahlen: `random()` liefert eine Zufallszahl aus dem Intervall $[0, 1]$
- Vergleich: `min(x,y)` und `max(x,y)` geben das Minimum beziehungsweise Maximum der beiden Parameter zurück

Außerdem enthält die Klasse die beiden Konstanten `E` und `PI`. Sonderfälle entstehen, wenn

- das Argument nicht im Definitionsbereich liegt (*domain error*)
- das Ergebnis nicht mehr darstellbar ist (*range error*)

Die Funktionen geben in diesen Fällen definierte Sonderwerte zurück, die auch bei der Ausgabe als solche dargestellt werden. Das Beispiel

```
for ( double x=0; x<1000; x+=300. ) {
   System.out.println("x = "+ x +", exp(x) = "+ Math.exp(x) );
}
System.out.println( "log( 0.) = " + Math.log( 0.));
System.out.println( "log( -1) = " + Math.log( -1));
System.out.println( "wurzel(-1.) = " + Math.sqrt(-1.));
```

ergibt

```
x = 0.0, exp(x) = 1.0
x = 300.0, exp(x) = 1.94242639524125 58E130
x = 600.0, exp(x) = 3.7730203009299397E260
x = 900.0, exp(x) = Infinity
log( 0.) = -Infinity
log( -1) = NaN
wurzel(-1.) = NaN
```

Die Sonderwerte sind in der Klasse `Double` definiert. So ergibt

```
System.out.println( Double.MIN_VALUE );
System.out.println( Double.MAX_VALUE );
System.out.println( Double.NaN );
System.out.println( Double.POSITIVE_INFINITY );
```

die Ausgabe

```
4.9E-324
1.7976931348623157E308
NaN
Infinity
```

Allerdings kann man `NaN` und `POSITIVE_INFINITY` nicht direkt mit `==` mit anderen Werten vergleichen. Aber die Klasse `Double` stellt entsprechende Methoden zur Verfügung. Die beiden Zeilen

```
System.out.println( Math.sqrt(-1) == Double.NaN );
System.out.println( Double.isNaN( Math.sqrt(-1) ) );
```

mit dem Ergebnis

```
false
true
```

zeigen beispielhaft dieses Verhalten.

Frage 5.2. Kann man in Java gar nicht mit größeren Werten rechnen?

Doch, die Java-Bibliothek enthält die Klasse `BigDecimal` für solche Fälle. Als Beispiel wird in

```
var b = new java.math.BigDecimal(Double.MAX_VALUE);
System.out.format("b: %.4E\n", b);
b = b.multiply( b );
System.out.format("b: %.4E\n", b);
```

mit der Ausgabe

```
b: 1,7977E+308
b: 3,2317E+616
```

das Quadrat des größten Double-Wertes berechnet.

5.3 Umwandlung zwischen Datentypen

In einem Ausdruck können die Operanden verschiedene Datentypen haben. Der Typ des Ergebnisses hängt dann von den beteiligten Datentypen ab. Auch bei einer Zuweisung hat der Ausdruck auf der rechten Seite manchmal einen anderen Ergebnistyp als die Variable auf der linken Seite. In solchen Fällen erfolgt eine Umwandlung des Typs vor der Rechnung bzw. der Zuweisung. Die Umwandlung erfolgt dabei stets vom „kleineren“ zum „größeren“ Typ (erweiternde Konvertierung). Beispielsweise wird bei der Addition eines `int`- und eines `long`-Wertes zunächst der `int`-Wert in einen `long`-Wert gewandelt. Die Umwandlung erfolgt allerdings „Schritt für Schritt“. Selbst wenn etwa auf der linken Seite einer Anweisung eine `double`-Variable steht, werden nicht notwendigerweise alle Rechnungen in `double` ausgeführt. Das folgende Beispiel demonstriert diesen Effekt:

```
double test;
test = 10 / 3;
System.out.println( "Konvertierung 1, test = " + test);
test = 10. / 3;
System.out.println( "Konvertierung 2, test = " + test);
```

Im ersten Fall wird die rechte Seite noch als Integer-Rechnung ausgeführt. Erst das Ergebnis wird umgewandelt. Im zweiten Fall ist der erste Operand `10.` bereits ein `double`, so dass vor der Division die `3` umgewandelt wird. Entsprechend liefert das Beispiel

```
Konvertierung 1, test = 3.0
Konvertierung 2, test = 3.3333333333333335
```

Bei der erweiternden Konvertierung bleibt die Information erhalten. Sie wird daher als sicher angesehen und wenn notwendig vom Compiler automatisch eingefügt. Demgegenüber sind Konvertierungen in die andere Richtung (einschränkende Konvertierung) unsicher. Eventuell verliert man Genauigkeit oder der Wert passt nicht mehr in den geringeren Darstellungsbereich. Daher werden einschränkende Konvertierungen vom Compiler als Fehler behandelt. Die Anweisung

```
int i = 1.5;
```

führt zu der Fehlermeldung

```
FloatTest.java [35:1] possible loss of precision
found : double
required: int
       int i = 1.5;
               ^
1 error
Errors compiling main.
```

Man kann explizit eine Konvertierung mittels des so genannten Type-Cast-Operators anfordern. Die allgemeine Form, um einen Ausdruck a in einen anderen Datentyp zu wandeln, ist

```
(Datentyp) a
```

Sofern möglich, wird der Ausdruck in den in Klammern angegebenen Typ konvertiert. Damit werden auch einschränkende Konvertierungen in der Art

```
int i = (int) 1.5;
```

vom Compiler akzeptiert. Allerdings gilt dies nur für legale Konvertierungen. Der Versuch, einen `float`-Wert in einen `boolean` zu wandeln,

```
boolean b = (boolean) 1.5;
```

scheitert bereits beim Kompilieren mit der Fehlermeldung

```
FloatTest.java [36:1] inconvertible types
found   : double
required: boolean
        boolean b = (boolean) 1.5;
                              ^
1 error
Errors compiling main.
```

5.4 Gleitkommazahlen und BoSym

In unserer Anwendung BoSym gibt es nur eine Gruppe von Funktionen, die eine Gleitkommazahl als Parameter erwartet. Mit den drei Funktionen

```
symbolGroesse(int i, double r)
symbolGroesse2(int i, int j, double r)
symbolGroessen( double r)
```

kann die Größe der Symbole verändert werden. Angegeben wird der Radius `r`, wobei der Wert 0,5 einem vollen Feld entspricht. Ein Beispiel für wachsende Symbole zeigt das folgende Code-Abschnitt

```
int anzahl = 20;

formen( "*" );
groesse( anzahl, 1);
for (int i=0; i<anzahl; i++ ) {
    farbe( i, BLACK );
    symbolGroesse( i, 0.1 + 0.4 * i / anzahl );
}
```

und das resultierende Bild 5.1.

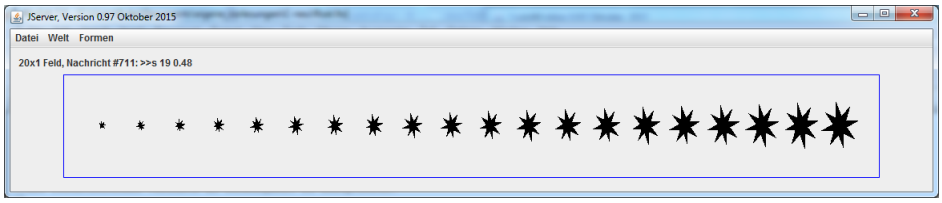


Abbildung 5.1: Symbole mit zunehmender Größe

5.5 Übungen

Übung 5.5.1. Schreiben Sie ein Programm, das die Lösungen der quadratischen Gleichung

$$x^2 + p \cdot x + q = 0$$

berechnet. Dabei soll q den festen Wert 0,1 haben und p in Schritten von 0,1 von 0 bis 2 laufen. Prüfen Sie jeweils, ob eine reelle Lösung existiert. Falls ja, berechnen Sie die beiden Lösungen. Zur Kontrolle setzten Sie die gefundenen Werte in die Gleichung ein und geben das Ergebnis ebenfalls aus. Im Fall von komplexen Lösungen geben Sie einen entsprechenden Hinweis aus. `Hinweis`: zum Berechnen der Wurzel gibt es die Methode `Math.sqrt()`;

Übung 5.5.2. Nach einer alten Legende wünschte sich der Erfinder des Schachspiels vom König:

- 1 Reiskorn auf dem ersten Feld eines Schachbrettes
- 2 Reiskörner auf dem 2. Feld
- 4 Reiskörner auf dem 3. Feld
- u.s.w.

Geben Sie die Anzahl der Reiskörner mit wachsender Anzahl von Feldern aus. Berechnen Sie zusätzlich, wie viele LKWs (je 7,5 Tonnen Ladung) man für den Transport benötigt, wenn jedes Reiskorn 30 mg wiegt. Wenn weiterhin ein Reiskorn ein Volumen von etwa $3.5 \cdot 10^{-8} m^3$ hat, wie hoch wird dann ein Fußballfeld (70 auf 105 m) bedeckt?

Übung 5.5.3. Sie nehmen ein Darlehen über 100.000 Euro auf. Der jährliche Zinssatz beträgt 4,5%. Jeden Monat zahlen Sie eine Rate von 500 Euro. Diese Rate beinhaltet zunächst die Zinsen für diesen Monat (4,5/12 auf Restguthaben). Mit dem verbleibenden Rest wird das Darlehen getilgt. Im ersten Monat beträgt der Zins $100.000 * 0,045/12 = 375$. Dementsprechend wird das Darlehen um 125 Euro auf 99875 Euro vermindert. Im zweiten Monat muss dann nur noch für diesen Betrag Zinsen bezahlt werden.

Verfolgen Sie per Programm die Entwicklung des Darlehens. Berechnen Sie dazu jeden Monat die verbliebene Restschuld und geben diesen Wert jeweils am Ende eines Jahres aus. Wie lange dauert es, das Darlehen zu tilgen? Wie viele Zinsen werden bis zum Ende insgesamt gezahlt worden sein?

Übung 5.5.4. Bestimmen Sie durch Monte Carlo-Simulation eine Näherung für die Zahl π. Betrachten Sie dazu ein Quadrat der Seitenlänge 1, in dem ein Viertelkreis mit Radius 1 liegt. Die Fläche $F_Q$des Quadrats ist 1, die des Viertelkreises $F_V = \pi/4$. Damit gilt die Beziehung $\pi = 4 \cdot F_V/F_Q$. Zur experimentellen Bestimmung des Verhältnises F_V/F_Q erzeugt man zufällige Punkte im Intervall $[0,1;0,1]$ und zählt, wie viele davon in den Viertelkreis fallen.

Übung 5.5.5. Funktion
Lassen Sie die Funktion $\sin(x)$ für Werte von x von 0 bis 2π darstellen.

- Rechnen Sie die Brettkoordinaten $0, 1, \ldots, N-1$ in entsprechende Werte aus dem Bereich $[0, 2\pi]$ um.
- Berechnen Sie dann für jedes Feld den entsprechenden Wert $sin(x)$.
- Negative Werte sollen rot, positive grün gefärbt werden.
- Die Funktion liefert Werte zwischen -1 und 1 zurück. Dementsprechend soll die Größe der Symbole gewählt werden. Beachten Sie, dass ein Radius von 0,5 das ganze Feld ausfüllt. Das Programm akzeptiert auch negative Werte.

Das Ergebnis könnte wie folgt aussehen:

Übung 5.5.6. Muster
Erweitern Sie die Lösung der Aufgabe 5.5.5 auf zwei Dimensionen und lassen die Funktion $sin(x) * cos(y)$ für Werte von x und y von 0 bis 2π darstellen.

Kapitel 6

Felder

In Kapitel 3 haben wir gelernt, dass wir über die Variablennamen Speicherplätze ansprechen können. Betrachten wir diesmal Häuser mit ihren Bewohnern als anschauliches Beispiel. Dann vergeben wir für jedes Haus einen Namen. Bild 6.1 zeigt ein entsprechendes Beispiel. Wir haben einen Platz oder eine Straße mit vier Häusern. Jedes Haus hat einen Namen und diese Namen dienen als Adresse.

Mit einigen wenigen Häusern funktioniert das gut. Wenn auf einem Paket angegeben ist, dass Sabine in Haus *Sonne* wohnt, kann der Kurier es gut zustellen. Schwierig wird es, wenn die Anzahl der Häuser wächst. Schon die Vergabe eindeutiger Namen ist nicht einfach und die Suche nach einem Haus kann langwierig werden. Die praktische Lösung ist, den Häusern keine Namen, sondern fortlaufende Nummern zu geben. Man benötigt dann nur noch einen Namen für die Straße. Die Kombination Straßenname und Hausnummer ist dann eine gut zu findende Adresse. Bild 6.2 veranschaulicht dieses Konzept.

In der Tat benötigt man in der Programmierung häufig nicht nur eine einzelne Variable eines bestimmten Datentyps, sondern eine Reihe oder Anzahl von gleichartigen Variablen. Beispiele sind:

- die Matrikelnummern aller Hörer dieser Vorlesung

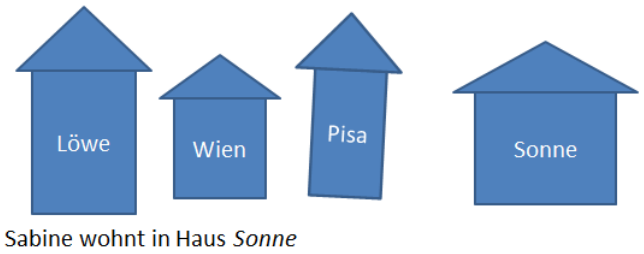


Abbildung 6.1: Häuser mit Namen

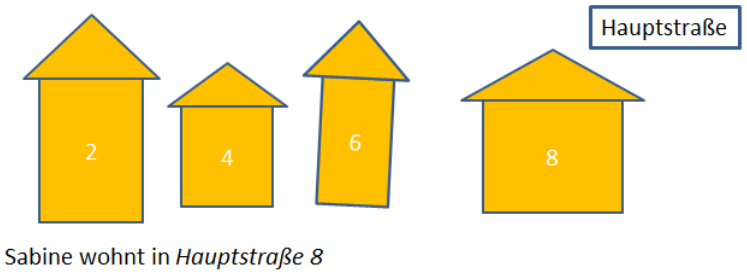


Abbildung 6.2: Häuser mit Namen

- alle Primzahlen kleiner 1000
- die Komponenten x, y, z eines Vektors $\overrightarrow{v}$

Charakteristisch ist, dass ein fester Datentyp mehrfach benötigt wird. Im Beispiel der Matrikelnummern braucht man eine Struktur, die nacheinander alle Nummern (jeweils in einer `int`-Variablen) enthält. Im Speicher hat man folgende Darstellung:

1. Matrikelnummer
2. Matrikelnummer
...
N-te Matrikelnummer

Die N aufeinander folgende Speicherzellen enthalten die Matrikelnummern der Studenten und Studentinnen. Die entsprechende Datenstruktur ist das Feld (engl. *array*). Da ein Feld aus den einfachen Datentypen aufgebaut ist, spricht man von einem strukturiertem Datentyp. Ein Feld ist charakterisiert durch:

- einen Namen (Bezeichner)
- einen Datentyp
- eine vorgegebene, feste Anzahl der Elemente, d. h. Variablen des Datentyps
- Indizes für die Elemente

Die Deklaration eines Feldes entspricht der Deklaration einer Variablen, ergänzt um ein Paar eckiger Klammern. Die Klammern können hinter dem Datentyp oder hinter dem Variablennamen stehen:

```
int ifeld[];
byte[] bfeld;      // meine bevorzugte Form
```

Nach der Deklaration muss noch Speicherplatz reserviert werden. Dies erfolgt mit dem Operator `new`. Um für das oben deklarierte Integerfeld eine Größe von 10 Werten anzulegen, schreibt man:

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

Deklaration und Speicherreservierung können auch gleichzeitig erfolgen:

```
float[] feld = new float[100];
```

Der neu reservierte Speicherbereich wird standardmäßig mit Nullen gefüllt. Bei Zahlentypen ist dies der Wert `0`, während Felder von `boolean` mit `false` belegt werden. Nach der Definition `double[] aktienWerte = new double[30]` als Beispiel gilt:

- Es gibt ein Feld mit dem Namen `aktienWerte`.
- Das Feld besteht aus 30 Speicherzellen.
- Jede Speicherzelle hat Platz für einen Wert vom Typ `double`.
- Alle Speicherzellen enthalten als Anfangswert 0.

Eine zweite Form der Initialisierung erlaubt es, die Werte (Literale) direkt anzugeben. Die Größe des Feldes wird dann automatisch bestimmt.

Beispiel 6.1. Anlegen eines Feldes mit 4 Werten

```
int[] feld = { 1, 3, 5, 7};
```

Diese Form kann nur zusammen mit der Deklaration verwendet werden. Die Felder in Java sind semidynamisch. Ihre Größe wird zur Laufzeit festgelegt, ist danach aber fest. Es ist nicht möglich, ein bestehendes Feld zu erweitern oder zu verkleinern. Allerdings kann jederzeit ein neuer Speicherbereich mit unterschiedlicher Größe angefordert werden.

```
ifeld = new int[10];
...
ifeld = new int[20];
```

Eine zweite oder weitere Reservierung legt einen vollständig neuen Speicherbereich an. Die alten Werte werden nicht in den neuen Speicherbereich übernommen.

6.1 Zugriff auf Elemente

In Java bieten die Felder einen Namen für eine Reihe gleicher Elemente. Man kann nicht direkt mit Feldern rechnen. Der Versuch in der Art

```
int[] a = new int[3], b = new int[3];
a += b;
```

führt zu der Fehlermeldung

```
Feldtest.java [22:1] operator + cannot be applied to int[],int[]
    a += b;
          ^
1 error
```

Zum Rechnen muss man auf die einzelnen Elemente zugreifen. Zugriff erfolgt über den Index, d. h. die Position im Feld. Die Zählung der Elemente beginnt in Java immer mit dem Index 0. Bei einem Feld mit 10 Elementen haben die einzelnen Elemente die Indizes $0, 1, 2, \ldots, 9$. Ansprechen kann man ein Element über den Namen des Feldes und den Index in eckigen Klammern:

```
a[2]
```

ist das 3. Element (das Element mit Index 2) des Feldes `a`. Mit einem solchen Element kann man umgehen wie mit einer Variablen. Es kann sowohl in Ausdrücken eingesetzt werden als auch Ziel einer Zuweisung sein.

```
int[] a = new int[3], b = new int[10];
a[0] = 50 * b[2] + 30 * b[3];
```

Um alle Elemente eines Feldes zu bearbeiten, kann man beispielsweise `for`-Schleifen benutzen. So wird in

```
int size = 10;
int[] feld = new int[size];
int i;

for (i = 0; i < size; i++) {
   feld[i] = i * i;
}

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

ein Feld mit den ersten 10 Quadratzahlen gefüllt:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Frage 6.1. Was bedeutet `Arrays.toString`?

`Arrays` ist ein Bibliotheksklasse mit diversen Hilfsmethoden für Felder. Die Methode `toString` gibt eine Textbeschreibung mit den Werten im Feld zurück. Der direkte Aufruf

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

gibt nur eine für uns wenig hilfreiche Kennung des Feldes zurück (mehr dazu später in Abschnitt 9.2). Der Zugriff auf ein Element außerhalb des definierten Bereichs führt zu einem Laufzeitfehler. Der Versuch

```
int size = 10;
int[] feld = new int[size];

feld[11] = 111;
```

führt zur Fehlermeldung

```
java.lang.ArrayIndexOutOfBoundsException
        at Feldtest.main(Feldtest.java:31)
Exception in thread "main"
```

Damit werden eine Vielzahl von Programmierfehlern abgefangen. In anderen Sprachen wie z. B. C ohne Bereichsprüfung sind Fehler durch falsch berechnete Zugriffe oft nur sehr schwer zu finden. Der Preis dafür ist die langsamere Ausführung durch die internen Prüfabfragen.

Intern wird ein Feld als Objekt behandelt. In einem Feld-Objekt ist auch die Information über die Länge abgelegt. Diese Information kann über eine Variable namens `length` abgefragt werden. Die genaue Syntax ist *feldname*`.length`. Dann kann ein Feld in der Form

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

ausgegeben werden. Es ist auf diese Art und Weise jederzeit möglich, die aktuelle Länge eines Feldes abzufragen. Seit Version 1.5 bietet Java für solche Fälle eine erweiterte Form der for-Schleife an. Anstelle eines expliziten Zählers werden nacheinander alle Elemente eines Feldes angesprochen (Mengenschleife). Diese Art von Schleifen wird allgemein als foreach-Schleife bezeichnet. Die Ausgabe aller Elemente wird dann in der Form

```
for (int w : feld) {
  System.out.println( w );
}
```

realisiert. In der Schleife werden nacheinander alle Elemente des Feldes `feld` der Variablen `w` zugewiesen. Dies entspricht der ausführlichen Schreibweise

```
for (int j = 0; j < feld.length; j++) {
  int w = feld[j];
  ...
}
```

Die Variable `w` ist eine Kopie des entsprechenden Feld-Inhaltes. Änderungen an `w` haben daher keine Auswirkungen auf das Feld. Das Beispiel

```
int[] feld = {1, 3, 5, 7, 9};
System.out.println( Arrays.toString( feld ) );
for (int w : feld) {
    ++w;
    System.out.print( w + " ");
}
System.out.println( );
System.out.println( Arrays.toString( feld ) );
```

ergibt daher die Ausgabe

```
[1, 3, 5, 7, 9]
2 4 6 8 10
[1, 3, 5, 7, 9]
```

mit einem unveränderten Array nach der Schleife.

Frage 6.2. Hat ein Array auch selbst Methoden z. B. zum elementweisen Ausgeben?

Dazu kann man in Java Streams benutzen. Aus dem Array wird ein Stream erzeugt und dann dessen Methode `forEach` aufgerufen. Im Argument der Methode wird ein Lambda-Ausdruck mit ausführbarem Code übergeben. So kann man mit

```
Arrays.stream(feld).forEach(w -> System.out.println(w));
```

die Elemente ausgeben lassen. Zum übersichtlichen Vergleich zeigt der folgende Code die drei besprochenen Möglichkeiten, beispielhaft die Quadrate der Elemente auszugeben.

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

for (int w : feld) {
  System.out.println(w * w);
}

Arrays.stream(feld).forEach(w -> System.out.println(w * w));
```

6.2 Mehrdimensionale Felder

Felder sind nicht auf eine Dimension beschränkt. Mehrdimensionale Felder werden durch mehrere Klammerpaare spezifiziert:

```
// Zweidimensionales Feld mit 10 * 20 Elementen
int[][] zifeld = new int[10][20];
```

Zweidimensionale Felder sind intern ineinander geschachtelte Felder. Damit kann man auch nicht-rechteckige Felder erzeugen. Die Anweisung

```
zifeld[0] = new int[30];
```

führt dazu, dass das erste Unterfeld jetzt 30 Elemente hat. Im folgenden Code wird ein Feld für alle Tage eines Jahres geordnet nach Monaten angelegt. Für jeden Tag wird über den logischen Wert dargestellt, ob es sich um einen Urlaubstag handelt.

```
boolean[][] urlaub = new boolean[12][];
urlaub[0] = new boolean[31];
urlaub[1] = new boolean[28];
...
urlaub[7][0] = true; // 1. August ist ein Urlaubstag
urlaub[11][24] = true; // 25. Dezember ist ein Urlaubstag
```

6.3 Felder und BoSym

Felder sind sehr gut geeignet, um Anwendung für BoSym einfacher und übersichtlicher zu schreiben. Wir können zum Beispiel eine Farbenfolge in einem Feld speichern und dann direkt die Felder setzen. Der Code-Abschnitt

```
int[] farben = { RED, GREEN, BLUE, RED, RED, GREEN };

for ( int i=0; i<farben.length; i++ ) {
    farbe( i, farben[i] );
}
```

generiert das Muster in Bild 6.3. Der Feldindex entspricht dabei der Nummerierung der Brettfelder. In diesem Beispiel geben wir die Farben an und die Nummerierung läuft mit dem Feldindex. Wir können dies auch umkehren: für eine gegebene Farbe speichern wir die Nummern in ein Feld. In

```
int[] blau = { 3, 5, 7, 11, 15, 16, 2 };

for ( int i=0; i<blau.length; i++ ) {
    farbe( blau[i], BLUE );
}
```

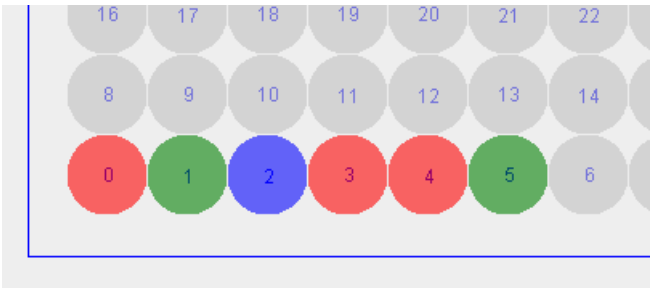


Abbildung 6.3: Farbenfolge in Feld

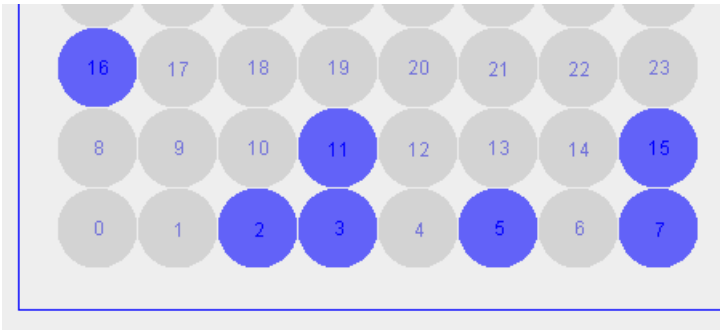


Abbildung 6.4: Felder in Blau

geben wir vor, welche Brettfelder blau gefärbt werden sollen (Bild 6.4). Die größte Flexibilität erreichen wir, indem Feldnummer und Farbe jeweils paarweise abgespeichert werden. Wertepaare lassen sich äquivalent in zweidimensionalen Feldern abspeichern. Die entsprechende Variante ist

```
int[][] felder = { {3, BLUE}, {5, RED}, {1, GREEN} };

for ( int i=0; i<felder.length; i++ ) {
    farbe( felder[i][0], felder[i][1] );
}
```

mit drei Wertepaaren Feldnummer – Farbe (Bild 6.5).

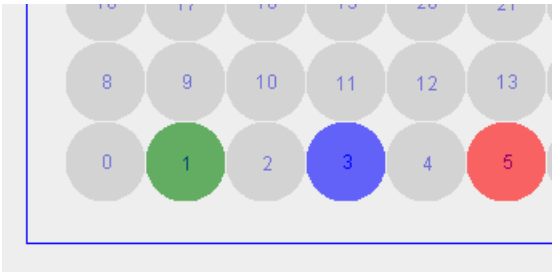


Abbildung 6.5: Felder und Farben

6.3.1 Beispiel Brettspiel

Felder sind besonders gut geeignet, um die Informationen für Spiele zu speichern. Dabei gibt es verschiedene Möglichkeiten:

- Für jeden Spielergibt es ein eigenes Feld mit den Positionen der Spielsteine.
- Der Spieler-Index wird als zusätzliche Dimension in ein einziges Feld aufgenommen.
- Ein Feld repräsentiert das Spielbrett und die Spielsteine werden dort eingetragen.

Welche Struktur am besten passt, lässt sich schwer allgemein beantworten. Beschränken wir uns auf Spiele mit nur zwei Spielern und quadratische Bretter wie zum Beispiel Dame, Vier gewinnt oder Go, so ist ein zweidimensionales Feld als Repräsentation des Spielbretts eine gute Lösung. Die Spielsteine werden dann – je nach Datentyp des Feldes – als Zahlenwerte oder als Zeichenketten in die einzelnen Zellen eingetragen. In Listing 6.1 ist der Aufbau eines 8×8-Spielfeldes mit der Anfangsstellung des Dame-Spiels gezeigt. Dabei stehen die Zahlen 1 und 2 für die Steine der beiden Spieler (noch ohne Dame). In zwei verschachtelten for-Schleifen wird das Spielfeld mit schwarz-weißen Hintergründen und den Spielsteinen dargestellt (Bild 6.6).

Frage 6.3. Warum sind die Positionen bei der Definition des Feldes `brett` gedreht?

Intern sind die N-dimensionalen Felder aus N-1-dimensionalen Feldern aufgebaut. Daher läuft der innerste (letzte) Index am schnellsten. Wir wollen aber ein xy-System benutzen. Daher müssen wir beim Anlegen die Dimensionen in umgekehrter Reihenfolge eingeben.

6.4 Übungen

Übung 6.4.1. Rechnen mit Feldern
Im folgenden Programm-Schnipsel wird ein Feld mit Zufallszahlen zwischen 0,1 und 0,5 gefüllt. Dazu passend werden Symbole auf die jeweilige Größe gesetzt.

```
int anzahl = 20;
double[] feld = new double[anzahl];

groesse( anzahl, 1);
loeschen();

for (int i = 0; i < anzahl; i++) {
```

Listing 6.1: Dame-Spiel

```
int[][] brett = {
       {1, 0, 1, 0, 0, 0, 2, 0 },
       {0, 1, 0, 0, 0, 2, 0, 2 },
       {1, 0, 1, 0, 0, 0, 2, 0 },
       {0, 1, 0, 0, 0, 2, 0, 2 },
       {1, 0, 1, 0, 0, 0, 2, 0 },
       {0, 1, 0, 0, 0, 2, 0, 2 },
       {1, 0, 1, 0, 0, 0, 2, 0 },
       {0, 1, 0, 0, 0, 2, 0, 2 }
    };

for ( int x=0; x<brett.length; x++ ) {
   for ( int y=0; y<brett.length; y++ ) {
      if( brett[x][y] == 1 ) {
         farbe2( x, y, BLUE );
         symbolGroesse2( x, y, 0.4 );
      } else if( brett[x][y] == 2 ) {
         symbolGroesse2( x, y, 0.4 );
         farbe2( x, y, YELLOW );
      } else {
         form2( x, y, "none" );
      }
      if( (x + y ) % 2 == 0 ) {
         hintergrund2( x, y, LIGHTGRAY );
      } else {
         hintergrund2( x, y , WHITE );
      }
   }
}
```

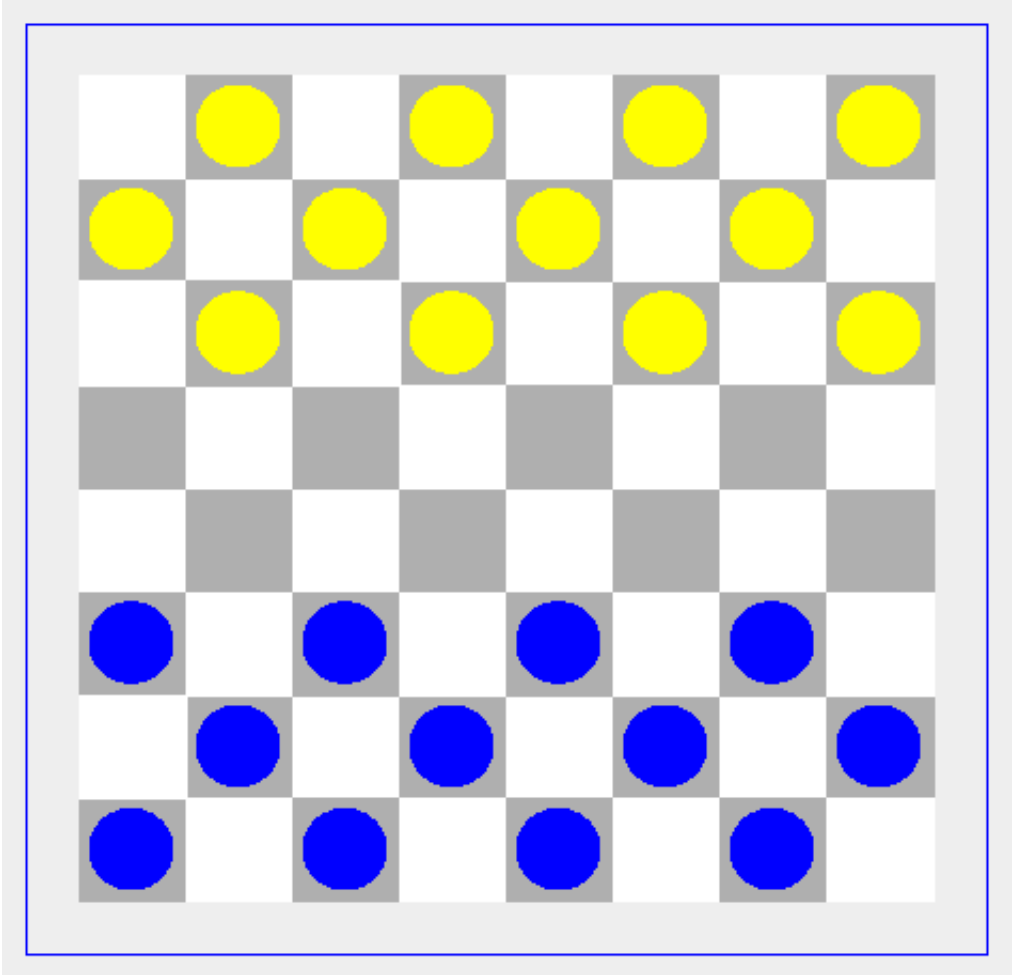
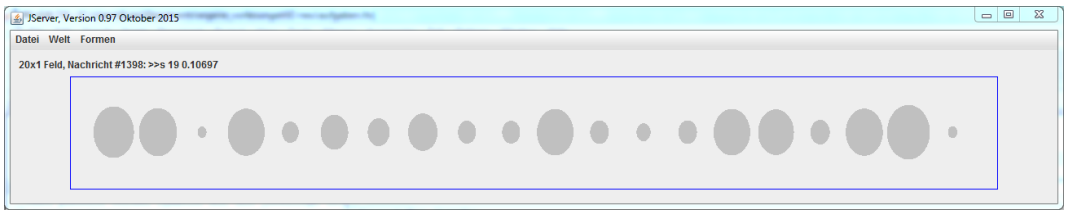


Abbildung 6.6: Ausgangsstellung für Dame-Spiel

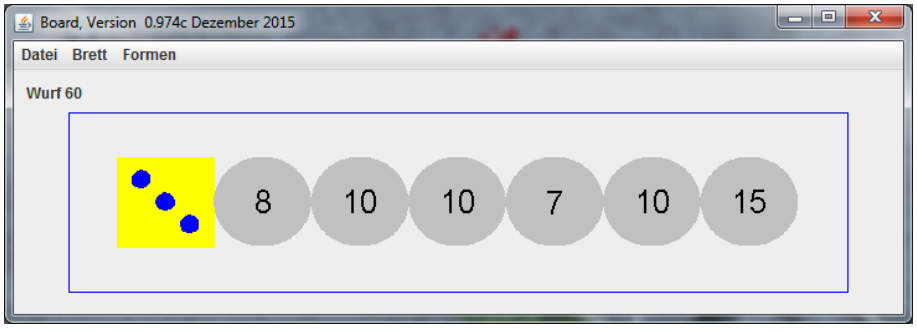
```
    feld[i] = 0.1 + 0.4 * Math.random();
    symbolGroesse(i, feld[i]);

}
```

Das Ergebnis sollte wie folgt aussehen:



- Bestimmen Sie Minimum und Maximum des Feldes und markieren es farblich (z. B. Minimum Rot, Maximum Grün).
- Geben Sie zusätzlich Minimum, Maximum und Mittelwert des Feldes mit `print` aus.

Übung 6.4.2. Felder als Zähler,
Implementieren Sie auf der Basis der Methode `Math.random()` eine Würfel-Simulation, die Zahlen zwischen 1 und 6 erzeugt. Zählen Sie in einem Feld entsprechender Größe, wie oft jede Zahl vorkommt und geben das Ergebnis mit `print` aus. Lassen Sie die gezählten Häufigkeiten mit BoSym darstellen. Dazu können Sie in der Art `text(i, "Text")` Texte in Felder schreiben. Eine Lösung:



Kapitel 7

Methoden

Ein wesentlicher Gedanke der strukturierten Software-Entwicklung ist die Aufteilung einer komplexen Aufgabe in einzelne Teilaufgaben. Programmtechnisch entspricht dies der Aufteilung des Programms in mehrere Untereinheiten oder Module. In Java ist die entsprechende Untereinheit eine Methode. Eine Methode ist ein eigenständiger Verarbeitungsblock mit definierten Eingangs- und Ausgangswerten. Vorteile von Methoden sind:

- Berechnungen, die mehrfach benötigt werden, brauchen nur einmal programmiert zu werden
- Kapselung von Funktionalitäten (*black box* Prinzip, *information hiding*)
- verbesserte Testmöglichkeiten
- Wiederverwendbarkeit in anderen Projekten

In verschiedenen Programmiersprachen gibt es unterschiedliche Bezeichnungen und Konzepte für Module. Man kann grob unterscheiden:

- Funktionen verfügen über Argumente und Rückgabewerte
- Prozeduren oder Unterprogramme (*subroutines*) haben keinen Rückgabewert
- Methoden sind in objektorientierten Sprachen Funktionen, die zu einer Klasse gehören
- Makros sind Quellcode-Bausteine, bei denen vor dem Kompilieren ein gemeinsames, in der Regel längeres Stück Code an die entsprechend markierten Stellen eingefügt werden (Makro-Ersetzung).

In diesem Kapitel betrachten wir Methoden als Bausteine mit spezifischen Funktionalitäten. Später werden wir auf ihre Rolle in der objektorientierten Programmierung zurück kommen. Wir verwenden fürs Erste nur so genannte Klassen-Methoden. Diese Methoden haben das zusätzliche Attribut `static`.

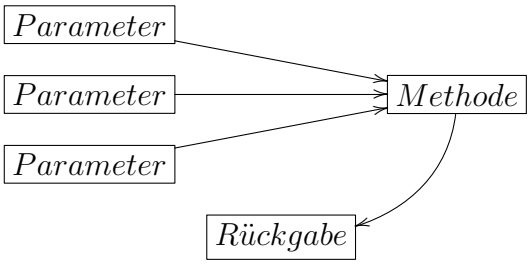


Abbildung 7.1: Beispiel Methode mit drei Parametern

7.1 Definition

Eine Methode ist ein Block von Definitionen und Anweisungen mit einem Namen sowie Eingangswerten (Parametern) und einem Ausgangswert. Bild 7.1 zeigt dieses Grundprinzip beispielhaft. Methoden stehen auf der ersten Ebene, d. h. die Definitionen können nicht ineinander geschachtelt werden. Allerdings ist es sehr wohl möglich, dass eine Methode eine weiter Methode aufruft. Der allgemeine Aufbau einer Methode in Java ist

```
rückgabe-typ   methodenName( parameterlist ) {
    ...
}
```

Zunächst wird der Datentyp des Rückgabewertes spezifiziert. Anschließend folgt der Name der Methode und dann in runden Klammern die Liste der Eingangswerte (Parameter). Jeder einzelne Parameter besteht aus dem Typ und einem Namen. Mehrere Parameter werden durch Komma getrennt. Die verkürzte Schreibweise wie etwa `(int i, j)` ist nicht erlaubt, jeder Parameter benötigt eine eigene Typangabe: `(int i, int j)`.

Die eigentliche Methode folgt dann als Block begrenzt durch geschweifte Klammern. Für die Rückgabe steht der Befehl `return ausdruck;` zur Verfügung. Bei Erreichen eines `return` Befehls wird die Bearbeitung der Methode beendet. Der Ausdruck wird ausgewertet und an das aufrufende Programm zurück gegeben. Innerhalb einer Methode können mehrere `return` Anweisungen stehen, um alternative Rücksprünge zu realisieren. Betrachten wir als erstes Beispiel eine Methode zur Berechnung der Kreisfläche:

```
double kreisFlaeche( double r ) {
   return Math.PI * r * r;
}
```

Sie erhält den Radius und gibt die daraus berechnete Fläche zurück. Sowohl der Parameter als auch der Rückgabewert ist dabei vom Typ `double`.

7.1.1 Methoden in BoSym

Fügt man diese Methode direkt als Code-Schnipsel ein, so scheitert die Übersetzung mit zwei Fehlern. Dies resultiert aus dem internen Ablauf. Ein Code-Schnipsel wird automatisch in eine Methode gepackt. Innerhalb dieser Methode kann keine weitere Methode definiert werden. Diesen Mechanismus kann man allerdings durch eine Zeile

```
@Complete
```

im Quellcode ausschalten. Dann wird der Code direkt übernommen. Allerdings kommt jetzt eine andere Fehlermeldung (gekürzt):

```
ATest.java:145: error: cannot find symbol

  symbol:   method mySend()
```

Benötigt wird eine Methode `mySend()` als Start der Ausführung. Mit der entsprechenden Ergänzung

```
@Complete

void mySend() {
   double x = 0.5;
   System.out.println(x + ": " + kreisFlaeche(x));
}

double kreisFlaeche(double r) {
   return Math.PI * r * r;
}
```

erhalten wir schließlich korrekten Code und die Ausführung ergibt

```
0.5: 0.7853981633974483
```

7.1.2 Beispiel Fakultät

Ein weiteres Beispiel zeigt die Berechnung der Fakultät $n! = 1 \cdot 2 \cdot \ldots \cdot n$:

```
int fakultaet(int wert) {
  int result = 1;
  // Argument ausserhalb des Wertebereiches ?
  if (wert < 0) {
    return -1;
  }
  while (wert > 1) {
    result *= wert;
```

```
    --wert;
  }
  return result;
}
```

Die Methode berechnet für den Eingangswert `wert` die Fakultät und gibt das Ergebnis zurück. Um eine Methode aufzurufen, wird sie über den Namen und mit entsprechenden Argumenten angesprochen. Eine Methode liefert in den meisten Fällen einen Wert zurück. Das aufrufende Programm (Hauptprogramm) kann diesen Wert ignorieren oder in einem Ausdruck als rvalue weiter verwenden.

```
// gibt den Wert 5! aus
System.out.println(fakultaet(5));
// legal aber wenig wirksam
fakultaet(2 * 3);
```

Bei dem Aufruf wird der Ausdruck in der Klammer ausgewertet und das Resultat wird an die Methode übergeben. Auch wenn man direkt eine Variable als Argument angibt, wird nur der Wert bzw. Inhalt der Variablen übergeben (*call by value*). Daher erfolgen alle Änderungen in der Methode an einem Argument nur an der lokalen Kopie. Im obigen Beispiel bleibt etwa der Wert der Variablen `z` unverändert.

Grundsätzlich hat einen Methode keinen Zugriff auf die Variablen im Hauptprogramm. Aufgrund der strikten Trennung gibt es auch keine Konflikte bei gleichen Namen für Variablen. Variablen innerhalb einer Methode haben nur die Lebensdauer eines Methodenaufrufs. Sie werden beim Aufruf angelegt und nach Ende der Methode wieder gelöscht (*automatic variable*). Dieser Vorgang wird für jeden neuen Aufruf wiederholt. Unsere Methode `mySend()` hatte weder Parameter noch Rückgabe.. In diesem Fall gilt

- Die Parameterliste kann leer sein
- Eine Methode ohne Rückgabewert hat den Typ `void` (engl. leer, nichtig).

Eine `void`-Methode endet entweder mit einem `return` ohne Wert oder an der schließenden Blockklammer.

7.2 Überladen von Methoden

Es ist in Java möglich, unter einem Namen mehrere Methoden mit unterschiedlichen Parameterlisten zu definieren. Die Methoden werden damit *überladen*. Selbstverständlich sollten die Methoden einen engen Bezug zueinander haben. Gebräuchlich ist dieser Mechanismus unter anderem, um zusätzliche Parameter zu übergeben. Wir können beispielsweise zwei Methoden zum Ausdrucken definieren:

```
void ausgeben() {
   System.out.println("Hallo");
}

void ausgeben(String titel) {
    System.out.println(titel);
}
```

Je nach Aufruf – mit oder ohne String-Argument – wählt der Compiler die richtige Version aus. Die verschiedenen Definitionen müssen nur eindeutig sein, d. h. die Parameter müssen sich im Typ oder in der Anzahl unterscheiden. Der Name einer Methode und die Information über ihre Parameter bilden zusammen die Signatur. Sind mehrere Methoden mit dem gleichen Namen vorhanden, wird beim Aufruf die am besten passende gewählt. Sofern erforderlich werden dabei Typ-Umwandlungen durchgeführt. Im folgenden Beispiel

```
void mySend() {
  f1(2);
  f1(2.);
  f2(2);
  f2(2.);
}

void f1(int i) {
  System.out.println("f1: int " + i);
}

void f1(double x) {
  System.out.println("f1: double " + x);
}

void f2(double x) {
  System.out.println("f2: double " + x);
}
```

werden die beiden Methoden `f1` und `f2` jeweils mit einem `int`- und einem `double`-Argument aufgerufen. Die Ausgabe

```
f1: int 2
f1: double 2.0
f2: double 2.0
f2: double 2.0
```

zeigt, dass bei Methode `f1` jeweils die direkt zum Argument passende Variante ausgeführt wird. Bei `f2` ist nur die Variante mit `double`-Argument vorhanden.

Die Umwandlung von `int` nach `double` ist erlaubt und wird automatisch ausgeführt. Versucht man den Aufruf mit einem Argument, für das es weder eine passende Variante gibt noch eine legale Typumwandlung möglich ist, so meldet der Compiler dies als Fehler. In unserem Fall führt

```
f1( true );
```

zu der Fehlermeldung

```
error: no suitable method found for f1(boolean)
```

Man kann dieses Konzept einsetzen, um die gleiche Funktionalität für unterschiedliche Datentypen bereit zu stellen. Beispielsweise kann man eine Methode `power` zur Berechnung von Potenzen sowohl für ganzzahlige als auch Gleitkomma-Werte implementieren:

```
double power( double x ) { ... }
int power( int i ) { ... }
```

Abhängig vom Argument wählt der Compiler die passende Methode aus. Soweit möglich werden dabei Typumwandlungen durchgeführt. Exitistiert beispielsweise ein Methode nur mit einem Parameter vom Typ `double`, so wird beim Aufruf mit einem Wert vom Typ `int` das Argument entsprechend umgewandelt. Eine andere häufig angewandte Technik simuliert Standardbelegungen (Default-Werte) für Parameter. Betrachten wir als Beispiel eine Methode

```
double wurzel( double x, int n ) {
    ...
}
```

zur Berechnung der n-ten Wurzel. Für den Standardfall Quadratwurzel ($n = 2$) soll zur Vereinfachung eine eigene Methode bereit stehen. Dies wird durch eine Methode mit nur einem Parameter implementiert:

```
double wurzel( double x ) {
   return wurzel( x, 2);
}
```

Diese Variante ruft wiederum die vollständige Variante auf und füllt den zweiten Parameter mit dem Standardwert. Die beiden Methoden können dann in der Art

```
System.out.println(wurzel(4.));
System.out.println(wurzel(4., 2));
System.out.println(wurzel(4., 3));
```

verwendet werden.

7.3 Beispiel BoSym

Die vorgegebenen Methoden in BoSym gelten jeweils nur für ein einzelnes Feld. Sollen z. B. mehrere nebeneinander liegende Felder eingefärbt werden, muss die Methode `farbe()` entsprechend oft aufgerufen werden. Praktisch wäre es, diese Aufgabe in eine Methode auszulagern. Diese Methode benötigt folgende Informationen zum Einfärben:

- Startfeld
- Länge (Anzahl der Felder)
- Farbe

Ein Rückgabewert wird nicht benötigt. Eine entsprechende Methode ist

```
/**
 * Methode zum F&auml;rben einer waagrechten Linie
 *
 * @param x Spalte des Startfeldes
 * @param y Zeile des Startfeldes
 * @param n Anzahl der Felder
 * @param farbe zu setzende Farbe
 */
void waagrecht(int x, int y, int n, int farbe) {
  for (int i = 0; i < n; i++) {
    farbe2(x + i, y, farbe);
  }
}
```

Vor der eigentlichen Methode steht ein Kommentar im Format von Javadoc. Javadoc ist ein Werkzeug, das aus dem Quellcode automatisch eine Dokumentation als HTML-Dokument generiert. Javadoc-Kommentaren werden mit der Zeichenfolge `/**` eingeleitet. Innerhalb dieser Kommentare sucht Javadoc nach so genannten Tags, die jeweils mit `@`-Zeichen beginnen.

Frage 7.1. Was bedeutet `&auml;`?

Um bei der Darstellung in der späteren HTML-Seite Problem mit Sonderzeichen wie Umlauten zu vermeiden, kann man in HTML entsprechende Referenzen anstelle der Zeichen verwenden. Hier steht `auml` für *a umlaut*. Unsere neue Methode kann jetzt in der Art

```
void mySend() {
   waagrecht(2, 3, 5, RED);
}
```

verwendet werden. Ähnlich können wir eine weitere Methode zum Setzen der Symboltypen schreiben. Wir müssen dazu nur anstelle der Farbe den Typ übergeben und in der Methode von `farbe2()` auf `form2()` umstellen:

```
void waagrecht(int x, int y, int n, String typ) {
    for (int i = 0; i < n; i++) {
       form2(x + i, y, typ);
    }
}
```

Die Methode darf den gleichen Namen haben. Anhand des letzten Parameters kann eindeutig entschieden werden, welche Variante zu nehmen ist. Mit

```
waagrecht(1, 2, 4, 0xff);
waagrecht(1, 2, 4, "*");
```

wird jetzt eine Linie mit vier blauen Sternen gezeichnet. Wahrscheinlich wird man häufiger Farben und Formen gleichzeitig ändern. Daher lohnt sich die Definition einer weiteren Methode

```
void waagrecht(int x, int y, int n, int farbe, String typ) {
    waagrecht(x, y, n, farbe);
    waagrecht(x, y, n, typ);
}
```

die beides verbindet. Die neue Methode greift auf die beiden bereits vorhandenen zurück. In diesem Beispiel hätte man auch gut wieder eine Schleife schreiben können. Aber es ist übersichtlicher, die Ausführung nur einmal zu schreiben und dann wieder zu verwenden. Falls irgendwann Änderungen notwendig werden sollten, müssen sie nur an einer Stelle durchgeführt werden.

Frage 7.2. Warum haben dann die beiden Methoden `farbe` und `farbe2` unterschiedliche Namen? Ein einheitlicher Namen und unterschiedliche Signaturen wären doch übersichtlicher.

Das wäre in der Tat eine gute Lösung. Allerdings sind die Namen auch für andere Programmiersprachen wie C vorgesehen, bei den Methoden beziehungsweise Funktionen nicht überladen werden können.

7.4 Übergabe von Feldern

Der Übergabemechanismus *call by value*, bei dem die Methode mit lokalen Kopien arbeitet, beschränkt sich auf die einfachen Datentypen. Bei komplexeren Objekten wie Feldern wäre dies nicht sinnvoll. Felder stets vollständig zu kopieren würde einen hohen Aufwand bedeuten. Außerdem ist es oft gerade gewünscht, per Methodenaufruf den Inhalt eines Feldes zu verändern. So ist es bei einer Methode

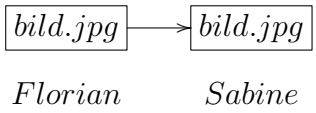


Abbildung 7.2: Kopieren einer Bilddatei

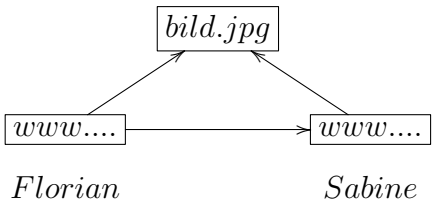


Abbildung 7.3: Übergabe eines Verweises

zum Sortieren effizienter, das Feld direkt zu bearbeitet. Daher wird bei Feldern statt des Inhalts ein Verweis (Referenz) an die Methode übergeben. Dementsprechend nennt man dieses Verfahren *call by reference*. Über diese Referenz können die einzelnen Zellen angesprochen werden.

Damit gibt es neben der uns schon bekannten Übergabe des Wertes (*call by value*) eine zweite Möglichkeit, Informationen an eine Funktion zu übermitteln. Man kann sich die prinzipiellen Unterschiede gut an einem praktischen Beispiel klar machen. Angenommen Florian hat eine Bilddatei `bild.jpg` und möchte das Bild an Sabine weitergeben. Eine Möglichkeit ist, ihr die Datei zu kopieren. Wie in Bild 7.2 dargestellt, erhält Sabine eine Kopie der Datei. Dies entspricht der Übergabe mittels call by value. Das Verfahren ist aufwändig bezüglich Übertragung und Speicherbedarf. Das komplette Bild wird übertragen und anschließend wird doppelter Speicherplatz belegt. Die beiden Versionen existieren von nun an getrennt. Ändert zum Beispiel Sabine etwas an ihrem Bild, so hat dies keinerlei Auswirkungen auf Florians Bild.

Die Übergabe als call by reference kann man sich demgegenüber wie folgt vorstellen: Florians Bild liegt irgendwo im Internet. Anstelle der Datei schickt er Sabine nur den Verweis (Link) auf diese Stelle. Bild 7.3 zeigt den Ablauf. In diesem Fall wird nur ein wenig Text für den Verweis übertragen. Florian und Sabine verwenden anschließend das gleiche Bild. Ändert Sabine etwas an dem Bild, so sieht Florian ebenfalls die Auswirkung.

Beide Übergabeverfahren haben Vor- und Nachteile. Je nach Anwendung kann das eine oder das andere besser geeignet sein. Wichtig ist, dass man sich die Unterschiede klar macht und dann die Konsequenzen in der weiteren Verarbeitung berücksichtigt. Formal verwendet man die gleiche Schreibweise. Der Ablauf ist im folgenden Beispiel dargestellt.

```
void mySend() {
   int[] test = {1,2,3,4,5,6,7,8,9,10};
```

```
   tausche(test, 0, 1);
   System.out.println(Arrays.toString(test));
}

void tausche(int[] feld, int i, int j) {
   int tmp = feld[i];
   feld[i] = feld[j];
   feld[j] = tmp;
}
```

Nach der Ausführung der Methode sind die beiden ersten Werte in `test` vertauscht. Die Methode greift über die Referenz auf das in `mySend()` eingeführte Feld zu. Ähnlich ist die Situation bei der Rückgabe eines Feldes. Wird ein Feld in einer Methode angelegt und dann als Rückgabewert an die aufrufende Methode übergeben, so bleibt der Speicherplatz mit den Werten erhalten.

```
int[] legeQuadrateAn(int l) {
  int[] feld = new int[l];
  for (int i = 0; i < l; i++) {
    feld[i] = i * i;
  }
  return feld;
}

void mySend() {
    int[] q = legeQuadrateAn( 5 );
    System.out.println( Arrays.toString( q ) );
}
```

liefert die Ausgabe

```
[0, 1, 4, 9, 16]
```

Der Datentyp Feld wurde hier stellvertretend für alle komplexere Datentypen verwendet. Die vorgestellten Mechanismen gelten in gleicher Weise für allgemeine Objekte, die wir später behandeln werden. Wichtig ist die Unterscheidung zwischen den beiden Übergabeverfahren. Einfache Werte werden kopiert und Änderungen an den lokalen Kopien haben keine weiteren Auswirkungen. Bei komplexen Werten wird eine Referenz übergeben, mittels derer die Methode auf die Inhalte zugreifen und sie dauerhaft verändern kann.

7.5 Rekursion

In der Übung 4.7.3 wird mit den Fibonacci-Zahlen ein Beispiel für eine rekursive Definition gegeben. In der Bestimmungsgleichung

$$i_n = i_{n-2} + i_{n-1}$$

ist das n-te Element als Summe seiner beiden Vorgänger definiert. Um den Wert von i_n zu bestimmen ist es gemäß dieser Definition notwendig, die beiden Vorgänger zu bestimmen. Diese wiederum beruhen auf ihren eigenen Vorgängern. Auf diese Art und Weise ist es notwendig, immer weiter zurück zu gehen bis man schließlich auf einen der beiden Startwerte

$$i_1 = 1,\ i_2 = 1$$

trifft. Zur Lösung der Aufgaben wird man allerdings eher den umgekehrten Weg gehen. Ausgehend von den beiden Startwerten wird die Folge bis zu dem gesuchten Wert berechnet. Allerdings ist mit Java durchaus auch möglich, die rekursive Bestimmungsgleichung direkt in einen rekursiven Programmablauf umzusetzen. Im folgenden wird das allgemeine Konzept vorgestellt.

Erfahrungsgemäß fällt der Einstieg in die Programmierung rekursiver Methoden nicht ganz leicht. Betrachten wir daher zunächst als ganz einfaches Beispiel die Berechnung der Summe der ersten 50 Zahlen. Mit der Methode $S(N)$ als Summe der ersten N Zahlen erhalten wir

$$\begin{aligned} S(1) &= 1 \\ S(2) &= S(1) + 2 \\ S(3) &= S(2) + 3 \\ &\vdots \quad\quad \vdots \\ S(50) &= S(49) + 50 \end{aligned}$$

Dies entspricht unmittelbar der Realisierung durch eine Schleife in der Art

```
int S=0;

for (int i = 1; i <= 50; i++) {
    S = S + i;
}
System.out.println( S );
```

Der gesuchte Wert wird durch Addition aller Zahlen von 1 bis 50 berechnet. Formal lässt sich die Reihenfolge der Berechnung genauso gut umkehren. Wenn wir zunächst so tun, als wäre $S(49)$ bekannt, dann können wir einfach

$$S(50) = S(49) + 50$$

schreiben. Mit dem gleichen Argument kann $S(49)$ auf $S(48)$ zurück geführt werden. Insgesamt erhält man damit:

$$\begin{aligned} S(50) &= S(49) + 50 \\ S(49) &= S(48) + 49 \\ S(48) &= S(47) + 48 \\ &\vdots \quad\quad \vdots \\ S(2) &= S(1) + 2 \\ S(1) &= 1 \end{aligned}$$

Da die Berechnung vom Ende her beginnt, spricht man von Rekursion (lat. recurrere *zurücklaufen*). Demgegenüber wird das schrittweise Berechnen ab dem Anfang als Iteration (lat. iterare *wiederholen*) bezeichnet.

Wir haben gesehen, dass eine Methode mit lokalen Kopien der Argumenten arbeitet und alle Variablen temporärer Natur sind. Daher ist es möglich, dass eine Funktion sich selbst – oder besser gesagt eine neue Instanz von sich selbst – wieder aufruft. Bei jedem Aufruf werden neue lokale Variablen angelegt, so dass es zu keinen Zugriffskonflikten kommt. Rekursion lässt sich daher in Java leicht realisieren. Dazu programmieren wir für die Addition der Zahlen eine Methode, die entweder bei $n = 1$ den Wert 1 zurück gibt oder einen Rekursionsschritt ausführt:

```
int S(int n) {
    if( n == 1 ) {
        return 1;
    } else {
        return S(n-1) + n;
    }
}
```

Der gesuchte Wert kann dann einfach mit

```
System.out.println( S(50 ) );
```

ausgegeben werden. In dieser Form ist die Schleife verschwunden. Stattdessen steckt die Logik in der Methode. Die Methode ruft sich selbst mit einem um 1 kleineren Parameter immer wieder auf, bis irgendwann der Wert 1 erreicht ist. Dann wird rückwärts die Kette abgearbeitet.

Frage 7.3. Was passiert, wenn Abfrage nach der Endbedingung bei $n = 1$ fehlt? In diesem Fall würde sich die Methode theoretisch unendlich oft immer wieder selbst aufrufen. In der Praxis ist irgendwann der Speicher, in dem die Aufrufe abgelegt werden, voll. Es handelt sich dabei um einen so genannten Stapelspeicher (*Stack*) und die Anwendung bricht mit der Fehlermeldung *stack overflow* ab.

Diese Beispiel zeigt, wie ein Problem sowohl iterativ als auch rekursiv gelöst werden kann. In diesem Fall bietet die rekursive Lösung außer der eleganten Formulierung keinen Vorteil. Es ist sogar davon auszugehen, dass sie aufgrund der vielen Methodenaufrufe mehr Rechenzeit und Speicherplatz benötigt. Das klassische Beispiel für die Rekursion ist das Berechnen der Fakultät. Die Rekursion $n! = n \cdot (n-1)!$ mit $0! = 1! = 1$ lässt sich unmittelbar in Java umsetzen:

```
int fakultaet_r( int n ) {
    if( n == 0 || n == 1 ) {
        return 1;
    } else {
        return fakultaet_r(n - 1) * n;
    }
}
```

Wenn das Argument 0 oder 1 ist, wird direkt $0! = 1! = 1$ zurückgegeben. Ansonsten wird der aktuelle Wert mit der Fakultät des um 1 kleineren Wertes multipliziert. Beginnend mit einem positiven Wert wird diese Rekursion wiederholt, bis schließlich der immer wieder verminderte Wert auf 1 gefallen ist. Häufig findet man in solchen Fällen auch eine kompakte Schreibweise mit dem ?-Operator anstelle der if-Abfrage:

```
int fakultaet_r( int n ) {
    return ( n == 0 || n == 1 )? 1 : fakultaet_r(n - 1) * n;
}
```

7.5.1 Türme von Hanoi

Ein weiteres klassisches Beispiel für rekursive Problemlösung ist die Aufgabe *Türme von Hanoi*. Gegeben ist ein Turm aus N Scheiben wobei die Scheiben von unten nach oben kleiner werden. Die Aufgabe ist, diesen Turm von einem Punkt A nach B zu bewegen. Dabei gelten die Regeln

- es gibt einen Platz Z als Zwischenlager.
- in jedem Schritt darf eine Scheibe bewegt werden.
- dabei dürfen immer nur kleinere auf größere Scheiben gelegt werden.

Das Problem für N Scheiben kann auf ein Problem mit $N-1$ Scheiben zurück geführt werden:

- zunächst muss man die obersten $N-1$ Scheiben nach Z bringen – dabei wird B als Zwischenlager verwendet.
- dann kann die unterste, größte Scheibe an ihr Ziel B bewegt werden.

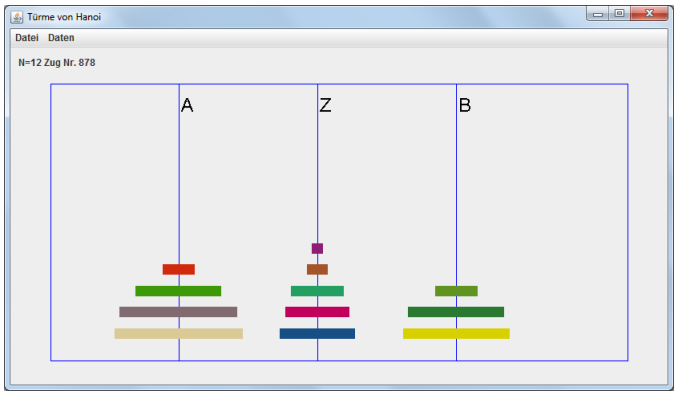


Abbildung 7.4: Zwischenstand bei *Türme von Hanoi*

- anschließend muss man den kleineren Turm von Z ans Ziel bewegen (mit A als Zwischenlager).

In Bild 7.4 ist ein Zwischenstand bei der Lösung des Problems für $N = 12$ dargestellt. Der Code in 7.1 zeigt eine Implementierung dieses Lösungsverfahrens. Für $N = 3$ ist dann die Lösung

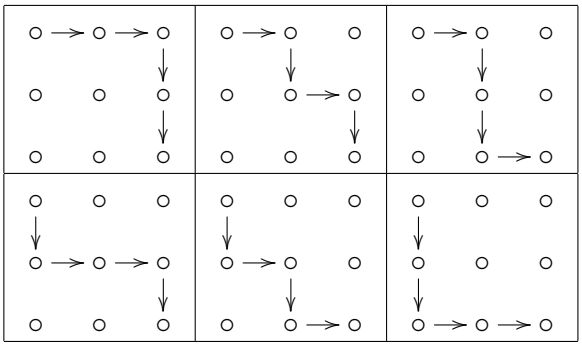
```
Schritt 1: 1-te scheibe von A nach B
Schritt 2: 2-te scheibe von A nach Z
Schritt 3: 1-te scheibe von B nach Z
Schritt 4: 3-te scheibe von A nach B
Schritt 5: 1-te scheibe von Z nach A
Schritt 6: 2-te scheibe von Z nach B
Schritt 7: 1-te scheibe von A nach B
```

7.5.2 Beispiel Wegesuche

Betrachten wir ein weiteres Beispiel. Angenommen, Sie wohnen in einer Stadt mit einem regelmäßigen, rechteckigen Straßenmuster (z. B. im Stadtteil Manhattan von New York). Wohnung und Arbeitsplatz liegen dabei auf den gegenüber liegenden Ecken eines Quadrates mit N Blocks. Sie möchten gerne die Entfernung auf möglichst vielen verschiedenen Wegen zurück legen – allerdings ohne Umwege. Nun stellt sich die Frage, wie viele derartige Wege es gibt. Etwas formaler gesprochen: vorgegeben ist ein Raster mit $N \times N$-Feldern. Wie viele mögliche Wege führen von einer Ecke zu der diagonal gegenüber liegenden Ecke? Die Wege sollen nur horizontal oder vertikal verlaufen und keine Umwege enthalten. Im Fall $N = 2$ gibt es insgesamt 6 Möglichkeiten:

Listing 7.1: Lösung Türme von Hanoi

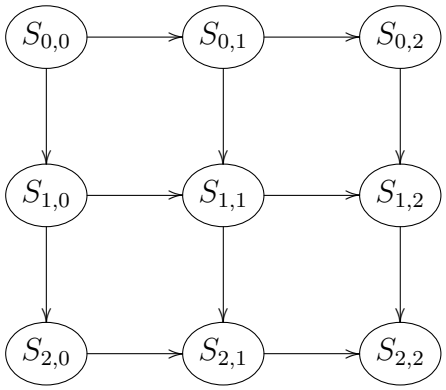
```
int schritt = 1;

void mySend() {
   lege( 3, "A", "B", "Z" );
}

void lege(int n, String von, String nach, String zwischen) {
   if (n>0) {
      lege(n-1, von, zwischen, nach);
      System.out.println("Schritt " + schritt + ": "
        + n + "-te scheibe von " + von + " nach " + nach );
      schritt++;
      lege(n-1, zwischen, nach, von);
   }
}
```



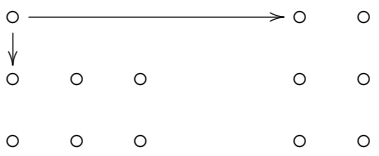
Allgemein gilt für die Anzahl N_W der Wege die Formel

$$N_W = \binom{2N}{N} = \frac{(2N)!}{N! \times N!}$$

Wir wollen allerdings die Werte per Programm berechnen lassen. Betrachten wir dazu den Fall eines Rasters mit 2×2-Feldern. Im folgenden Bild sind die Eckpunkte mit den abgehenden Möglichkeiten dargestellt.



Dabei bezeichne $S_{i,j}$ die Anzahl der Wege ab dem jeweiligen Punkt bis zum Endpunkt. Mit Ausnahme der Randpunkte gibt es an jedem Punkt zwei Verzweigungen. Der weitere Verlauf hängt nicht von dem bisherigen Weg ab. Daher kann man die weiteren Berechnungen unabhängig betrachten. Mit diesem Argument lässt sich das Hauptproblem – die Anzahl der Wege ab dem Startpunkt – in zwei kleinere Teilprobleme aufspalten. Man betrachtet jeden der beiden Wege ab dem Startpunkt für sich und berechnet die Anzahl der möglichen Wege in dem verbleibenden Rechteck. Folgendes Bild veranschaulicht diese Vorgehensweise:



Auf diese Art und Weise ist die Berechnung von $S_{0,0}$ auf die Summe der beiden Teile $S_{1,0}$ und $S_{0,1}$ vereinfacht. Diese Argument lässt sich verallgemeinern und es gilt für alle außer den Randpunkten die rekursive Beziehung

$$S_{i,j} = S_{i+1,j} + S_{i,j+1}$$

Für die Randpunkte ist die Situation einfach. Von jedem Randpunkt gibt es nur noch einen einzigen Weg – nach unten oder nach rechts, je nachdem ob man sich am rechten oder unteren Rand befindet. Damit kann man zusammenfassend schreiben:

$$S_{i,j} = \begin{cases} 1 & , \text{ wenn } i = n+1 \\ 1 & , \text{ wenn } j = n+1 \\ S_{i+1,j} + S_{i,j+1} & , \text{ sonst} \end{cases}$$

Diese Vorschrift lässt sich unmittelbar als Methode schreiben:

```
void mySend() {
   System.out.println( S( 0,0, 2 ) );
}

long S( int i, int j, int n ) {
```

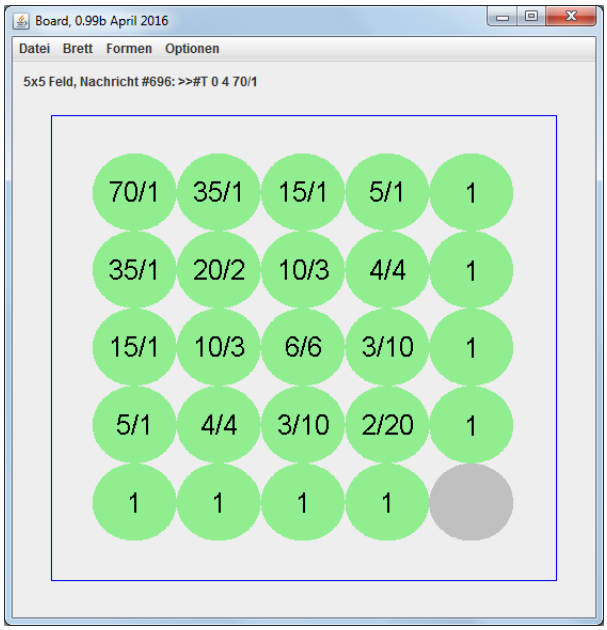


Abbildung 7.5: Anzahl von Aufrufen am Ende der Wege-Berechnung, N/M bedeutet: ab diesem Punkt gibt es N Wege, diese Information wurde M-mal abgefragt.

```
   if( i == n + 1 || j == n + 1 ) {
      return 1;
   } else {
      return S( i + 1, j, n ) + S(i, j + 1, n );
   }
}
```

In `main` wird die Methode mit dem Startpunkt $0,0$ aufgerufen. Intern werden rekursiv die Teillösungen berechnet und addiert. Diese Lösung liefert korrekte Ergebnisse. Allerdings beobachtet man ab etwa $n = 15$ extrem lange Rechenzeiten. Die Lösung ist zwar elegant aber nicht effizient. Das Problem ist, dass jede Teilsumme immer wieder neu berechnet wird. Der Punkt $1,1$ wird beispielsweise auf zwei Wegen erreicht. Daher wird auch die Funktion $S_{1,1}$ zweimal ausgewertet.

In Bild 7.5 ist die Anzahl der Aufrufe nach Ende der Berechnung dargestellt. Die erste Zahl ist die Anzahl der Wege ab diesem Punkt und die zweite die Anzahl der Aufrufe. Bei kleinem N spielt dies praktisch noch keine Rolle, aber mit wachsender Feldgröße führt diese Effekt schnell zu einem dramatischen Anstieg des Rechenaufwandes.

In Bild 7.6 ist die gemessen Rechenzeit bis $n = 16$ dargestellt. Für größere Werte von n liegt das Ergebnis nicht mehr im Bereich von `long`. In der halblogarithmischen Darstellung erkennt man deutlich den exponentiellen Anstieg der Rechenzeit. Bereits bei $n = 15$ benötigt mein Rechner mehr als 10 Sekunden[1].

[1]Die Messung liegt schon einige Jahre zurück. Mein aktueller Rechner schafft $n = 15$ in 2

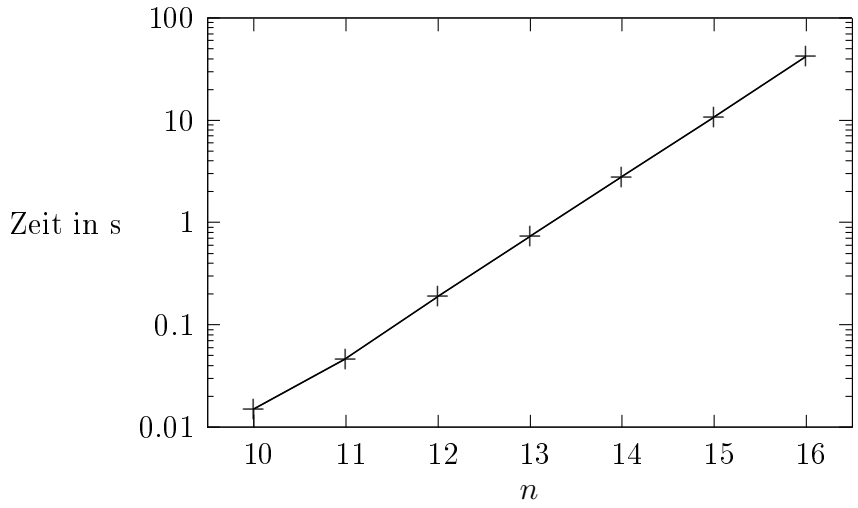


Abbildung 7.6: Gemessene Rechenzeit für rekursive Berechnung der Anzahl der Wege bei $n \times n$ Feldern.

Glücklicherweise gibt es einen einfachen Ausweg, um die Mehrarbeit zu vermeiden. Dazu wird lediglich das einmal gewonnene Wissens nicht vergessen sondern für die spätere Wiederverwendung aufgehoben. Der Code zeigt eine entsprechende Lösung:

```
long[][] bekannt;

void mySend() {
   bekannt = new long[16][16];
   System.out.println(S(0, 0, 15));
}

long S( int i, int j, int n ) {
   if (i == n + 1 || j == n + 1) {
      return 1;
   } else if( bekannt[i][j] == 0 ) {
      bekannt[i][j] = S(i + 1, j, n) + S(i, j + 1, n);
   }
   return bekannt[i][j];
}
```

Sobald ein Wert für $S_{i,j}$ feststeht, wird er in die entsprechende Zelle eines zweidimensionalen Feldes abgelegt. Wenn ein Wert für $S_{i,j}$ benötigt wird, wird zunächst

Sekunden

nachgeschaut, ob die zugehörige Zelle bereits belegt ist. Falls ja, wird einfach dieser Wert verwendet. Nur falls die Zelle noch nicht belegt ist, wird eine neue Berechnung notwendig. Mit dieser Erweiterung liefert das Programm auch für größere Felder schnell die Lösung. Auf meinem Rechner liegt die Ausführungszeit unterhalb der Messgenauigkeit.

Das beschriebene Vorgehen ist ein Beispiel für ein allgemeines Verfahren zur Lösung derartiger Probleme, der so genannten *dynamischen Programmierung*. Der Ansatz eignet sich für Probleme, die sich in viele kleine Teilprobleme zerlegen lassen, die wiederum voneinander unabhängig gelöst werden können. Typischerweise werden die Teilergebnisse in Tabellen abgelegt, auf die dann später wieder zurückgegriffen werden kann. Die Speicherung der Rückgabewerten von Methoden im speziellen nennt man Memoisation (engl. *memoization*).

7.5.3 Schleife oder Rekursion?

Rekursive Lösungen bieten sich immer dann an, wenn man ein komplexes Problem auf ein einfacheres Problem zurückführen kann. Man kann dann den Lösungsansatz direkt als Java-Code umsetzen. Die Lösung ist damit elegant und leicht verständlich. Ob die rekursive oder die (fast) immer auch mögliche iterative Lösung aufwandsgünstiger ist, muss im Einzelfall betrachtet werden.

Typisch ist diese Vorgehen für Probleme in der Art von Sudoku. Bei einigermaßen anspruchsvollen Sudoku kommt irgendwann der Punkt, wo es keine eindeutige Lösung für das nächste Feld gibt. Man schreibt dann versuchsweise eine der noch möglichen Zahlen in ein freies Feld und führt das Verfahren mit dieser Annahme weiter. Es gibt demnach eine Funktion, die ein teilweise gefülltes Sudoku erhält, einen weiteren Schritt ausführt und dann sich selbst weiter aufruft bis das Problem gelöst ist oder ein Widerspruch auftritt. Ergibt sich ein Widerspruch, so wird dieser Zweig abgebrochen und die nächste Hypothese ausprobiert. Bei der Übergabe für den nächsten Schritt kann entweder eine Kopie des aktuellen Standes verwendet werden oder die Änderung, die sich als falsch erwiesen hat, wird wieder rückgängig gemacht.

7.6 Übungen

Übung 7.6.1. Schreiben Sie nach dem Muster `waagrecht()` im Skript weitere drei Methoden `senkrecht()`. Schreiben Sie mit diesen Methoden eine neue Lösung, um zwei der Muster aus Aufgabe 2.2.1 zu zeichnen.

Übung 7.6.2. Quersumme
Gesucht sind Methoden zur Berechnung der Quersumme einer Zahl.

- Schreiben Sie eine Methode mit iterativer Berechnung.
  (Tipp: $2345 \bmod 10 = 5$)

- Schreiben Sie eine Methode mit rekursiver Berechnung (Beispiel: $Q(2345) = Q(234) + 5$).
- Testen Sie die beiden Methoden. Rufen Sie dazu die Methoden z. B. mit zufälligen Werten auf und prüfen, dass die Ergebnisse überein stimmen.

Übung 7.6.3. Wie sieht eine rekursive Lösung zur Berechnung der Fibonacci-Zahlen aus? Wie beurteilen Sie diese in Bezug auf den Rechenaufwand?

Kapitel 8

Entwicklungsumgebung

Mit größer werdenden Code-Schnipsel stoßen wir früher oder später an die Grenzen der in BoSym eingebauten Eingabemöglichkeit. Daher ist jetzt ein guter Zeitpunkt, um zu einer Entwicklungsumgebung für Java zu wechseln. In diesem Kapitel wird der Umstieg auf Eclipse beschrieben. Die Darstellung beschränkt sich auf die Grundprinzipien. Technische Details zu Programmversionen und ähnliches finden sich auf den Web-Seiten BoSym.

Eclipse ist eine integrierte Entwicklungsumgebung (*integrated development environment*, IDE) mit allen Komponenten zur Software-Entwicklung wie Editor und Compiler. Ursprünglich wurde Eclipse als Entwicklungswerkzeug für Java entwickelt. Mittlerweile werden auch andere Programmiersprachen unterstützt. Über Erweiterungen (Plug-ins) können bei Bedarf weitere Funktionalitäten hinzu gefügt werden.

8.1 Erstes Beispiel

Grundlage der Programmierung in Java sind Klassen. Bisher wurden die Code-Schnipsel automatisch in eine Klasse integriert. Über entsprechende Menüpunkte kann man neue Klassen erstellen. Wir legen als Beispiel eine Klasse `Erste` an. Klassennamen beginnen in Java üblicherweise mit einem Großbuchstaben. Wenn man noch das entsprechende Häkchen für `main` setzt, wird folgendes Gerüst generiert:

```
public class Erste {

  public static void main(String[] args) {
      // TODO Auto-generated method stub
  }
}
```

Die erste Zeile definiert die Klasse mit dem angegebenen Namen. Innerhalb des Blocks stehen dann Variablen und Methoden, die zu dieser Klasse gehören. In un-

serem Fall wurde eine Methode `main` angelegt. Diese Methode hat eine besondere Bedeutung als Einstiegspunkt. Führt man ein Klasse aus (in Eclipse Menüpunkt `run`), so wird diese Methode aufgerufen. Der Name ist fest vorgegeben. Ändert man den Namen z. B. in `main1`, so erhält man die Fehlermeldung

```
Fehler: Hauptmethode in Klasse Erste nicht gefunden.
```

(verkürzt). In dieser Methode hat Eclipse einen Kommentar mit der Kennung `TODO` eingefügt. Dies ist die Aufforderung, an dieser Stelle eigenen Code einzufügen. Der Editor zeigt diese Kommentare an, so dass man auch bei einer größeren Klasse jederzeit die noch offenen Baustellen sieht. Wir folgen der Aufforderung und ersetzen den Kommentar durch eine Ausgabe:

```
public class Erste {

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

Jetzt erscheint bei der Ausführung die Ausgabe im Konsolen-Fenster von Eclipse.

Frage 8.1. Was bedeutet `public` bei der Klasse und der Methode?

In Java kann man über Sichtbarkeitsmodifizierer festlegen, in wie weit Klassen und Methoden (und Attribute die wir später kennen lernen werden) in anderen Klassen sichtbar sind. Vorerst belassen wir es bei `public` – einer uneingeschränkten Sicht.

Frage 8.2. und `String[] args`?

Man kann beim Aufruf Werte mitgeben, die dann im Feld `args` übergeben werden. Wir werden diese Möglichkeit nicht nutzen, aber der Parameter muss trotzdem stehen bleiben.

8.2 BoSym

Auch nach dem Umstieg auf eine andere Entwicklungsumgebung können wir BoSym weiter nutzen. Allerdings müssen wir jetzt die Einbindung in eine Klasse selbst übernehmen. Die eigentliche Kommunikation mit dem Brett-Fenster übernimmt die Klasse `XSend`. Allerdings können wir sie nicht direkt ansprechen sondern benötigen als Verbindungsstück die weitere Klasse `XSendAdapter`. Das folgende Beispiel zeigt den Ablauf

```
import jserver.XSendAdapter;
```

```
public class Erste {

    public static void main(String[] args) {
        XSendAdapter sender = new XSendAdapter();
        sender.groesse(10, 10);
        sender.farbe(2, 0xff);
        sender.form(2, "*");
    }
}
```

Zunächst legen wir die Variable `sender` mit einem Objekt der Klasse `XSendAdapter` an. Über diese Variable können wir dann die bekannten Methoden aufrufen. In dem Beispiel wird die Größe des Bretts gesetzt und das Symbol 2 als blauer Stern gezeichnet. In Java sind die Klassen in Paketen (*packages*) organisiert. Für eine Klasse kann man zum Beispiel mit der Anweisung

```
package pg1;
```

festlegen, dass sie zum Paket `pg1` gehört. Klassen ohne `package`-Angabe liegen in einem unbenannten Paket (engl. *unnamed package* oder *default package*). Die Klasse `XSendAdapter` befindet sich im Paket `jserver`. Um sie zu verwenden kann man den vollen Namen in der Art

```
jserver.XSendAdapter xsend = new jserver.XSendAdapter();
```

angeben. Für eine verkürzte Schreibweise kann man andere Pakete importieren. Durch die `import`-Anweisung wird die Klasse `XSendAdapter` direkt verwendbar. Eclipse hilft dabei, die `import`-Anweisung zu verwalten. Für neu verwendete Klassen werden passende Pakete vorgeschlagen. Anderseits können nicht mehr benötigte `import`-Anweisungen automatisch entfernt werden.

Frage 8.3. Kann man die Farbnamen nicht mehr verwenden?

Doch, sie sind in der Klasse `XSendAdapter` definiert und lassen sich über den Klassennamen ansprechen:

```
xsend.farbe(1, XSendAdapter.BLUE);
```

8.2.1 Benutzereingaben

Mit dem Wechsel der Entwicklungsumgebung können wir auch interaktive Anwendungen schreiben. Für das Lesen von Eingaben kann die Bibliotheksklasse `Scanner` verwendet werden. Das folgende Beispiel zeigt die Eingabe von Feldindex und Farbe:

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

```
    System.out.print("Feld & Farbe (hex): ");
    int i = sc.nextInt();
    if (i < 0) {
        break;
    }
    int f = sc.nextInt(16);
    xsend.farbe(i, f);
}
sc.close();
```

Zunächst wird ein `Scanner`-Objekt angelegt. Dabei bezeichnet `System.in` – das Gegenstück zu `System.out` – die Eingabe in der Konsole. In einer Endlosschleife werden mit der Methode `nextInt()` die Werte für Feldindex und Farbe eingelesen. Mit dem optionalen Argument 16 legen wir fest, dass die Eingabe – passend für RGB-Werte – als Hexadezimalzahl erfolgt. Bei einem negativen Feldindex wird die Schleife mit `break` verlassen. Ansonsten werden die beiden eingelesenen Werte beim Aufruf der Methode `farbe` übergeben.

Kapitel 9

Klassen

Dieses Kapitel ist noch *in Arbeit*. Insbesondere fehlen noch Erläuterungen zu UML.

Klassen sind das wesentliche Element von Java. Selbst einfache Programme wie unser einführendes Beispiel benötigen eine Klasse - auch wenn bei der Programmierung Objektorientierung keinerlei Rolle spielt. In diesem Kapitel wird das Konzept von Klassen im Detail besprochen. Wir beginnen mit einem grundlegenden Zitat:

„*Objektorientierte Programmierung ist eine Implementierungsmethode, bei der Programme als kooperierende Ansammlung von Objekten angeordnet sind. Jedes dieser Objekte stellt eine Instanz einer Klasse dar, und alle Klassen sind Elemente einer Klassenhierarchie, die durch Vererbungsbeziehungen gekennzeichnet ist.*“

Aus Grady Booch: Objektorientierte Analyse und Design, Addison-Wesley [Boo94]

Aus diesem Zitat ergeben sich die drei Kernpunkte:

- Programme basieren auf Objekten
- Objekte sind Instanzen einer Klasse
- Zwischen den Klassen besteht eine Vererbungshierarchie

Klassen bestehen im wesentlichen aus Attributen und Methoden. Die Attribute definieren, welche Daten in einem Objekt gespeichert werden können. Durch die Methoden ist das Verhalten festgelegt. Das folgende Beispiel zeigt an Hand einer Klasse `Student`, wie durch Variablen und Methoden Eigenschaften und Verhalten spezifiziert werden.

Beispiel 9.1. Klasse `Student`
Daten:

```
String  name;
```

```
String  vorname;
int     matrikel;
int     fachsememester;
int     CP;
```

(CP steht für *Credit Points*)
Methoden:

```
int         getCP();
void        erhoeheCP( int p );
```

Eine Klasse definiert einen Datentyp. Dann können Variablen dieses Datentyps – als Objekte oder Instanzen bezeichnet – angelegt werden. Jede Instanz füllt die Variablen mit individuellen Werten. Eine Instanz der Klasse `Student` könnte sein:

```
Student miriam;
```

mit den Eigenschaften

```
miriam.vorname        = "Miriam";
miriam.name           = "Maier";
miriam.fachsememester = "2";
miriam.CP             = "30";
```

Die Methoden „hängen“ an dem Objekt. Der Aufruf erfolgt ähnlich wie der Zugriff auf die Variable mit Objektname – Punktoperator – Methodenname wie in

```
miriam.erhoeheCP( 6 );
```

oder

```
miriam.erhoeheCP( modul.getCP() );
```

Klassen in OOP sind aber nicht einfach nur Datenstrukturen mit angehängten Funktionen. Eine solche Art der Programmierung lässt sich bereits recht gut mit prozeduralen Sprachen wie C realisieren. Zur Unterscheidung spricht man in diesem Fall von objektbasierter Entwicklung. Wesentlich für OOP ist der Charakter der Klassen als Teil einer Hierarchie von Klassen.

Im folgenden werden wir zunächst den Aufbau einer einzelnen Klasse an einem konkreten Beispiel kennen lernen. Auf das Thema Vererbung werden wir dann später im Kapitel 11 zurückkommen.

Übung 9.0.1. Fußball-Turnier
Die folgende Aufzählung enthält Klassen und Objekte.

Spieler	Paris	Termin
Team	Wales	Trainer
Spielort	Cristiano_Ronaldo	Juli_7
Island	Manuel_Neuer	Portugal

Allerdings ist die Reihenfolge durcheinander gekommen. Beseitigen Sie das Chaos und ordnen – wie im eingetragenen Beispiel `Sender` – die Objekte jeweils passenden Klassen zu (es müssen nicht von jeder Klasse Objekte dabei sein).

Klasse	Objekte
Sender	ARD, ZDF, RTL

Nennen Sie noch eine weitere Klasse, die zu dem Szenario Fußball-Turnier passen würde und geben einige Objekte dazu an.

9.1 Klasse Bruch

Als Beispiel wollen wir das Rechnen mit Brüchen betrachten. Kernstück ist dabei eine Klasse – nennen wir sie `Bruch` – mit entsprechenden Eigenschaften und Verhalten. Die Eigenschaften (Attribute) sind sehr überschaubar. Ein Bruch besteht aus einem Nenner und einem Zähler. Eine Klasse wird in Java mit dem Schlüsselwort `class` definiert. Dann folgt die Aufzählung der Eigenschaften. Wir schreiben also

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

}
```

und verwenden den Datentyp `int`, um Zähler und Nenner als ganze Zahlen zu definieren.

Frage 9.1. Muss man die Attribute nicht initialisieren?

Nein, das ist nicht notwendig. In Java werden Attribute standardmäßig mit 0 (oder einem passenden Wert wie `false` bei `boolean`) initialisiert.

Frage 9.2. Warum setzen wir die Attribute auf `private`?

Das hängt mit der vorgesehenen Verwendung zusammen. Wir wollen Bruch-Objekte anlegen und mit ihnen rechnen. So soll eine Rechnung in der Art

$$\frac{2}{3} + \frac{1}{2} = \frac{7}{6}$$

ausgeführt werden. In diesem Bild ändern sich die beiden zu addierenden Brüche nicht. Vielmehr entsteht als Ergebnis ein neuer Bruch. So soll auch später unsere Klasse funktionieren: Brüche sind unveränderlich. Bei Addition zweier Brüche entsteht als Ergebnis ein neuer Bruch.

Indem wir die Sichtbarkeit auf `private` setzen, verhindern wir einen beliebigen Zugriff auf die Attribute. Natürlich benötigen wir eine Möglichkeit, die Werte trotzdem zu lesen. Üblicherweise erfolgt dies über so genannte Getter-Methoden. Dazu bildet man aus dem Namen des Attributs und der Vorsilbe *get* einen Methodennamen. Diese Methode gibt dann den Wert zurück. In unserem Fall schreiben wir

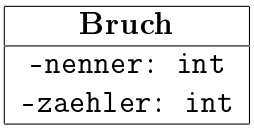
```
public int getZaehler() {
    return zaehler;
}

public int getNenner() {
    return nenner;
}
```

Im Prinzip ist der jeweilige Methodenname frei wählbar, aber viele Tools unterstützen diese Namenskonvention. Daher folgen wir ihr trotz des unschönen Sprachenmix. Wir setzen die Sichtbarkeit der Methoden auf `public`, da beide von außen beliebig aufrufbar sein sollen. Das Gegenstück zu Getter-Methoden sind Setter-Methoden, mit denen man Attributen neue Werte zuweisen kann. In unserem Fall implementieren wir keine Setter-Methoden, so dass die Inhalte nicht verändert werden können. Zusammengefasst gilt

- mit `private` verhindern wir den direkten Zugriff von außen auf die Attribute
- dann können wir über Getter- und Setter-Methoden den Zugriff nach Wahl erlauben

Durch unser Design haben wir dafür gesorgt, dass der Wert von einmal angelegten Bruch-Objekten nicht mehr geändert werden kann. Solche Klassen bezeichnet man als unveränderlich (*immutable*).

Klassen können übersichtlich als Klassendiagramme dargestellt werden. Dies ist einer der Diagrammtypen der *Unified Modeling Language* (UML). Klassen werden dabei als Rechtecke gezeichnet. Im oberen Teil steht der Name. Dann folgen weitere Rubriken mit den Attributen und Methoden. Unsere Klasse – zunächst nur mit Attributen – hat dann die Form



wobei das Minuszeichen die Sichtbarkeit `private` kennzeichnet. Mit dem Klassennamen können wir Variablen deklarieren:

```
Bruch b;
```

legt eine Variable `b` für ein Bruch-Objekt an. Allerdings wird nicht automatisch ein entsprechendes Objekt erzeugt. Vielmehr müssen wir mit dem `new`-Operator explizit ein Objekt anlegen und der Variablen `b` zuweisen:

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

oder einfacher in einer Zeile

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

Dabei ist `Bruch()` der Aufruf des so genannten Konstruktors. Jetzt wurde ein Objekt angelegt, die Attribute haben die Standardwerte 0, und `b` zeigt auf dieses Objekt. Wir können dann in der Form

```
System.out.println( b.getZaehler() + " / " + b.getNenner() );
```

den Wert des Bruchs ausgeben. Der Konstruktor `Bruch()` ist als Standard vorgegeben. Für unsere Zwecke ist er aber nicht ausreichend, da wir beim Anlegen eines neuen Bruchs die Werte für Zähler und Nenner festlegen wollen. Dazu benötigen wir einen eigenen Konstruktor. Konstruktoren werden als spezielle Form von Methoden geschrieben. Ein Konstruktor hat den Namen der Klasse und eine Liste mit Parametern. Konstruktoren haben keine Rückgabewerte, dürfen aber `return`-Anweisungen enthalten. Wir wollen unserem Konstruktor Werte für Nenner und Zähler mitgeben. Zur besseren Übersicht nennen wir die Parameter genau so wie die Attribute. Bei der Ausführung sollen dann die übergebenen Werte in die Attribute kopiert werden. Um die Attribute von den gleichnamigen Parametern zu unterscheiden, sprechen wir sie über das Objekt selbst an. Die Referenz auf das Objekt, in dem wir uns gerade befinden, ist `this`. Damit haben wir die Unterscheidung

- `nenner`: Parameter
- `this.nenner`: Attribut

Insgesamt erhalten wir damit den Konstruktor

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

Dann können wir den Konstruktor mittels `new` aufrufen. Im Beispiel

```
Bruch b = new Bruch(1, 2);
System.out.println( b.getZaehler() + " / " + b.getNenner() );
```

wird der Bruch 1/2 angelegt und anschließend ausgegeben. Sobald man einen eigenen Konstruktor geschrieben hat, wird der parameterlose Konstruktor nicht mehr angeboten. Es ist aber durchaus möglich, mehrere Konstruktoren zu schreiben. Sie müssen sich allerdings in ihrer Signatur unterscheiden. In unserem Beispiel ist der parameterlose Konstruktor nicht sonderlich sinnvoll, aber wir können mit

```
public Bruch( int zahl ) {
    this.zaehler = zahl;
    this.nenner = 1;
}
```

eine Alternative einbauen, um aus einer Zahl einen Bruch mit Nenner 1 zu erzeugen. Konstruktoren können andere Konstruktoren aufrufen. Dies geschieht wieder über das Schlüsselwort `this`, diesmal in der Form eines Methodenaufrufs. Der Konstruktur hat dann die Form

```
public Bruch( int zahl ) {
    this( zahl, 1 );
}
```

Dieser Aufruf muss die erste Anweisung im Konstruktor sein. Durch diese Verkettung von Konstruktoren vermeidet man Wiederholung von Code.

9.2 Ausgabe

9.2.1 println

Die oben verwendete Ausgabe mit Aufruf der beiden Getter-Methoden ist etwas umständlich. Einfacher wäre die direkte Ausgabe in der Form

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

Das ist durchaus möglich, allerdings ist die Ausgabe[1]

```
Bruch@15db9742
```

nicht sonderlich hilfreich. Von der Methode `println` gibt es keine spezielle Variante für Brüche. Es wird daher die am besten passende Variante `println(Object)` verwendet. `Object` ist die allgemeine Basis- oder Oberklasse in Java. Klassen wie `Bruch` ohne andere Elternklasse werden implizit direkt von `Object` abgeleitet. Damit ist jeder Bruch auch eine Instanz von `Object` und kann an dessen Stelle verwendet werden. In `println` wird dann die aus der Klasse `Object` geerbte

[1] Zahlenwerte können abweichen

Methode `toString()` aufgerufen. Die Implementierung baut aus dem Namen der Klasse, dem `@`-Zeichen und dem Hash-Code (der internen Kennung) eine Zeichenkette zur textlichen Beschreibung des Objektes zusammen. Soll eine Klasse eine passendere Beschreibung erzeugen, so muss diese Methode überschrieben werden. Dies geschieht, indem man eine Methode mit der gleichen Signatur in die eigene Klasse einbaut. In unserem Fall ist

```
public String toString() {
   return zaehler + " / " + nenner;
}
```

eine einfache Realisierung. Jetzt findet `println` bei der Ausführung die speziellere Methode und verwendet diese für die Ausgabe

```
1 / 2
```

9.2.2 BoSym

Als Alternative zur Ausgabe in der Konsole benutzen wir wieder BoSym. Mit der Methode `text` wird eine Zeichenkette in das angegebene Symbol geschrieben. Zur besseren Übersicht legen wir eine neue Klasse `Anzeiger` an, in der wir die entsprechenden Funktionalitäten bündeln.

Frage 9.3. Können wir nicht einfach eine entsprechende Methode `anzeige(Board)` in die Klasse `Bruch` aufnehmen?

Das wäre keine gute Idee. Damit würden wir eine unnötige Abhängigkeit einbauen. Die Klasse `Bruch` würde dann die Klasse `Board` verwenden. Dadurch muss jeder, der `Bruch` verwenden möchte, auch `Board` und alle dazu gehörigen Klassen übernehmen – selbst wenn der Einsatz von BoSym gar nicht geplant ist. Insgesamt würden wir die Weitergabe und Wiederverwendung der Klasse `Bruch` erschweren. Besser ist es, die Kopplung zu vermeiden und die Darstellung in eine eigene Klasse auszulagern.

Listing 9.1 zeigt diese Klasse. Sie verwendet einen `XSendAdapter`, um die bekannten Befehle an BoSym zu schicken. Im Konstruktor wird die Größe eingestellt. Die beiden Methoden `zeige` übernehmen die Darstellung eines Bruchs sowie eines beliebigen Textes. Das Attribut `pos` speichert dabei die Position zum Schreiben. Es wird automatisch nach jeder Ausgabe passend erhöht.

Nach diesen Vorbereitungen können wir eine kleine Rechenaufgabe darstellen lassen. Bild 9.1 zeigt einen entsprechenden Code-Abschnitt und die resultierende Darstellung. In diesem Beispiel werden zwei Brüche angelegt und zusammen mit Plus- und Gleichheitszeichen angezeigt.

Listing 9.1: Klasse Anzeiger

```
public class Anzeiger {
    private XSendAdapter xsend = new XSendAdapter();
    private int pos = 0;

    public Anzeiger( int n, int m ) {
        xsend.groesse(n, m);
    }

    public void zeige( Bruch b ) {
        xsend.text( pos++, " " + b.getZaehler());
        xsend.text( pos++, " / ");
        xsend.text( pos++, " " + b.getNenner());
    }
    public void zeige( String text ) {
        xsend.text( pos++, text );
    }
}
```

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

Anzeiger anzeiger = new Anzeiger(12, 1);
anzeiger.zeige(b);
anzeiger.zeige("+");
anzeiger.zeige(c);
anzeiger.zeige("=");
```

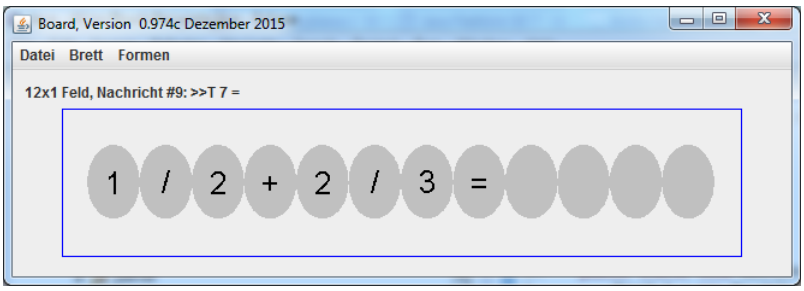


Abbildung 9.1: Darstellung von Brüchen in BoS

9.3 Methoden

Jetzt ist es an der Zeit, die Rechenaufgabe lösen zu lassen. Wir brauchen also eine Methode, um zwei Brüche zu addieren. Im Sinne der Objektorientierung haben wir folgendes Bild: gegeben ist ein Bruch-Objekt. Diesem schicken wir ein zweites Objekt mit der Aufforderung, die Addition auszuführen. Das Ergebnis soll als neuer Bruch zurückgegeben werden. Die beiden zu addierenden Brüche selbst ändern sich dabei nicht. Wir benötigen also

- eine Methode zur Addition
- diese Methode erhält einen Bruch als Parameter
- sie addiert diesen Bruch zum Empfänger
- das Resultat ist ein neues Bruch-Objekt

Eine Implementierung mit dem passenden Name `add` ist:

```
public Bruch add( Bruch b ) {
   int z = this.zaehler * b.nenner + b.zaehler * this.nenner;
   int n = this.nenner * b.nenner;
   return new Bruch( z, n );
}
```

Die beiden beteiligten Brüche sind das Objekt, in dem wir uns befinden (`this`), sowie der übergebene Summand (`b`). Aus diesen beiden Summanden werden die Werte für Zähler und Nenner der Summe berechnet. Die Kennung (`this`) könnte man in diesem Fall auch weglassen, da die Bezeichnungen eindeutig sind. Schließlich wird eine neue Instanz mit den berechneten Werten angelegt und zurückgegeben. Im Klassendiagramm wird die Methode in einer weiteren Rubrik in der Form

Bruch
-nenner: int -zaehler: int
+add(Bruch): Bruch

eingetragen. Die neue Zeile besagt, dass `add` eine öffentliche Methode ist, einen `Bruch` als Parameter erwarten und einen `Bruch` zurück gibt. Mit der neuen Methode können wir unser Beispiel durch

```
Bruch s = b.add( c );
anzeiger.zeige(s);
```

ergänzen. Die resultierende Darstellung 9.2 zeigt jetzt die vollständige Rechnung.

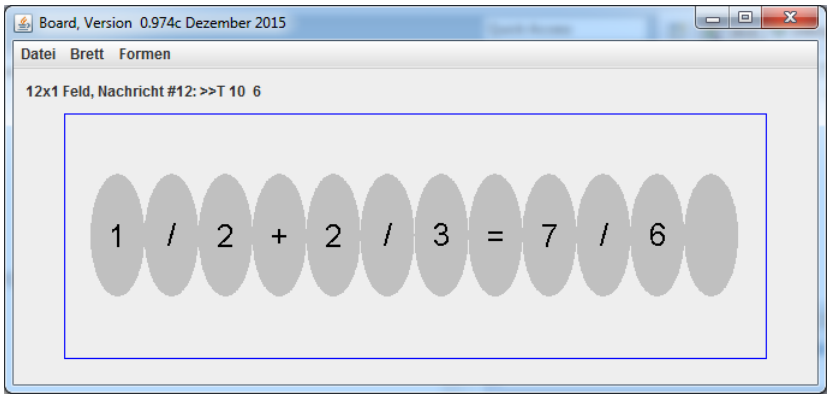


Abbildung 9.2: Bruch-Addition in BoSym

9.3.1 Beispiel Harmonische Reihe

Mit der Addition von Brüchen können wir bereits ein erstes Beispiel rechnen lassen. Wir betrachten dazu die Harmonische Reihe. Sie ist gemäß

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

definiert als Summe der Kehrwerte der positiven ganzen Zahlen. Mit der neuen Klasse sollen die Werte von H_n als Brüche berechnet werden. Das Vorgehen ist:

- die Summe wird mit 0 initialisiert
- dann werden die Brüche $\frac{1}{n}$ nacheinander von $n = 1$ bis zu einem Maximum erzeugt
- die Brüche werden zur Summe addiert und ausgegeben

Das lässt sich wie folgt in Java umsetzten:

```
Bruch summe = new Bruch( 0 );
for ( int n=1; n<=10; n++ ) {
    summe = summe.add( new Bruch( 1, n ) );
    System.out.println( "H(" + n + ") = " + summe );
}
```

Dabei ist das Maximum auf 10 gesetzt und die Ausgabe

```
H(1) = 1 / 1
H(2) = 3 / 2
H(3) = 11 / 6
H(4) = 50 / 24
H(5) = 274 / 120
H(6) = 1764 / 720
```

```
H(7) = 13068 / 5040
H(8) = 109584 / 40320
H(9) = 1026576 / 362880
H(10) = 10628640 / 3628800
```

resultiert. Wir können die Ausgabe erweitern, indem wir zusätzlich den Wert des Bruchs als Dezimalzahl ausgeben lassen. Die Berechnung des Dezimalwertes aus Nenner und Zähler wird wahrscheinlich häufiger benötigt. Daher lohnt es sich, dafür eine Methode zu schreiben. Wir ergänzen daher die Klasse `Bruch` durch

```
public double toDouble() {
    return (double) zaehler / nenner;
}
```

und können dann bei der Ausgabe

```
System.out.println( "H(" + n + ") = " + summe
       + " = " + summe.toDouble() );
```

schreiben. Zur Erinnerung: der cast-Operator `(double)` erzwingt die Umwandlung von `zaehler` in einen double-Wert. Dies muss vor der Division erfolgen. Ansonsten würde die Division als Integer-Rechnung durchgeführt werden, und erst das Ergebnis – ohne Nachkommastellen – würde dann in den geforderten double-Wert gewandelt werden. Mit diesen Änderungen erhalten wir die Ausgabe

```
H(1) = 1 / 1 = 1.0
H(2) = 3 / 2 = 1.5
H(3) = 11 / 6 = 1.8333333333333333
H(4) = 50 / 24 = 2.0833333333333335
H(5) = 274 / 120 = 2.283333333333333
H(6) = 1764 / 720 = 2.45
H(7) = 13068 / 5040 = 2.592857142857143
H(8) = 109584 / 40320 = 2.717857142857143
H(9) = 1026576 / 362880 = 2.828968253968254
H(10) = 10628640 / 3628800 = 2.9289682539682538
```

9.4 Klassen-Methoden und -Variablen

Beim Vergleich unserer Werte $H(n)$ mit z.B. den Angaben bei Wikipedia fallen Unterschiede auf. So wird dort $H(5) = 137/60$ angegeben. Wir haben demgegenüber noch den Faktor 2 in Nenner und Zähler. Es fehlt nach der Addition das Kürzen auf die Grunddarstellung. Dazu benötigt man die Bestimmung des größten gemeinsamen Teilers (ggT) von Nenner und Zähler. Ein effizientes Verfahren dazu ist der euklidische Algorithmus. Wir wollen hier nicht weiter auf den Algorithmus eingehen, sondern direkt zur Implementierung übergehen:

```
static public int euclid(int n, int m) {
    if (m == 0)
        return n;
    return euclid(m, n % m);
}
```

Die Methode erhält als Parameter zwei ganze Zahlen und berechnet dann mittels Modulo-Rechnung und Rekursion den gesuchten ggT. Interessant ist in unserem Zusammenhang vielmehr der zusätzliche Modifizierer `static`.

Frage 9.4. Was ist an dieser Methode anders oder besonders?

Im Gegensatz zu allen anderen bisherigen Methoden der Klasse `Bruch` bezieht sich diese Methode nicht auf die Attribute einer Instanz. Die anderen Methoden folgten immer dem Bild einer Nachricht an eine Instanz, die dann entsprechend reagiert. Offensichtlich ergibt z. B. `toDouble()` nur in einer Bruch-Instanz einen Sinn. Demgegenüber kann `euclid` gut auch ohne eine Bruch-Instanz aufgerufen werden. Sie erhält alle Informationen als Parameter und verändert auch nichts am Zustand eines Bruchs. Durch die Angabe `static` wird sie zu einer Klassenmethode im Gegensatz zu den Objektmethoden (auch als Instanzmethoden bezeichnet). Klassenmethoden werden direkt über die Klasse aufgerufen

```
System.out.println( Bruch.euclid(12, 8) );
```

ohne dass vorher ein Objekt generiert wurde. Formal ist allerdings auch der Aufruf über ein Objekt in der Art

```
Bruch b = new Bruch( 24, 6);
System.out.println( b.euclid(12, 8) );
```

erlaubt. Diese Form ist allerdings eher verwirrend. Schließlich hat der Bruch 24/6 nichts mit der Berechnung des ggT zu tun. Daher sollte man Klassenmethoden auch über den Klassennamen aufrufen. Mit der neuen Methode erweitern wir den Konstruktor, so dass der Bruch bei jeder Initialisierung in seine Grunddarstellung gebracht wird:

```
public Bruch(int zaehler, int nenner) {
    int teiler = euclid(zaehler, nenner);
    this.zaehler = zaehler / teiler;
    this.nenner = nenner / teiler;
}
```

Wie dieses Beispiel zeigt, kann man innerhalb von Instanzmethoden auf Klassenmethoden zugreifen. Umgekehrt geht es nicht: Innerhalb einer Klassenmethode befinden wir uns außerhalb aller Instanzen und können damit nicht auf Instanzmethoden oder Instanzvariablen zugreifen.

Frage 9.5. Was wären die Folgen, wenn `euclid()` nicht `static` wäre?

In diesem Beispiel wäre der Schaden nicht allzu groß. Am Aufruf im Konstruktor würde sich nichts ändern. Nur beim Aufruf aus anderen Klassen müsste man den Umweg über eine Instanz gehen. Wenn eine Methode logisch zu einer Klasse gehört, allerdings keinerlei Verbindung zu einer Instanz hat, sollte sie auch konsequenterweise als Klassenmethode angelegt werden. Das gilt insbesondere, wenn sie in anderen Klassen genutzt wird. Dies bietet auch einen Schutz gegen Fehler. Wenn eine Methode als Klassenmethode konzipiert ist und dann – versehentlich durch einen Programmierfehler – auf Instanzmethoden zugreift, wird beim Übersetzen ein Fehler angezeigt.

Klassenmethoden können auch als Alternativen zu Konstruktoren zur Erzeugung von Objekten eingesetzt werden. Für Testzwecke ist es oft hilfreich, Objekte mit mehr oder weniger zufälligen Werten zu generieren. Die folgende Methode

```
static public Bruch zufallsBruch( int max ) {
    int z = (int) (Math.random() * max );
    int n = (int) (Math.random() * max );
    return new Bruch(z, n );
}
```

übernimmt diese Aufgabe. Bei jedem Aufruf erzeugt sie einen neuen Bruch, wobei Nenner und Zähler zufällig gewählte, ganze Zahlen aus dem Intervall $[0, max]$ sind. Sie kann dann in der Art

```
Bruch z = Bruch.zufallsBruch(20);
```

alternativ zu einem Konstruktor verwendet werden.

9.4.1 Klassenvariablen

Jede Instanz hat einen eigenen Satz von Attributen als Instanzvariablen. In manchen Fällen gibt es auch gemeinsame Eigenschaften, die sich alle Instanzen teilen. Dazu kann man wieder mit dem Attribut `static` eine Variable als Klassenvariable deklarieren. Eine solche Variable gibt es dann nur einmal innerhalb der Klasse. Der Zusammenhang zwischen Methoden und Variablen einer Klasse und ihrer Instanzen ist in Bild 9.3 graphisch dargestellt.

Als Beispiel ergänzen wir unsere Klasse um einen Zähler für die Anzahl der angelegten Instanzen. Ein solcher Zähler kann durchaus für Testzwecke hilfreich sein. Wir definieren ihn mit

```
private static int anzahlInstanzen;
```

wobei er mit `private` wieder vor Zugriff von außen geschützt wird. Im Konstruktor ergänzen wir eine Anweisung zum Inkrementieren der Variablen:

```
public Bruch(int zaehler, int nenner) {
    int teiler = euclid(zaehler, nenner);
    this.zaehler = zaehler / teiler;
```

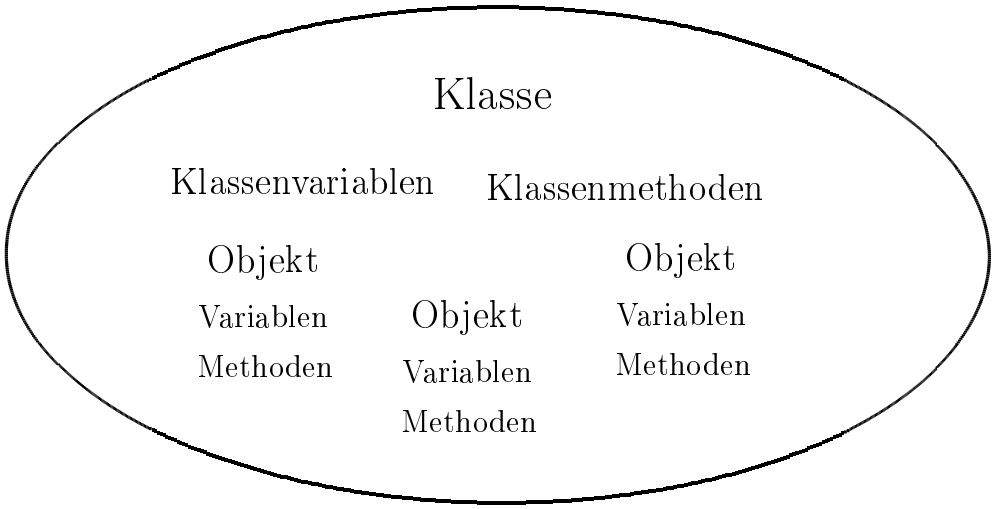


Abbildung 9.3: Objekt- und Klassenvariablen und Methoden

```
    this.nenner = nenner / teiler;
    ++anzahlInstanzen;
}
```

Für den lesenden Zugriff von außerhalb der Klasse stellen wir einen Getter bereit:

```
public static int getAnzahlInstanzen() {
    return anzahlInstanzen;
}
```

Der Code-Abschnitt

```
System.out.println( Bruch.getAnzahlInstanzen() + " Bruch Instanzen" );
Bruch t1 = new Bruch( 2, 3 );
Bruch t2 = new Bruch( 2, 3 );
Bruch t3 = new Bruch( 2, 7 );
System.out.println( Bruch.getAnzahlInstanzen() + " Bruch Instanzen" );
```

liefert dann die Ausgabe

```
0 Bruch Instanzen
3 Bruch Instanzen
```

Die Definition von allgemeinen Konstanten ist ein anderes Beispiel für Klassenvariablen. Beim Rechnen mit Brüchen werden wahrscheinlich recht oft die Werte 0 oder 1 benötigt. Wir definieren zwei entsprechende Konstanten mit

```
public static final Bruch NULL = new Bruch( 0 );
```

```
public static final Bruch EINS = new Bruch( 1 );
```

Dabei gilt

- mit `static` legen wir die Konstanten als Klassenvariablen fest
- durch `final` werden sie unveränderlich
- in diesem Fall ist ein allgemeiner Zugriff mit `public` angebracht
- die Namen von Konstanten werden nach Java-Konvention in Großbuchstaben geschrieben.

Die Berechnung der Harmonischen Reihe lässt sich dann als

```
Bruch summe = Bruch.NULL;
for ( int n=1; n<=100; n++ ) {
    summe = summe.add( new Bruch( 1, n ) );
}
```

umschreiben. Die Konstanten verbessern die Lesbarkeit des Codes. Außerdem spart man Aufwand durch die mehrfache Verwendung des gleichen Objektes. In Listing 9.2 ist der bis hier erreichte Stand unserer Klasse `Bruch` vollständig dargestellt.

Listing 9.2: Klasse Bruch

```
public class Bruch {
    public static final Bruch NULL = new Bruch( 0 );
    public static final Bruch EINS = new Bruch( 1 );
    private int nenner;
    private int zaehler;
    private static int anzahlInstanzen;

    public Bruch(int zaehler, int nenner) {
        int teiler = euclid(zaehler, nenner);
        this.zaehler = zaehler / teiler;
        this.nenner = nenner / teiler;
        ++anzahlInstanzen;
    }

    public Bruch(int zahl) {
        this( zahl, 1 );
    }

    public int getZaehler() { return zaehler; }
    public int getNenner() { return nenner; }
```

```
    public static int getAnzahlInstanzen() {
        return anzahlInstanzen;
    }

    public String toString() {
        return zaehler + " / " + nenner;
    }
    public double toDouble() {
        return (double) zaehler / nenner;
    }

     public Bruch add(Bruch b) {
        int z = this.zaehler * b.nenner + b.zaehler * this.nenner;
        int n = this.nenner * b.nenner;
        return new Bruch(z, n);
    }

    static public int euclid(int n, int m) {
        if (m == 0)
            return n;
        return euclid(m, n % m);
    }

    static public Bruch zufallsBruch( int max ) {
        int z = (int) (Math.random() * max );
        int n = (int) (Math.random() * max );
        return new Bruch(z, n );
    }
}
```

Frage 9.6. Ist die Reihenfolge der Deklarationen wichtig?

Im Prinzip kann man die Deklarationen frei innerhalb der Klasse verteilen. Allerdings sollte man im Sinne der Lesbarkeit eine gewisse Struktur einhalten. Im Beispiel ist die Reihenfolge

1. Konstanten
2. Klassenvariablen
3. Instanzvariablen
4. Konstruktoren
5. Methoden

angelehnt an [R+00].

9.5 Beispiel Kapselung

Wir betrachten als Anwendung unserer Bruch-Klasse eine spezielle Darstellung der Zahl π als Produkt von Brüchen:

$$\frac{\pi}{2} = \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \ldots$$

(Wallis-Produkt). Die folgende Methode berechnet mit einer zusätzlichen Methode `mpy()` in der Klasse `Bruch` nach dieser Formel die ersten 60 Werte der Näherung:

```
void wallis() {
    Bruch product = Bruch.EINS;
    int n = 1;
    int z = 2;
    for ( int i=1; i<60; i++ ) {
        Bruch f = new Bruch( z, n);
        product = product.mpy( f );
        System.out.println( (i-1) + ": " + product + " "
                            + 2 * product.toDouble() );
        if( i % 2 == 0 ) {
            z += 2;
        } else {
            n += 2;
        }
    }
}
```

Als Ausgabe erhält man

```
0: 2 / 1 4.0
1: 4 / 3 2.6666666666666665
2: 16 / 9 3.5555555555555554
3: 64 / 45 2.8444444444444446
...
31: 4611686018427387904 / 2980705490751054825 3.09435872328693
32: -9223372036854775808 / 6129560826237051145 -3.009472390705362
```

Die Methode rechnet richtig, aber nach der 31. Multiplikation wird der Zähler größer als die größte int-Zahl und das Ergebnis liegt im negativen Bereich. Wir müssen daher von int auf einen anderen Datentyp wechseln, der auch größere Werte enthält. In Java gibt es eine Bibliotheksklasse `BigInteger` für beliebig große (oder kleine) ganzzahlige Werte. Ohne in die Details zu gehen: mit relativ wenig Aufwand kann man in der Klasse `Bruch` die Attribute `nenner` und `zaehler` in diesen Typ ändern. Der folgende Ausschnitt vermittelt einen Eindruck der geänderten Klasse:

```
public class Bruch {
       public static final Bruch NULL = new Bruch(0);
       public static final Bruch EINS = new Bruch(1);
...
       private BigInteger nenner;
       private BigInteger zaehler;

       public Bruch mpy(Bruch b) {
              BigInteger z = this.zaehler.multiply(b.zaehler);
              BigInteger n = this.nenner.multiply(b.nenner);

              return new Bruch(z, n);
       }
...
```

Die Struktur – beispielsweise die Namen der Methoden – bleibt erhalten. Lediglich die Implementierungsdetails müssen angepasst werden. Bleibt die Frage: Was muss in der Methode `wallis()` angepasst werden? Die einfache Antwort lautet: Nichts. Die Methode verwendet nur die öffentlichen Methoden der Klasse. Deren inneres Funktionieren hat sich zwar geändert, aber nicht ihr Verwendung. Daher können wir `wallis()` unverändert belassen und der Aufruf mit der überarbeiteten Klasse liefert dann das gewünschte Ergebnis:

```
...
31: 461168601842738790 4/2980705490751054825 3.09435872328693
32: 922337203685477580 8/5786075364399106425 3.18812716944714
33: 184467440737095516 16/11912508103174630875 3.09703782174865
...
```

9.6 Klassen für primitive Datentypen

In Java stehen für die elementaren Datentypen wie Zahlen oder Zeichen primitive Datentypen wie `double` oder `char` zur Verfügung. Für diese sind Rechen- und Vergleichsoperatoren definiert. Allerdings sind es keine Klassen und die Werte sind keine Objekte. Gelegentlich benötigt man allerdings doch Objekte und nicht nur die Werte. Im Kapitel 13 werden wir Datenstrukturen kennen lernen, die als Elemente Objekte erwarten. Für solche Fälle stellt Java so genannte Wrapper-Klassen zur Verfügung. Wie der Name sagt, bilden sie Objekt-Hüllen um die Werte. Die entsprechenden Klassen tragen den jeweils gleichen Namen, dann allerdings der Namenskonvention entsprechend mit einem Großbuchstaben am Anfang. So gehört zu `double` die Klasse `Double`. Lediglich `Integer` für `int` und `Character` für `char` weichen von dieser Namenskonvention ab. Wie üblich kann man dann Instanzen mittels `new` erzeugen:

```
Integer v = new Integer( 12 );
```

Die Objekte haben – wie bei `String` und unserem Beispiel `Bruch` – einen festen, unveränderlichen Wert. Daher ist es nicht sonderlich effizient bei gleichem Wert jeweils ein neues Objekt anzulegen. Die Konstruktoren gelten daher als veraltet (*depracted*) und sollten durch die jeweilige Klassenmethode `valueOf()` ersetzt werden:

```
Integer v = Integer.valueOf( 12 );
```

Die Methode kann dann feststellen, ob es ein entsprechendes Objekt bereits gibt und gegebenenfalls direkt die Referenz darauf zurückgeben. Bei häufigerem Wechsel zwischen primitiven Datentypen und Wrapper-Objekten führt entsprechender Code schnell zu unübersichtlichem Code. Daher kann der Java-Compiler mit *Autoboxing* und *Autounboxing* dieses Aufgabe übernehmen. Man kann daher einfach

```
Integer v = 12;
```

schreiben. Durch die automatischen Konvertierungen kann man primitive Datentypen und Wrapper-Objekten ziemlich frei mischen. Einige Beispiele zeigt

```
Double a = 2.;
Double b = 3.;
System.out.println( "a: " + a );
++a;
System.out.println( "a: " + a );
System.out.println( "a + b: " + (a + b) );
System.out.println( "a < 10.: " + (a < 10.) );
System.out.println( "a == b: " + (a == b) );
System.out.println( "a equals b: " + a.equals( b ) );
double c = b + 20;
System.out.println( "c: " + c );
```

mit den Ausgaben

```
a: 2.0
a: 3.0
a + b: 6.0
a < 10.: true
a == b: false
a equals b: true
c: 23.0
```

Das Mischen von `Double` Objekten und `double` Werten und Operatoren funktioniert gut. Lediglich beim Vergleich muss man unterscheiden zwischen dem Vergleich auf gleiche Objekte mit `==` und dem Vergleich auf gleichen Inhalt mit `equals()`.

9.7 Lokale Klassen und Aufzählungstypen

Der Vollständigkeit halber sollen an dieser Stelle kurz lokale Klassen eingeführt werden. Lokale Klassen werden innerhalb einer anderen Klasse definiert (innerhalb des äußersten {}-Blocks). Sie sind nur innerhalb der äußeren Klasse sichtbar haben aber umgekehrt vollen Zugriff auf diese Klasse.

Typen, die nur eine bekannte, überschaubare Anzahl von Werten annehmen können, lassen sich durch Enumerations definieren. Ein Beispiel aus der Klasse `PatternGenerator` mit einer Aufzählung der verschiedenen Modi ist

```
enum Mode {
       SINGLE, MULTI, TRIANGLE, FRAME, ARROW, STAIRWAY
}
```

Nach dieser Definition können Instanzen von `Mode` in der Art

```
Mode m = Mode.STAIRWAY;
```

angelegt werden. In diesem Fall wird beim Aufruf der Methode zum Erzeugen eines Musters

```
generate(Mode mode) {
```

ein Parameter diesen Typs übergeben. In einer if-else Konstrukt wird dann in Abhängigkeit vom Modus der entsprechende Zweig ausgewählt:

```
if (mode == Mode.SINGLE) {
       xsa.farbe2(sstart, zstart, color);

} else if (mode == Mode.MULTI) {
       for (int z = zstart; z < N; z += zinc) {
...
```

Die Werte einer Enumeration werden als Text abgezeigt. So ergibt

```
Mode m = Mode.STAIRWAY;
System.out.println(m);
System.out.println( Arrays.toString( Mode.values() ) );
```

die Ausgabe

```
STAIRWAY
[SINGLE, MULTI, TRIANGLE, FRAME, ARROW, STAIRWAY]
```

Eine Variable kann auch nur die definierten Werte annehmen. Der Versuch

```
m = 12;
```

führt zu einer Fehlermeldung `incompatible types`. Dadurch sind Enumerations in der Verwendung übersichtlicher, sicherer und transparenter als die Alternative mit Konstanten in der Art

```
public static final int SINGLE = 0;
public static final int MULTI = 1;
...
```

und sollten in den meisten Fällen bevorzugt werden.

Kapitel 10

Zeichenketten

10.1 Einleitung

Die Basis für die Verarbeitung von Zeichen ist der Datentyp `char` (Kurzform für *character*). Natürlich ist es möglich, Felder von einzelnen Zeichen anzulegen. In Java gibt es darüber hinaus eine eigene Klasse `String` für Zeichenketten. Die Verwendung von `String` Instanzen erleichtert wesentlich die Programmierung. Java stellt eine ganze Reihe von Methoden unter anderem zum Suchen in Zeichenketten sowie zum Vergleichen und Verändern von Zeichenketten zur Verfügung.

Aufgrund der großen praktischen Bedeutung der String-Verarbeitung wird die Klasse `String` bevorzugt behandelt. Der Compiler kennt den internen Aufbau und nutzt diese Kenntnis, um den Code zu optimieren. Weiterhin gibt es für diese Klasse einen speziellen Verknüpfungsoperator.

10.2 Datentyp char

Einzelne Zeichen werden in Java durch den Datentyp `char` dargestellt. Die Konstanten werden in einfache Anführungsstriche gesetzt:

```
char zeichen = 'x';
```

Dabei wird für die Zeichenkodierung [C+12] Unicode verwendet, so dass auch nationale Sonderzeichen erfasst sind. Steuerzeichen wie der Zeilenumbruch werden durch ein vorgestelltes \-Zeichen notiert:

`\n`	*newline* Zeilenumbruch
`\t`	Horizontaler Tabulator
`\'`	Einfaches Anführungszeichen
`\\`	*Backslash*
`\uXXXX`	Unicode-Zeichen mit den vier Hexadezimalziffern XXXX

Intern ist ein `char`-Wert der Index in einer Zeichentabelle und daher eine ganze, positive Zahl. Java erlaubt es, direkt mit diesen Index-Werten zu arbeiten. Folgender Code-Abschnitt zeigt einige Möglichkeiten:

```
char u = 'A';
System.out.print( u );
System.out.print(" " + ++u );
u += '0';
System.out.print(" " + u );
u += Character.getNumericValue( '5' );
System.out.print(" " + u );
u = 77;
System.out.println( " " + u );
```

mit der Ausgabe

```
A B r w
```

Mit dem Operator `++` wurde der Index erhöht und deutet jetzt auf das nächste Zeichen in der Tabelle. In Zeile 4 werden nicht die nummerischen Werte sondern die Indizes addiert. Ein zuverlässige Methode, um den nummerischen Wert eines Zeichens zu bestimmen, ist die Bibliothek-Methode in Zeile 6. Diese Methode erkennt auch andere als arabische Ziffern (z. B. römische), für die es eigene Unicode-Zeichen gibt. Weiterhin können `char`-Werte untereinander und mit ganzen Zahlen verglichen werden. So ist

```
u >= 'A' & u <= 'Z'
```

ein Test auf Großbuchstaben. Das funktioniert aufgrund der Anordnung der Zeichen in Unicode, ist aber beschränkt auf die Standardbuchstaben und wird z. B. bei Umlauten scheitern. Besser ist auch in diesem Fall die Verwendung eine der zahlreichen Methoden der Klasse `Character`:

```
Character.isUpperCase( u )
```

Zeichen lassen sich leicht in BoSym darstellen. Mit dem Funktionenpaar

```
zeichen(int i, char c)
zeichen2(int x, int y, char c)
```

können einzelne Zeichen auf das Brett geschrieben werden. Die Zeichen werden jeweils zentriert auf das angegebene Feld gesetzt. Das folgende Beispiel nutzt die Methoden zur Darstellung einiger Unicode-Zeichen:

```
groesse( 8, 4 );
formen( "s" );
farben( 0xe8e8e8 );

getBoard().receiveMessage( ">>fontsize 32" );
```

Abbildung 10.1: Darstellung von Zeichen

```
getBoard().receiveMessage( ">>fonttype Dialog" );

//Lateinischer Großbuchstabe E mit Hatschek
zeichen( 0, '\u011A' );
//Griechischer Großbuchstabe Delta
zeichen( 1, '\u0394' );
//Kyrillischer Großbuchstabe abchasisches Cha
zeichen( 2, '\u04A8' );
// römische Zahl
zeichen( 3, '\u216C' );

// CJK-Ideogramme
for ( int z=0; z<24; z++ ) {
   zeichen(z + 8, (char)('\u3400' + z));
}
```

Bild 10.1 zeigt die resultierende Darstellung.

Frage 10.1. Was bedeutet `getBoard().receiveMessage(">>fonttype Dialog")`?

Unicode definiert den Vorrat an Schriftzeichen. Um ein Zeichen darstellen zu können, benötigt man zusätzlich eine grafische Darstellung (Glyphe). Diese Information liegt in einem Font. Allerdings enthält nicht jeder Font Informationen zu allen Zeichen. So fehlen im Standard-Font Arial die CJK[1]-Zeichen. Mit dem Befehl `fonttype` wird BoSym der Name eines neuen Fonts (genauer der Name der Font-Familie) geschickt. In diesem Fall ist die Font-Familie Dialog angegeben, mit der zumindest auf meinem Rechner die Zeichen dargestellt werden. Der Befehl gehört zu einer Gruppe von selten verwendeten BoSym-Befehlen, für die keine eigene Methode implementiert wurde. Daher muss der Befehl über eine allgemeine Nachrichtenmethode geschickt werden. Details dazu findet man im Anhang A.1.

[1] chinesische, japanische und koreanische Schrift

10.3 Konstruktoren für String

Eine Referenz der Klasse `String` wird wie bei jeder anderen Klasse durch die Anweisung in der Form

```
String text;
```

angelegt. Nach dieser Anweisung ist `text` eine Referenz auf ein `String`-Objekt. Entweder direkt bei der Definition oder später an beliebiger Stelle kann die Instanz dann mit einer Zeichenketten-Konstanten (Literal) durch einfache Zuweisung belegt werden:

```
text = "Frankfurt";
```

Dadurch wird ein `String`-Objekt mit dem entsprechenden Text als Inhalt erzeugt und die Referenz auf dieses Objekt gesetzt. Der folgende Code zeigt ein einfaches Beispiel mit einem String, der nacheinander mit zwei verschiedenen Inhalten gefüllt wird.

```
@Complete
String text;

void mySend() {
    System.out.println( "text: " + text );
    text = "München";
    System.out.println( "text: " + text );
    text = "Frankfurt";
    System.out.println( "text: " + text );
}
```

Die Ausgabe lautet:

```
text: null
text: München
text: Frankfurt
```

Bei der ersten Ausgabe ist die Referenz noch leer -– angezeigt durch den speziellen Wert `null`. Bei der Zuweisung wird automatisch jeweils ein ausreichender Speicherbereich reserviert. Diese Speicherverwaltung übernimmt das Java-Laufzeitsystem. So ist es kein Problem, dass bei der zweiten Zuweisung der Text länger ist. Zur Vollständigkeit seien hier noch die beiden BoSym-Methoden zur Ausgabe von Strings erwähnt:

```
text(int i, String t)
text2(int x, int y, String t)
```

Man könnte das obige Beispiel in der Art

```
@Complete
String text;

void mySend() {
   groesse( 3, 3 );
   text2( 1, 2, text );
   text = "München";
   text2( 1, 1, text );
   text = "Frankfurt";
   text2( 1, 0, text );
}
```

für die Ausgabe mit BoSym umschreiben. Die Klasse String enthält weiterhin noch eine Reihe von Konstruktoren. Typisch ist der Konstruktor

```
public String(char[] zeichen)
```

Der Konstruktor erhält ein Feld von Zeichen. Aus diesen Zeichen bildet er ein `String`-Objekt mit der entsprechenden Zeichenkette. Ein Beispiel dazu ist:

```
char[] zeichen = { 'a', 'b', 'c' };
text = new String( zeichen );
```

Dann enthält `text` die Zeichenkette „`abc`“. Dieses Vorgehen kann nützlich sein, um Texte automatisch zu erzeugen. Dabei ist zu beachten, dass die Zeichen kopiert werden. Spätere Änderungen im Feld wirken sich nicht mehr auf das `String`-Objekt aus Andere Konstruktoren erlauben es, nur einen Ausschnitt des Feldes mit Zeichen zu verwenden oder anstelle der Zeichen byte-Werte zu übergeben.

10.4 Länge von Zeichenketten und einzelne Zeichen

Die erzeugten String-Objekte können wie andere Objekte benutzt werden. Insbesondere gibt es eine ganze Reihe von Methoden, um mit Strings zu arbeiten. Die Länge der Zeichenkette erhält man mit der Methode `length()`. Der Aufruf `text.length()` gibt die Länge – d. h. die Anzahl der enthaltenen Zeichen – zurück. Einzelne Zeichen aus der Zeichenkette kann man mit `charAt( int index )` abfragen. Wie bei Feldern beginnt die Zählung mit dem Index 0. Die folgenden Programmzeilen bilden eine Schleife zur zeichenweise Ausgabe:

```
for ( int i=0; i<text.length(); i++ ) {
   System.out.println( i + ": " + text.charAt(i) );
}
```

10.5 Arbeiten mit Zeichenketten

Die Zeichenketten in `String`-Objekten sind konstant. Nach der Initialisierung in einer der oben beschriebenen Formen liegt der Text fest und kann nicht mehr verändert werden. Auf den ersten Blick erscheint dies als eine große und wenig einleuchtende Einschränkung. Schließlich ist die Veränderung von Zeichenketten eine wesentliche Aufgabe vieler Anwendungen.

Der Widerspruch löst sich auf, wenn man sich klar macht, dass man eigentlich stets mit den Referenzen auf die `String`-Objekte arbeitet. Ein Text wird dann verändert, indem ein neues Objekt angelegt wird und anschließend die Referenz auf diese neue Objekt gesetzt wird. Das alte Objekt bleibt unverändert bestehen.

Betrachten wir als Beispiel die Anwendung der Methode `trim()` in der Klasse `String`. Diese Methode hat die Aufgabe, Leerzeichen am Anfang und Ende einer Zeichenkette zu entfernen. Betrachten wir als Beispiel

```
String  text = "  München  ";
```

einen Text mit Leerzeichen. Der Aufruf `text.trim()` erzeugt daraus ein neues `String`-Objekt, füllt es mit dem Text ohne die Leerzeichen und gibt es zurück. Mit der Zuweisung

```
text = text.trim();
```

wird die Referenz auf das neue Objekt gesetzt. Auf diese Art und Weise wird das Ziel erreicht: `text` enthält jetzt den gewünschten Text ohne Leerzeichen. Intern existieren jetzt zwei Objekte: der ursprüngliche String mit den Leerzeichen und ein neuer String ohne Leerzeichen. Falls das erste Objekt – mit den Leerzeichen – auch nicht mehr von anderen Referenzen angesprochen wird, ist es Aufgabe des Garbage Collectors dieses Objekt zu löschen und den Speicherplatz wieder frei zu geben. Der Ablauf sieht zusammen gefasst wie folgt aus:

	Ref.		Objekte
1.)	text	→	`"  München  "`
2.)	text	→	`"  München  "`
		`trim()` →	`"München"`
3.)			`"  München  "`
	text	→	`"München"`

Ist keine Veränderung notwendig, weil keine Leerzeichen an den Ränder stehen, so wird auch kein neues Objekt benötigt. Der Aufruf gibt dann einfach die Referenz auf das bestehende Objekt zurück. Der Programmierer braucht sich um diese Details nicht zu kümmern. Aus seiner Sicht ist mit der obigen Anweisung der Text verändert worden. Die Verwaltung der Objekte kann er Java überlassen. Wichtig ist nur, dass er die „Falle“ vermeidet, lediglich die Methode aufzurufen ohne das neue Objekt bzw. die neue Referenz anschließend einer Variablen zuzuweisen. Der Aufruf der Methode `text.trim()` alleine verändert nicht den Inhalt von `text`. Die intuitive Anwendung der `String`-Objekte zeigt folgendes Beispiel:

```
text = "Frankfurt";
text += " am Main";
System.out.println( "text: " + text );
```

Mittels des Operators zur Verkettung von Strings wird scheinbar die Angabe `" am Main"` direkt angehängt. Intern ist der Vorgang komplexer. Ausführlich geschrieben lautet die Anweisung

```
text = text + " am Main";
```

Aus den beiden Strings – der Variablen `text` und dem Literal `" am Main"` – wird ein neues `String`-Objekt erzeugt. Die Referenz auf das neue Objekt wird dann wieder `text` zugewiesen.

Diese Verarbeitung – Erzeugung eines neuen `String`-Objektes für jede Veränderung – bedeutet einen gewissen Mehraufwand. In den meisten Anwendungen ist dies jedoch nicht relevant. Ansonsten besteht noch die Möglichkeit, die Klasse `StringBuffer` zu verwenden, die dynamisch veränderbare Zeichenketten realisiert.

10.6 Teilketten

Die Methode zum Ausschneiden von Teilen aus Zeichenketten (*substrings*) ist

```
String substring( int beginIndex, int endIndex)
```

Sie liefert die Teilkette ab der Position des ersten Arguments bis zur Position vor dem zweiten Argument. So ergibt der Aufruf

```
System.out.println( "Die Wetterau".substring(4,10) );
```

den Text `Wetter`. Das Verhalten zeigt folgendes Bild:

D	i	e		W	e	t	t	e	r	a	u
0	1	2	3	4	5	6	7	8	9	10	11
				↑						↑	

Die Zeichen von Index 4 bis 9 bilden den neuen `String`. Dadurch, dass die zweite Angabe die Position 10 unmittelbar nach dem Ende der Teilkette bezeichnet, gilt der Zusammenhang

```
beginIndex + Länge = endIndex
```

wodurch sich in vielen Fällen die Berechnung der Endposition vereinfacht. Ein zweite Version von `substring` mit nur einem Argument liefert die Zeichenkette bis zum Ende des ursprünglichen Strings. Damit ergibt

```
System.out.println( "Die Wetterau".substring(4) );
```

den Text `Wetterau`.

10.7 Vergleichen und Suchen

10.7.1 Vergleichen

Der Vergleich zweier Strings mit dem == Operator testet, ob beide Referenzen auf das gleiche Objekt deuten. In der Regel ist dies eine zu starke Forderung. Meist interessiert nur, ob die beiden Strings die gleiche Zeichenkette enthalten. (Zur Erinnerung: bei primitiven Typen vergleicht der Operator == den Inhalt, bei Referenztypen die Referenz.) Die allgemeine Methode zum Vergleichen der Inhalte ist

```
boolean equals(Object anObject)
```

Die Methode liefert `true` wenn das Argument ein String ist und die gleiche Zeichenkette enthält.

```
text = "Frankfurt";
System.out.println( text.equals( "Frankfurt") );
```

Der Vergleich ergibt `true`. Der folgende Code zeigt den Unterschied zwischen beiden Vergleichsweisen.

```
String s1, s2;
char[] feld = {'a', 'b', 'c'};

// 2 verschiedene String-Objekte mit gleichem Inhalt
s1 = new String( feld );
s2 = new String( feld );
System.out.println( s1 + " " + s2 );
System.out.println( s1 == s2 );
System.out.println( s1.equals( s2 ) );

// 2 Referenzen auf gleiche String-Konstante
s1 = "def";
s2 = "def";
System.out.println( s1 + " " + s2 );
System.out.println( s1 == s2 );
System.out.println( s1.equals( s2 ) );
```

Im ersten Teil werden durch explizite Verwendung von Konstruktoren zwei String-Objekte mit dem gleichen Inhalt erzeugt. Dementsprechend ergibt der erste Vergleich ein `false`, `s1` und `s2` referenzieren unterschiedliche Instanzen. Der Vergleich des Inhalts liefert dann allerdings `true`.

Im zweiten Teil verweisen `s1` und `s2` auf einen String-Literal. In der internen Verwaltung der String-Literale wird bei Bedarf eine String-Instanz mehrfach verwendet. Bei der zweiten Angabe von `def` wird erkannt, dass es eine ensprechende Zeichenkette bereits gibt. Daher wird keine neue – doppelte und daher unnötige

– Instanz angelegt. Somit beziehen sich `s1` und `s2` tatsächlich auf das gleiche Objekt und der Vergleich mit `==` liefert ein `true` zurück. Selbstverständlich ist das Objekt auch inhaltsgleich mit sich selbst und auch der letzte Vergleich trifft zu.

Die Methode `equals` vergleicht auf exakte Übereinstimmung. In dem Beispiel hätte `text.equals( "frankfurt")` das Resultat `false`. Ein Vergleich ohne Berücksichtigung von Groß- und Kleinschreibung bietet

```
boolean equalsIgnoreCase(String anotherString)
```

Das Methoden-Paar

```
boolean startsWith(String prefix)
boolean endsWith(String suffix)
```

erlaubt gezielte Vergleiche mit dem Anfang beziehungsweise Ende der Zeichenkette. Die Details zu diesen Methoden sowie weiteren Varianten findet man in der Dokumentation zu Java. Für z. B. die Implementierung von Sortierverfahren wird häufig ein Vergleich von Strings auf ihre lexikalischen Reihenfolge benötigt. Mit den Methoden

```
int compareTo(String str)
int compareToIgnoreCase(String str)
```

wird ein Integer-Wert berechnet, der den lexikalischen Abstand anzeigt. Das Resultat des Ausdrucks `text.compareTo(str)` ist:

negativ	`text` kleiner als `str`
0	`text` gleich `str`
positiv	`text` größer als `str`

10.8 Verändern von Zeichenketten

Einige Methoden erzeugen ein neues `String`-Objekt mit geändertem Text. Umwandlung in Groß- oder Kleinbuchstaben übernehmen

```
String toUpperCase()
String toLowerCase()
```

Als Beispiel liefert

```
System.out.println("Münchner Straße".toUpperCase() );
```

```
MÜNCHNER STRASSE
```

Die Methoden erkennen auch Sonderzeichen wie Umlaute oder ß und behandeln sie richtig. Daneben gibt es noch eine einfache Methode zum Ersetzen von Zeichen:

```
String replace(char oldChar,  char newChar)
```

Damit werden alle vorkommenden Zeichen `oldChar` durch `newChar` ersetzt. Das Beispiel

```
String text = "Die Wetterau";
// alle e durch x ersetzen
System.out.println( text.replace( 'e', 'x' ) );
```

ergibt

```
Dix Wxttxrau
```

Anstelle einzelner Zeichen kann man auch Zeichenketten ersetzen lassen:

```
String email = "euler@mnd.thm.de";

// Ersetze mnd durch noe :
String email2 = email.replace("mnd", "noe");
System.out.println( email2 );
```

Ausgabe:

```
euler@noe.thm.de
```

Bei dieser Methode werden nur exakte Übereinstimmungen behandelt. Flexible Suchmuster kann man mit sogenannten regulären Ausdrücken angeben. Ein regulärer Ausdruck beschreibt eine Menge von Zeichenketten. Es handelt sich um ein allgemeines Konzept, das in vielen Programmiersprachen und auch Editoren umgesetzt ist. An dieser Stelle sollen nur einige Grundregeln angegebenen werden:

- „normale“ Zeichen (Buchstaben, Ziffern) stehen für sich selbst
- der Punkt ist ein Platzhalter für ein beliebiges Zeichen
- mit eckigen Klammern kann für ein Zeichen eine Auswahl angegeben werden (Beispiele: `[aeio]`, `[0-9]`)
- mehrere Zeichen werden mit runden Klammern zusammen gefasst
- der Stern bedeutet: das was davor steht, kann hier beliebig oft (auch 0-mal) stehen

Hierzu gibt es das Methoden-Paar

```
String replaceAll(String regex, String replacement)
String replaceFirst(String regex, String replacement)
```

Dabei werden alle beziehungsweise nur das erste Vorkommen des regulären Ausdrucks `regex` durch den String im zweiten Argument ersetzt.

Beispiel 10.1. Ersetzen mit Regulären Ausdrücken

```
String email = "euler@mnd.thm.de";

// Ersetze alle Felder durch den Text xxx.
// Ein Feld ist:
// "ein Buchstabe gefolgt von 0 oder mehr Buchstaben"
String email3 = email.replaceAll("[a-z][a-z]*", "xxx");

System.out.println( email3 );
```

Ausgabe:

```
xxx@xxx.xxx.xxx
```

10.8.1 Suchen

Die Methode zum Suchen in Strings ist `indexOf`. Es gibt davon verschiedene Varianten, je nachdem ob man nach anderen Strings oder einzelnen Zeichen suchen möchte. Als optionales zweites Argument kann man einen Offset, ab dem erst die Suche beginnen soll, übergeben. Weiterhin gibt es Methoden `lastIndexOf`, bei denen die Suche am Ende des Strings beginnt. Alle Methoden geben die gefundene Position als Integerwert zurück. Wird das Muster nicht gefunden, so ist der Rückgabewert –1.

Beispiel 10.2. Suche in Strings

```
String text2 = "In der schönen Wetterau";
//              0123456789012345678901
//                        1111111111222
```

Methodenaufruf	Ergebnis
`text2.indexOf('e')`	4
`text2.indexOf("Wett")`	15
`text2.indexOf("WETT")`	-1
`text2.lastIndexOf("e")`	19
`text2.lastIndexOf("e", 19)`	19
`text2.lastIndexOf("e", 18)`	16
`text2.indexOf(' ', text2.indexOf(' ')+ 1)`	6

10.8.2 Aufspalten

Häufig soll eine Zeichenkette – z. B. eine eingelesene Textzeile – für die weitere Analyse in einzelne Bestandteile aufgeteilt werden. Hierfür kann die Methode

`split` verwendet werden. Sie erwartet als Argument einen regulären Ausdruck. Dann wird die Zeichenkette nach entsprechenden Mustern durchsucht und an den Treffern aufgeteilt. Die gefundenen Teile werden als Feld von Strings zurück gegegeben. Ein Beispiel mit der Aufteilung an Leerzeichen:

```
String zeile = "Vom Eise befreit sind Strom und Bäche";
String[] woerter = zeile.split(" ");
for ( String w: woerter ) {
   System.out.println( w );
}
```

mit der Ausgabe

```
Vom
Eise
befreit
sind
Strom
und
Bäche
```

Übung 10.8.1. Split:
Was erhält man, wenn in der Zeichenkette mehrere Leerzeichen aufeinander folgen (z. B. `"Vom  Eise  befreit ..."`)? Wie kann man das Verhalten für solche Fälle verbessern?

Ein anderes Beispiel ist das Auftrennen einer Email-Adresse an den Trennzeichen `.` und `@`. Dies wird elegant mit einer Zeichenauswahl in der Art

```
String[] teile = email.split("[@.]");
```

gelöst.

10.9 Konvertierungen

Für z. B. die Ausgabe ist es notwendig, andere Objekte oder primitiven Datentypen in Zeichenketten umzuwandeln. Alle Objekte implementieren eine Methode `toString()`, die eine Darstellung als Zeichenkette liefert. Weiterhin stellt die Klasse `String` Klassenmethoden `valueOf()` zum Konvertieren der primitiven Datentypen bereit. Explizit kann man dann zur Ausgabe schreiben:

```
double x = 33.44;
System.out.println( "x = " + String.valueOf(x) );
```

Der Ausdruck `String.valueOf(x)` liefert einen String, der wiederum mit dem ersten String verbunden wird. In solchen Ausdrücken braucht die Umwandlung nicht explizit angegeben zu werden. Man kann einfacher schreiben

```
double x = 33.44;
System.out.println( "x = " + x );
```

und die Konvertierung wird automatisch eingefügt.

10.10 Die Klasse String ist endgültig

Die Klasse `String` enthält bereits sehr viele Methoden. Trotzdem sind nicht alle Anwendungen abgedeckt. So fehlt beispielsweise für spezielle Fälle eine Methode `startsWithIgnoreCase`. Im Sinne der Objekt-orientierten Programmierung würde man in einem solchen Fall eine eigene Klasse aus der `String`-Klasse ableiten und die fehlenden Methoden ergänzen beziehungsweise vorhandene Methoden mit eigenen Implementierungen überlagern.

Diese Vorgehensweise ist für die Klasse `String` nicht vorgesehen. Diese Klasse hat das Attribut `final`. Damit wird verhindert, dass abgeleitete Klassen erzeugt werden. Der Hauptgrund ist ein Gewinn an Performanz. Wir werden später sehen, wie abgeleitet Klassen die Erzeugung von Code schwieriger machen. Wenn es aber keine abgeleiteten Klassen gibt, kann der Compiler effizienteren Code erzeugen. Neue Methoden lassen sich dann nur in anderen Klassen einbauen.

10.11 Übungen

Übung 10.11.1.
```
String text =  "Die Wetterau";
String text2 = "In der schönen Wetterau";
//              0123456789012345678901
//                        1111111111222
```

1. Was liefert `text.substring(7)`?
2. Was ist der Unterschied zwischen `text.charAt( 0 )` und `text.substring(0,1)`?
3. Erzeugen Sie aus `text` durch Anfügen den neuen String `" ** Die Wetterau ** "`
4. Was ergibt `text2.indexOf( "n W" )`?
5. Wie kann man in dem String nach dem ersten `"er"` irgendwo nach einem `"W"` suchen?
6. Testen Sie, ob ein String mit einem `?` endet.

Übung 10.11.2. Tannenbaum
Verwenden Sie folgende Klasse als Startpunkt:

```
public class Strings {
     public static void main(String[] args) {
        String text = "Tannenbaum";
     }
}
```

Lassen Sie dann eine neue Zeichenkette mit dem Text umgewandelt in Großbuchstaben erzeugen. Geben Sie dann in einer Schleife unter Verwendung der Methode `substring` den Text wie folgt aus:

```
T
TA
TAN
TANN
TANNE
TANNEN
TANNENB
TANNENBA
TANNENBAU
TANNENBAUM
```

Die Ausgabe soll für beliebigen Inhalt der String-Variablen `text` funktionieren.

Lassen Sie nun in einer weiteren Variablen den umgekehrten Text zusammenbauen (Tipp: Buchstaben einzeln entnehmen und neu zusammensetzen). Mit dem Beispieltext soll die Variable dann den Inhalt `MUABNENNAT` haben. Erweitern Sie dann die bereits vorhandene Schleife zu folgender Ausgabe

```
          TT
         ATTA
        NATTAN
       NNATTANN
      ENNATTANNE
     NENNATTANNEN
    BNENNATTANNENB
   ABNENNATTANNENBA
 UABNENNATTANNENBAU
MUABNENNATTANNENBAUM
```

Übung 10.11.3. Reguläre Ausdrücke:
Wie lassen sich mit der Methode `replaceAll` alle Folgen von mehreren Leerzeichen in jeweils ein einziges Leerzeichen verkürzen? Zum Beispiel soll aus

```
"Dies   ist   ein Beispiel"
```

die Zeichenkette `"Dies ist ein Beispiel"` werden.

Übung 10.11.4. Telefonnummer:

1. Wie kann man mit `split` die Telefonnummer
   `String telefon = "(49)6031.604-450";`
   in die Bestandteile zerlegen?

Übung 10.11.5. Umwandlung:
Wie ist die Ausgabe von `System.out.println( "7 + 5 = " + 7 + 5 )`?

Übung 10.11.6. Interaktiv
Im CodeWindow kann man im Menü Compiler - Neu interactive folgenden Code erzeugen lassen:

```
@Complete
void mySend() throws InterruptedException {
   board.receiveMessage(">>showInput");
   for ( ;; ) Thread.sleep( 100 );
}

public void receiveCommand( String s ) {
       // hier eigenen Code einbauen
       text2(3,3, "::" + s + "::");
}
```

Beim Ausführen wird in BoSym ein Eingabefeld eingeblendet. Mit dem Knopf `senden` wird der dort eingetragen Text an die Methode `receiveCommand` geschickt. In der Vorlage wird dieser Text zusammen mit den :-Zeichen wieder ausgegeben. Ändern Sie die Methode, so dass durch Wiederholung des Musters ein Passwort gegebener Länge erzeugt wird. Die Länge soll in der Variablen `n` gespeichert sein. Für das Muster `THM` und drei Beispiele für `n` gilt:

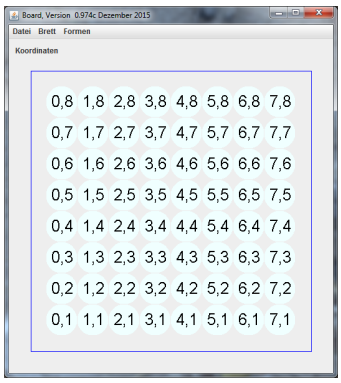
```
n: 3 passwort: THM
n: 5 passwort: THMTH
n: 9 passwort: THMTHMTHM
```

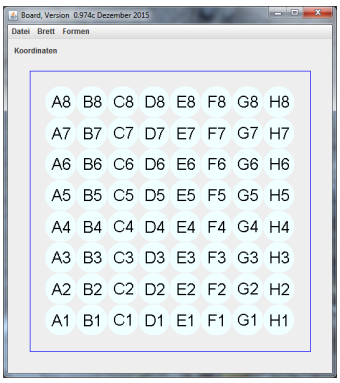
Übung 10.11.7. Passwort-Generator
Ein Programm soll mit Hilfe der Methode `Math.rand()` zufällige Passworte generieren. Jedes Passwort ist 8 Zeichen lang. Realisieren Sie folgende Versionen von Passworten:

1. 8 Buchstaben (Beispiel `ahgtzjui`)
2. 3 Buchstaben 1 Ziffer 3 Buchstaben 1 Ziffer (Beispiel `hgt4klx1`)
3. 1 Konsonant 1 Vokal 1 Konsonant 1 Ziffer 1 Konsonant 1 Vokal 1 Konsonant 1 Ziffer (Beispiel `hag6pox0`)

`Hinweis`: Legen Sie Strings mit den jeweiligen Zeichen (Vokale, Konsonaten) an. Lassen Sie dann an den einzelnen Stellen immer zufällig ein Zeichen aus dem jeweiligen Feld aussuchen.

Übung 10.11.8. Text-Muster
Schreiben Sie Anweisungen, um folgende Muster zu erzeugen:





Verwenden Sie dabei die BoSym-Methoden `text` oder `text2`, um Zeichenketten an entsprechende Positionen zu schreiben.

Kapitel 11

Vererbung

In diesem Kapitel kehren wir zu den Grundlagen der Objektorientierten Programmierung zurück. Bisher hatten wir noch keine Vererbung verwendet. Dies soll nun nachgeholt werden. Unser bisheriges Beispiel *Brüche* bietet sich dafür allerdings nicht so sehr an. Stattdessen gehen wir zunächst zurück zum Beispiel *Hochschule*. Wir werden dazu eine Klassenhierarchie für verschiedene Personen an einer Hochschule entwickeln. Allerdings werden wir uns auf den Entwurf der Klassen beschränken und sie nicht anwenden. Daher werden wir ein zweites Beispiel *geometrische Formen* betrachten, bei dem die Klassen auch zum Zeichnen einfacher Bilder verwendet werden.

11.1 Beispiel Hochschule

In unserer Beispielanwendung Hochschule können wir verschiedene Menschen wie Studenten, Tutoren, Professoren, Mitarbeiter und Lehrbeauftrage unterscheiden. In Kapitel 9 hatten wir bereits eine Klasse `Student` betrachtet. Das Beispiel

Student
`vorname: String`
`name: String`
`matrikel: int`
`fachsemester: int`
`CP: int`
`getCP(): int`
`erhoeheCP( int p ): void`

enthält einige Grundmerkmale sowie beispielhaft Methoden zum Abfragen und zum Erhöhen der Anzahl der Credit Points. Ähnlich können wir eine Klasse `Professor` definieren:

Professor
`vorname: String`
`name: String`
`dienstjahre: int`
`haeltVorlesung( Vorlesung v ): void`

Beim Vergleich fällt auf, dass beide Klassen einige Eigenschaften teilen. Sowohl `Student` als auch `Professor` enthalten die Attribute für Vorname und Name. Demgegenüber hat nur ein Student eine Matrikelnummer und Credit Points. Der Verbleib an der Hochschule wird bei Studenten in Fachsemestern gezählt während bei Professoren die Dienstjahre gelten.

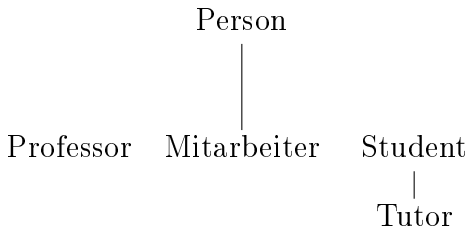
Zur Vereinfachung kann man die gemeinsamen Attribute und Methoden der beiden Klassen in eine allgemeinere Klasse auslagern. In unserem Fall bietet es sich an, Name und Vorname in eine allgemeinere Klasse – nennen wir sie `Person` – zu verschieben. Im weiteren würden wir auch zusätzliche Informationen, die jede Person hat, dort einbauen. Beispiele hierfür sind Geburtsdatum, Anschrift und Geschlecht. Durch Vererbung übernehmen die beiden Klassen `Student` und `Professor` dann alle Attribute und Methoden von `Person`. In diesem Bild ist `Person` die Elternklasse (auch Super-, Ober- oder Basisklasse) und die beiden anderen sind jeweils eine Kindklasse (auch Sub-, Unter- oder abgeleitete Klasse).

Person
`vorname: String`
`name: String`

Student
`matrikel: int`
`fachsemester: int`
`CP: int`
`getCP(): int`
`erhoeheCP( int p ): void`

Professor
`dienstjahre: int`
`haeltVorlesung( Vorlesung v ): void`

Die Klassen-Hierarchie bildet einen Baum vom Allgemeinen zum Speziellen:



Es gibt eine für alle gemeinsame Basisklasse. Jede andere Klasse hat eine direkte Elternklasse. Klassen übernehmen Eigenschaften und Verhalten von Oberklassen, spezialisieren oder ergänzen sie und geben sie an Unterklassen weiter. Sprachlich kann man die Beziehung mit „ist ein“ ausdrücken:

- ein Student *ist eine* Person.
- ein Professor *ist eine* Person.

Die „ist ein“ Beziehung gilt nicht nur für die direkt übergeordnete Klasse, sondern auch für alle anderen übergeordneten Klassen im entsprechenden Zweig:

- ein Tutor *ist ein* Student.
- ein Tutor *ist eine* Person.

In dieser Hierarchie werden Eigenschaften und Verhalten entlang eines Zweiges vererbt. Im Beispiel kann `Tutor` alle Attribute von `Student` übernehmen und um seine speziellen Eigenschaften ergänzen. Beispielsweise kann die Klasse `Tutor` eine zusätzliche Komponente zur Angabe der Tutoren-Schulung einführen. Diese Eigenschaft spielt bei den anderen Klassen keine Rolle.

11.2 Umsetzung in Java

Wir beginnen mit der allgemeinen Klasse:

```
package hochschule;

class Person {
  private String vorname;
  private String name;

  public Person(String vorname, String name) {
      this.vorname = vorname;
      this.name = name;
  }

  public String getName() {
      return name;
  }

  public String getVorname() {
      return vorname;
  }
```

```
  @Override
  public String toString() {
      return vorname + " " + name;
  }
}
```

Bei dieser Klasse haben wir keine Elternklasse angegeben. Java setzt dann automatisch die Klasse `Object` ein. Von `Person` leiten wir die beiden anderen Klassen ab. In Java erfolgt dies mit dem Schlüsselwort `extends` und dem Namen der Elternklasse. Konstruktoren werden nicht vererbt. Allerdings kann man mit `super()` einen Konstruktor der Elternklasse aufrufen. Wir können daher die Arbeit aufteilen:

- Der Konstruktor der Elternklasse initialisiert die Eltern-Attribute.
- Der Konstruktor der Kindklasse übernimmt dann alles Weitere.

Zur besseren Übersichtlichkeit implementieren wir nur das Attribut `CP` und erhalten dann:

```
package hochschule;

public class Student extends Person {
    int CP = 0;

    public Student(String name, String vorname, int CP) {
        super(name, vorname);
        this.CP = CP;
    }
}
```

Entsprechend gilt für Professoren:

```
package hochschule;

public class Professor extends Person {
    int dienstjahre;

    public Professor(String name, String vorname, int dienstjahre) {
        super(name, vorname);
        this.dienstjahre = dienstjahre;
    }
}
```

Nach diesen Vorbereitungen können wir Objekte anlegen und ausgeben lassen:

```
Student s = new Student( "Miriam", "Maier", 30);
```

```
Professor p = new Professor( "Leonard", "Gauss", 10);
Person m = new Person( "Jonas", "Schneider");

System.out.println( s );
System.out.println( p );
System.out.println( m );
```

Bei der Ausgabe wird die Methode `toString()` aufgerufen. Diese hatten wir in der Klasse `Person` eingetragen. Sie überlagert dort die bereits in `Object` definierte Version. Die beiden Kindklassen haben keine eigenen Versionen und verwenden daher die geerbte. Als Ausgabe resultiert:

```
Miriam Maier
Leonard Gauss
Klaus Bayer
```

Für eine differenzierter Ausgabe können wir wiederum in den Kindklassen `toString()` überschreiben. So können wir in `Student` die Ausgabe der Credit Points realisieren:

```
@Override
public String toString() {
   return getVorname() + " " + getName() + ", " + CP + " CPs";
}
```

Da wir in der Elternklasse die Attribute auf `private` gesetzt hatten, müssen wir sie über die Getter abfragen. Eleganter ist die Wiederverwendung der schon in der Elternklasse vorhandenen Methode. Diese kann über `super` aufgerufen werden und wir können die zurückgegebene Zeichenkette wie folgt einbauen:

```
return super.toString() + ", " + CP + " CPs";
```

Die Ausgabe wird dann zu

```
Miriam Maier, 30 CPs
Leonard Gauss
Jonas Schneider
```

Die Zeile

```
@Override
```

ist eine so genannte Annotation – erkennbar an dem führenden `@`-Zeichen. In diesem Fall ist ein optionaler Hinweis an den Kompiler. Wenn eine Methode so gekennzeichnet ist, prüft er dass auch tatsächlich eine Methode in einer Elternklasse überschrieben wird. Gibt es keine solche Methode, resultiert ein Kompiler-Fehler. Damit besteht ein Schutz z. B. gegen Tippfehler im Methodennamen. Ohne die Annotation würde der fehlerhafte Name akzeptiert. Die Methode könnte verwendet werden, hätte aber nicht die gewünschte Vererbungsbeziehung.

Zwischen den abgeleiteten Klassen und den Elternklassen besteht eine *ist ein* Beziehung. In unserem Beispiel gilt *ein Student ist eine Person*. Daher kann ein Objekt einer abgeleiteten Klasse – das ja alle Eigenschaften der Elternklasse aufweist – die Rolle eines Objektes der Elternklasse annehmen. Somit kann ein Objekt von `Student` einer `Person`-Variablen zugewiesen werden:

```
Person n = new Student( "Sabine", "Bayer", 4);
```

Damit stellt sich bei der Ausgabe

```
System.out.println( n );
```

die Frage, welches `toString()` aufgerufen wird – `Person` oder `Student`? Java interpretiert die Methodenaufrufe zur Laufzeit. Bei jedem Aufruf wird das Objekt untersucht und die am besten passende Methode ausgewählt. Man spricht daher von dynamischem oder spätem Binden. In dem Beispiel stellt der Interpreter fest, dass es sich um ein `Student`-Objekt handelt und ruft die Methode aus der Klasse `Student` auf. Dieses durchaus wünschenswerte Verhalten bedeutet allerdings einen gewissen Mehraufwand während der Ausführung. Bei jeder Variablen muss geprüft werden, welches Objekt aus dem entsprechenden Vererbungsbaum aktuell referenziert wird. Aus diesem Grund wurde die Klasse `String` als `final` deklariert. Damit ist klar, dass es keine abgeleiteten Klassen gibt und die Suche entfällt.

Allgemein kann innerhalb eines Zweiges der Vererbungshierarchie ein Objekt aus einer tieferen Ebene anstelle eines aus einer höheren Ebene verwendet werden. Umgekehrt geht es nicht:

```
Student ni = new Person( "Jonas", "Schneider");
```

führt zur Fehlermeldung

```
Type mismatch: cannot convert from Person to Student
```

Genauso wenig kann man zwischen den Zweigen wechseln:

```
Student ni = new Professor( "Leonard", "Gauss", 10);
```

ist ebenfalls ein `Type mismatch`. Anschaulich ist das nachvollziehbar: wenn nur irgendeine Person gebraucht wird, so ist ein Student oder auch ein Professor oder ... ausreichend. Umgekehrt, wenn ein Student gebraucht wird muss es auch ein Student (oder ein Tutor) sein. Ein Professor hat nicht alle Eigenschaften eines Students und kann daher nicht vollständig dessen Rolle übernehmen.

Frage 11.1. Ist es denn überhaupt sinnvoll, Instanzen der Klasse `Person` anzulegen?

Wahrscheinlich nicht – `Person` ist eine abstraktes Konzept. Niemand an einer Hochschule ist eine Person, sondern immer Student, Mitarbeiter, Professor, etc.. Wenn wir dies in unserer Klasse abbilden wollen, verwenden wir den Modifikator `abstract`:

`abstract class Person {`

Der Compiler gibt dann bei

`Person m = new Person( "Jonas", "Schneider");`

die Fehlermeldung

`Cannot instantiate the type Person`

aus.

11.2.1 Modifikatoren

Die Sichtbarkeit, Lebensdauer und Ableitbarkeit von Klassen, Methoden und Variablen kann durch verschiedene Modifikatoren bei der Definition festlegen werden. Die wichtigsten sind (*default* bedeutet *keine Angabe*):

`public`	Die weitestgehende Festlegung. Variablen, Methoden und Klassen sind überall sichtbar.
`private`	Nur die aktuelle Klasse sieht die Variablen oder Methoden.
`protected`	Abgeleitete Klassen und Klassen im gleichen Paket sehen die Variablen oder Methoden.
default	Klassen im gleichen Paket sehen die Variablen oder Methoden.
`static`	Definition von Klassenvariablen und -methoden
`final`	Klassen mit dem Attribut `final` können nicht abgeleitet, Methoden nicht überlagert und Variablen nicht verändert werden.
`abstract`	Abstrakte Methoden enthalten keinen Rumpf und können nicht selbst aufgerufen werden (Syntaxbeispiel: `abstract void test();`). Sie dienen lediglich als Vorlagen für die Methoden in den abgeleiteten Klassen. Eine Klasse mit abstrakten Methoden ist selbst abstrakt und kann nicht instantiiert werden.

Verschiedene Attribute können miteinander kombiniert werden.

11.3 Geometrische Formen

Als zweites Beispiel für Vererbung behandeln wir das Zeichnen mit geometrische Formen. Zur Darstellung greifen wir auf die Anwendung Plotter zurück. Sie ist unter anderem die Basis unseres BoSym [Eul15]. Mit Hilfe des Plotters lassen sich die Formen leicht darstellen. Man kann daher ohne großen zusätzlichen Programmieraufwand die Klassen nutzen, um damit Bilder zu erstellen.

Ausgangspunkt ist eine Klasse für einen einzelnen Punkt. Diese Klasse hat die beiden Attribute `x` und `y` für die Koordinaten, einen Konstruktor sowie eine Methode zum Bewegen des Punktes. Listing 11.1 zeigt die entsprechende Klasse.

Listing 11.1: Klasse Punkt

```
1  package geodirekt;
2
3  public class Punkt {
4      double x;
5      double y;
6
7      public Punkt(double x, double y) {
8          this.x = x;
9          this.y = y;
10     }
11
12     public void bewegeUm(double dx, double dy) {
13         x += dx;
14         y += dy;
15     }
16 }
```

Mit Hilfe von Punkten können nun Formen implementiert werden. Als erstes Beispiel betrachten wir eine Klasse `Dreieck` (Listing 11.2). Ein Dreieck besteht aus drei Punkten, die im Konstruktor übergeben werden. Um den Code übersichtlich zu halten und später leichter verwenden zu können, werden die Punkte in einem Feld gespeichert. Die Methode `bewegeUm` reicht die Informationen an alle Punkte weiter.

Listing 11.2: Klasse Dreieck

```
1  package geodirekt;
2
3  import plotter.Plotter;
4
5  public class Dreieck {
6      Punkt[] punkte;
7
8      public Dreieck(Punkt p1, Punkt p2, Punkt p3) {
9          punkte = new Punkt[3];
10         punkte[0] = p1;
11         punkte[1] = p2;
12         punkte[2] = p3;
13     }
14
```

```
15      public void zeichnen( Plotter plotter ) {
16          plotter.nextDataSet();
17          for ( Punkt p : punkte ) {
18              plotter.add( p.x, p.y );
19          }
20          plotter.add( punkte[0].x, punkte[0].y );
21      }
22
23      public void bewegeUm(double dx, double dy) {
24          for (Punkt p : punkte) {
25              p.bewegeUm(dx, dy);
26          }
27      }
28  }
```

Beim Zeichnen gehen wir davon aus, dass eine `Plotter`-Instanz vorhanden ist und die Methode eine Referenz darauf erhält. Intern verwaltet `Plotter` eine Anzahl von Datensätzen. Ein Datensatz wiederum enthält eine Liste mit Punkten. Beim Zeichnen werden die Punkte des Datensatzes miteinander verbunden. Mit `nextDataSet()` wird zunächst ein solcher Datensatz angelegt. Anschließend werden mit `add()` die Eckpunkte werden eingetragen. Um die Linie wieder zu schließen, wird zusätzlich der erste Punkt als Abschluss nochmal eingetragen.

Mit dieser Implementierung kann `Dreieck` wie eine Schablone verwendet werden. Wenn man ein `Dreieck` zeichnet, verschiebt und dann wieder zeichnet, bleibt die erste Darstellung erhalten und es werden insgesamt zwei Dreiecke gezeichnet. Das Dreieck-Objekt behält keine Informationen über die Zeichnung. Dieser lose Zusammenhang hält den Code einfach. Natürlich könnte man auch ein Konzept umsetzen, bei dem zu jeder Instanz von `Dreieck` auch genau ein gezeichnetes Dreieck gehört. Mit `bewegeUm()` würde dann das Dreieck an der alten Stelle gelöscht und neu gezeichnet werden.

Als Beispiel lassen wir in einer weiteren Klasse `Maler` ein `Dreieck` zweimal zeichnen (11.3). Zunächst wird ein `Graphic`-Objekt angelegt. Diese Klasse aus dem Plotter-Projekt stellt eine Anwendung rund um den eingebetteten Plotter bereit. Dann lassen wir uns das zugehörige `Plotter`-Objekt geben. Die beiden Methoden-Aufrufe setzen den Darstellungsbereich und schalten den automatischen Farbenwechsel bei mehreren Datensätzen aus.

Listing 11.3: main-Methode in Klasse `Maler`

```
1 public static void main(String[] args) {
2    Graphic graphic = new Graphic("Geometrie");
3    Plotter plotter = graphic.getPlotter();
4    plotter.setRange(-1., 11);
5    plotter.setAutoIncrementColor(false);
6
```

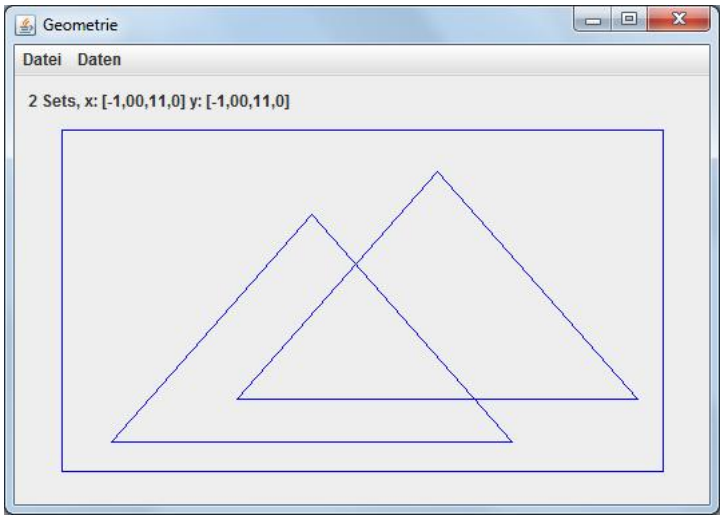


Abbildung 11.1: Zwei Dreiecke

```
7     Punkt p1 = new Punkt(0, 0);
8     Punkt p2 = new Punkt(8, 0);
9     Punkt p3 = new Punkt(4, 8);
10
11    Dreieck d1 = new Dreieck(p1, p2, p3);
12    d1.zeichnen(plotter);
13    d1.bewegeUm(2.5, 1.5);
14    d1.zeichnen(plotter );
15 }
```

Nach dem gleichen Schema wie `Dreieck` ist eine Klasse `Kreis` (Listing 11.4) geschrieben. Eine `Kreis`-Instanz wird dabei durch den Mittelpunkt sowie den Radius definiert. Um die Vergleichbarkeit mit `Dreieck` zu erhöhen, wird der Mittelpunkt als erstes und einziges Element in einem Feld mit einem Punkt abgelegt.

Listing 11.4: Klasse Kreis

```
1  package geodirekt;
2
3  import plotter.Plotter;
4
5  public class Kreis {
6      Punkt[] punkte;
7      double radius;
8
9      public Kreis(Punkt p1, double radius) {
10         punkte = new Punkt[1];
11         punkte[0] = p1;
```

```
12          this.radius = radius;
13      }
14
15      public void zeichnen( Plotter plotter ) {
16          plotter.nextDataSet();
17          for ( double t=0; t<2*Math.PI; t+=0.01 ) {
18              double x = punkte[0].x + radius * Math.cos(t);
19              double y = punkte[0].y + radius * Math.sin(t);
20              plotter.add( x, y );
21          }
22      }
23
24      public void bewegeUm(double dx, double dy) {
25          for (Punkt p : punkte) {
26              p.bewegeUm(dx, dy);
27          }
28      }
29  }
```

Man kann sich leicht vorstellen, wie weitere Formen wie Vierecke oder Ellipsen ähnlich implementiert werden können. Allerdings beinhalten bereits Dreieck und Kreis doppelten Code – deutlicher Hinweis, dass Vererbung angebracht sein könnte. Vergleicht man beide Klassen im Detail, so fällt auf

- beide Klassen beinhalten ein Feld `punkte`
- die Methoden `zeichnen()` sind deutlich unterschiedlich
- die Methoden `bewegeUm()` sind identisch

Daher bietet es sich an, die gemeinsamen Teile in eine Oberklasse auszulagern. Listing 11.5 zeigt eine entsprechende Klasse `Form`. Da wir davon ausgehen, dass jede daraus abgeleitet Klasse auch eine Methode zum Zeichnen benötigt, fügen wir eine entsprechende Definition ein. Durch das Schlüsselwort `abstract` wird die Methode abstrakt. Wir geben nur die Signatur an aber, aber keine Implementierung. In Java ist eine Klasse mit mindestens einer abstrakten Methode selbst auch abstrakt. Von abstrakten Klassen können keine Instanzen angelegt werden. Erst wenn eine abgeleitete Klasse die abstrakten Methoden implementiert wird sie konkret. In unserem Beispiel entspricht dies auch der Modellvorstellung. Eine allgemeine Form kann man nicht zeichnen, sondern nur Dreiecke, Kreise, etc.

Listing 11.5: Klasse Form

```
1  public abstract class Form
2  {
3      Punkt[] punkte;
```

```
4
5       abstract public void zeichnen( Plotter plotter );
6
7       public void bewegeUm( double dx, double dy ) {
8           for ( Punkt p : punkte ) {
9               p.bewegeUm( dx, dy );
10          }
11      }
12  }
```

Beispielhaft zeigt Listing 11.6 die neue Version von `Dreieck` als Unterklasse von `Form`. Die Klasse besteht jetzt nur noch aus dem Konstruktor und der implementierten Methode `zeichnen`. Alles andere ist bereits in der Oberklasse angelegt und wird von dort geerbt.

Listing 11.6: Klasse Dreieck abgeleitet von Form

```
1  package geodirekt;
2
3  import plotter.Plotter;
4
5  public class Dreieck extends Form {
6
7      public Dreieck(Punkt p1, Punkt p2, Punkt p3) {
8          punkte = new Punkt[3];
9          punkte[0] = p1;
10         punkte[1] = p2;
11         punkte[2] = p3;
12     }
13
14     public void zeichnen( Plotter plotter ) {
15         plotter.nextVector();
16         for ( Punkt p : punkte ) {
17             plotter.add( p.x, p.y );
18         }
19         plotter.add( punkte[0].x, punkte[0].y );
20     }
21 }
```

Ähnlich können wir eine neue Version von `Kreis` als Unterklasse schreiben. Aufgrund der Vererbung ist dann sowohl ein `Dreieck` als auch ein `Kreis` eine `Form`. Wir können daher eine Variable vom Typ `Form` definieren und ihr ein Objekt einer der konkreten Unterklassen zuweisen. Der Code-Abschnitte 11.7 zeigt diese Möglichkeit. Zwei Variablen vom Typ `Form` – der Einfachheit halber als Feld zusammengefasst – werden einmal ein `Dreieck` und einmal ein `Kreis`

zugewiesen. In der anschließenden `foreach`-Schleife wird bei beiden Objekten die Methode `zeichnen()` aufgerufen. Java prüft zur Laufzeit, um welche Objekte es sich handelt und ruft die dazu gehörige Methode auf.

Diese Möglichkeit, unterschiedliche Datentypen über eine gemeinsame Oberklasse anzusprechen, bezeichnet man als Polymorphismus (griechisch *Vielgestaltigkeit*). Wichtig ist allerdings, dass wir durch die abstrakte Methodendefinition in der Oberklasse festgelegt haben, dass jede konkrete Unterklasse diese Methode implementiert. Ohne diese Definition würde Java einen Fehler melden, da die Klasse `Form` die Methode selbst nicht kennt – selbst wenn alle abgeleitete Klassen über sie verfügen.

Listing 11.7: Formen

```
1 Form[] formen = new Form[2];
2 formen[0] = new Dreieck( p1, p2, p3 );
3 formen[1] = new Kreis( mp, .5 );
4
5 for ( Form form : formen ) {
6    form.zeichnen(plotter);
7 }
```

11.4 Mehrfachvererbung und Interfaces

11.4.1 Einleitung

Nicht immer lassen sich die Klassen in einen einzigen Zweig des Vererbungsbaums einordnen. In unserem Beispiel Hochschule könnten wir eine Klasse `HiWi` für wissenschaftliche Hilfskräfte – Studenten mit einem Arbeitsvertrag – einführen. Ein `HiWi` wäre dann ein `Student` und auch ein `Mitarbeiter`. Man spricht in diesem Fall mit mehreren direkten Oberklassen von Mehrfachvererbung. Der Nutzen der Mehrfachvererbung ist umstritten. Offensichtlich gibt es Anwendungsfälle, die durch eine hierarchische Baumstruktur nicht vollständig beschrieben werden können. Andererseits wirft die Mehrfachvererbung eine Reihe von Problemen auf. Beispielsweise gilt es dann, Namenskonflikte in den beiden (oder mehreren) Oberklassen zu vermeiden. Die Entwickler von Java haben sich für einen Mittelweg entschieden. Es gibt nur eine einfache Vererbung, aber über so genannte *Interfaces* (Schnittstellen) existiert ergänzend zu dem Vererbungsbaum eine zweite Vererbungsmöglichkeit.

Interfaces definieren ein bestimmtes Verhalten oder bestimmte Eigenschaften von Klassen. Man kann sie auch als eine Art von Protokoll interpretieren. Die Vererbungshierarchie spielt dabei eine weniger große Rolle. So stellt die Java-Klassenbibliothek ein Interface für die Eigenschaft *Vergleichbar* bereit. Diese Eigenschaft kann von ganz unterschiedliche Klassen erfüllt werden. Auch wenn meh-

rere Klassen diese Eigenschaft haben, begründet dies nicht notwendigerweise eine Verwandtschaftsbeziehung untereinander. Während die Ableitung eine *ist ein* Beziehung begründet, kann man bei implementierten Interfaces von einer *verhält sich wie* oder *hat Fähigkeiten von* Beziehung sprechen.

Ein Interface kann Konstanten und Methoden-Definitionen enthalten. Das Schlüsselwort `abstract` kann bei den Methoden-Definitionen weggelassen werden. Ab Java Version 8 können Interfaces auch eine Standard-Implementierung (Schlüsselwort `default`) von Methoden enthalten. Ein Interface ist stets abstrakt. Es können demnach keine Instanzen erzeugt werden. Eine Klasse kann ein Interface implementieren, indem sie alle abstrakten Methoden realisiert. Im Unterschied zur Ableitung bei Klassen spricht man bei Interfaces von Implementierung. Eine Klasse kann mehrere Interfaces implementieren, indem sie die Methoden konkretisiert. Mit jeder Implementierung verpflichtet sich die Klasse zu dem durch das jeweilige Interface festgelegte Verhalten.

11.4.2 Beispiel Hochschule

Interessant in Bezug auf Mehrfachvererbung sind die beiden Klassen `Tutor` und `Professor`. Die beiden Klassen befinden sich in unterschiedlichen Zweigen. Trotzdem haben sie eine Gemeinsamkeit: sie können Veranstaltungen übernehmen. Beide können in diesem Sinn als Lehrkraft angesehen werden. Dazu gehört, dass sie Vorlesungen übernehmen. Dieser Zusammenhang kann durch ein entsprechendes `Interface` realisiert werden. Dazu definieren wir ein `Interface` wie folgt:

```
package hochschule;

public interface Lehrkraft {
    public void uebernehmeVorlesung(String name, int stunden);
}
```

Die einzige Methode dient dazu, eine Vorlesung mit gegebenem Namen und einer Anzahl von Stunden zu übernehmen. Jede Klasse, die die Rolle einer Lehrkraft übernehmen will, verpflichtet sich dann, diese Methode zu implementieren. Wir erweitern unsere Klasse `Professor` entsprechend:

```
package hochschule;

public class Professor extends Person implements Lehrkraft{
    int dienstjahre;
    int SWS;

    public Professor(String name, String vorname, int dienstjahre) {
        super(name, vorname);
        this.dienstjahre = dienstjahre;
    }
```

```
    public void uebernehmeVorlesung(String vorlesung, int stunden) {
        System.out.println( "Prof. " + getName()
            + " übernimmt " + vorlesung);
        SWS += stunden;
        System.out.println( "SWS: " + SWS);
    }
}
```

Neu ist die Variable `SWS` für die Anzahl der Semesterwochenstunden. Die im Interface vorgegebene Methode wird passend realisiert. Die Klasse Tutor hat folgende Form:

```
package hochschule;

public class Tutor extends Student implements Lehrkraft {
    float vergütung = 0;

    public Tutor(String name, String vorname, int CP) {
        super(name, vorname, CP);
    }

    public void uebernehmeVorlesung(String vorlesung, int stunden) {
        System.out.println("Tutor " + getName()
            + " übernimmt " + vorlesung);
        vergütung += stunden * 10;
        System.out.println("Vergütung: " + vergütung + " Euro");
    }
}
```

In diesem Fall dient die Methode `übernehmeVorlesung` auch dazu, das Gehalt festzulegen.

Frage 11.2. Welchen Gewinn bietet die Verwendung des Interfaces?

Der Vorteil ist, dass jetzt sowohl `Professor` als auch `Tutor` als `Lehrkraft` verwendet werden können. Beide Klassen erfüllen die Bedingung „*ist eine Lehrkraft*“. Wir untersuchen diese Möglichkeit an Hand einer weiteren Klasse `Vorlesung`.

```
package hochschule;

public class Vorlesung {
    String name;
    int stunden = 2;

    public Vorlesung(String n, Lehrkraft lk) {
```

```
        name = n;
        lk.uebernehmeVorlesung(name, stunden);
    }

}
```

Der Konstruktor erwartet als Parameter eine Referenz auf eine `Lehrkraft`. Als Interface ist `Lehrkraft` abstrakt und es können keine Objekte erzeugt werden. Aber statt dessen kann ein Objekt einer Klasse, die dieses Interface implementiert, eingesetzt werden. Ein entsprechendes Beispiel ist:

```
public static void main(String args[]) {
    Professor p = new Professor( "Claudia", "Maier" );
    Tutor t = new Tutor( "Jens", "Schmidt");

    Vorlesung RN = new Vorlesung( "Rechnernetze", p);
    Vorlesung RNL1 = new Vorlesung( "Analysis 1", t);
}
```

Die Ausgabe ist:

```
Prof. Maier übernimmt Rechnernetze
SWS: 2
Tutor Schmidt übernimmt Analysis 1
Vergütung: 20.0 Euro
```

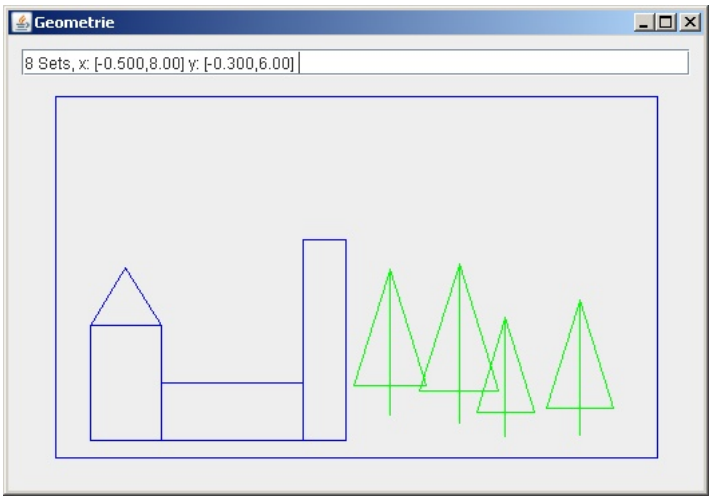
Durch die Einführung des Interfaces können beide Objekte – obwohl sie Instanzen unterschiedlicher Klassen sind – gleichermaßen als zweiter Parameter übergeben werden. Im Konstruktor wird die Methode `uebernehmeVorlesung()` ausgeführt. Automatisch wird die jeweils passende Methode ausgewählt und damit die entsprechende Aktion ausgeführt: bei der Professorin wird die Zahl der SWS erhöht und bei dem Tutor die Vergütung.

11.5 Übungen

Übung 11.5.1. GeoObjekte
Erweitern Sie unser Geometrie-Projekt aus Abschnitt 11.3:

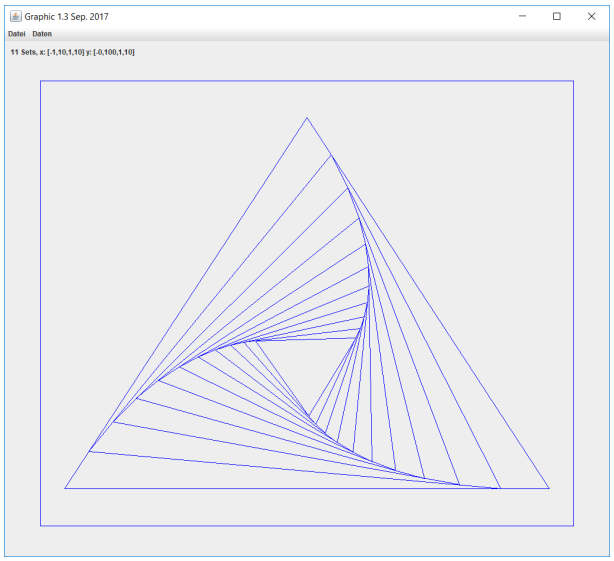
- Legen Sie Klassen für weitere Figuren an: Viereck, (achsenparallele) Ellipse, Stern, ...
- Überschreiben Sie die Methode `toString()` in den einzelnen Klassen
- Ergänzen Sie eine Methode `flaeche()`, die den jeweiligen Flächeninhalt zurück gibt

- Zeichnen Sie mit unseren Geometrie-Klassen ein kleines Bild mit einem Motiv nach Wahl. Beispiel Burg mit Zufallswald:



Übung 11.5.2. Verschachtelte Dreiecke
In ein vorgegebenes Dreieck sollen weitere Dreiecke wie im Bild gezeigt eingezeichnet werden.

- Welche neuen Methoden für die Klassen `Punkt` und `Dreieck` schlagen Sie vor?
- Wie wird damit das Zeichnen realisiert?
- Kann man diesen Ansatz auf Vierecke übertragen?



Kapitel 12

Sortierverfahren und Komplexität

Bei der Lösung der Aufgabe *Wegesuche* (Abschnitt 7.5.2) haben wir gesehen, wie eine geschickte Implementierung die Laufzeit der Suche dramatisch reduzieren kann. Häufig stehen zur Lösung eines Problems mehrere Algorithmen zur Verfügung. Dann stellt sich die Frage nach dem geeignetsten Algorithmus. Grundlage für die Entscheidung kann beispielsweise die Laufzeit oder der Speicherverbrauch sein – oder eine Kombination von beiden.

In diesem Kapitel werden wir anhand des Beispiels *Sortieren* einen solchen Vergleich durchführen. Das Sortieren oder Ordnen von Daten ist eine der am häufigsten vorkommenden Aufgabe in der Informationstechnik. Typische Anwendungen sind:

- Namenslisten alphabetisch sortieren (Einträge in Telefonbuch, Kundenlisten, o. ä.).
- Ordnen der Treffer bei einer Internetsuche nach der jeweiligen Bewertung.
- Sortieren von Angeboten bei Auktionshäusern nach unterschiedlichen Kriterien (Kategorie, Preis, Enddatum).

Dementsprechend wurden eine Vielzahl von Algorithmen entwickelt und verfeinert. Wir werden exemplarisch drei Algorithmen genauer vergleichen. Ausführliche Darstellungen findet man beispielsweise in den Büchern von Knuth und Sedgewick [Knu98, Sed02].

12.1 Aufgabenstellung

Als konkrete Anwendung betrachten wir die Konstruktion von Farey-Folgen[1]. Ausgangspunkt sind die vollständig gekürzten Brüche zwischen 0 und 1, deren

[1] John Farey, Sr., englischer Geologe, 1766 – 1826

jeweiliger Nenner den Index N_F nicht übersteigt – also Brüche $\frac{z}{n}$, die die Bedingungen

$$\begin{aligned} 1 \leq n \leq N_F \\ 0 \leq z \leq n \\ ggt(z,n) = 1 \end{aligned}$$

erfüllen. Diese Brüche sollen nach ihrer Größe sortiert werden. Drei Beispiele sind

$$\begin{aligned} F_1 &= \left(\frac{0}{1}, \frac{1}{1}\right) \\ F_2 &= \left(\frac{0}{1}, \frac{1}{2}, \frac{1}{1}\right) \\ F_5 &= \left(\frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{1}{1}\right) \end{aligned}$$

Ein direkter Ansatz zur Konstruktion einer Farey-Folge besteht aus den folgenden zwei Schritten:

1. Konstruktion aller entsprechenden Brüche.
2. Sortieren der Brüche.

Frage 12.1. Das klingt nicht besonders effizient.

In der Tat gibt es eine effizientere Methode. Wir werden später darauf zurück kommen.

12.1.1 Feld mit Brüchen

Als Basis für das Sortierverfahren benötigen wir ein Feld von Brüchen. Wir legen eine neue Klasse mit einem entsprechenden Attribut an:

```
public class Sortierer {
    Bruch[] brueche;
```

Das Problem dabei ist, dass wir vorab die benötigte Feldgröße nicht kennen. Wir könnten einen Wert wählen, der sicher groß genug ist (z. B. scheint N_F^2 ein guter Kandidat zu sein). Allerdings müssten wir uns dann zusätzlich merken, wie viele Brüche tatsächlich gefunden wurden. Die weitere Programmierung wird einfacher, wenn wir den Aufwand investieren und zunächst die Anzahl bestimmen. Eine entsprechende Lösung mit zwei Methoden zeigt Listing 12.1. Die Methode `alleBrueche()` wird zweimal durchlaufen, wobei über den Parameter `speichern` das Verhalten gesteuert wird. Beim ersten Durchgang wird nur die Anzahl der Brüche gezählt Erst beim zweiten werden tatsächlich Objekte angelegt und im Feld gespeichert.

Listing 12.1: Füllen des Feldes mit Brüchen

```
1  void fuelleFeld(int NF) {
2      brueche = new Bruch[ alleBrueche(NF, false) ];
3      alleBrueche(NF, true );
4  }
5
6  private int alleBrueche(int NF, boolean speichern) {
7      int anz = 0;
8      for (int n = 1; n <= NF; n++) {
9          for (int z = 0; z <= n; z++) {
10             if (Bruch.euclid(z, n) == 1) {
11                 if( speichern ) {
12                     brueche[anz] = new Bruch(z, n);
13                 }
14                 ++anz;
15             }
16         }
17     }
18     return anz;
19 }
```

Zur Darstellung greifen wir auf BoSym zurück. Jeder Bruch soll dabei durch ein Symbol dargestellt werden. Die Ausgabe erfolgt mit der Methode

```
void anzeigen() {
    Board board = xsend.getBoard();
    xsend.formen( "b" );
    board.setSize(900, 150);

    xsend.groesse(brueche.length, 1);

    for (int i = 0; i < brueche.length; i++) {
        xsend.farbe(i, XSendAdapter.BLUE);
        setzeGroesse(i, brueche[i]);
    }
}
```

An anderer Stelle wurde bereits `xsend` ein `XSendAdapter` zugeweisen. Dann wird zunächst die Größe des Fensters (`board`) eingestellt. Anschließend wird auf eine einzeilige Darstellung gewechselt. Schließlich wird in der Schleife für jeden Bruch die Größe des Symbols entsprechend seines Wertes gewählt. Da diese Aufgabe später mehrfach benötigt wird, ist sie in eine kleine Methode ausgelagert:

```
private void setzeGroesse(int i, Bruch bruch) {
    xsend.symbolGroesse(i, 0.1 + 0.4 * bruch.toDouble());
```

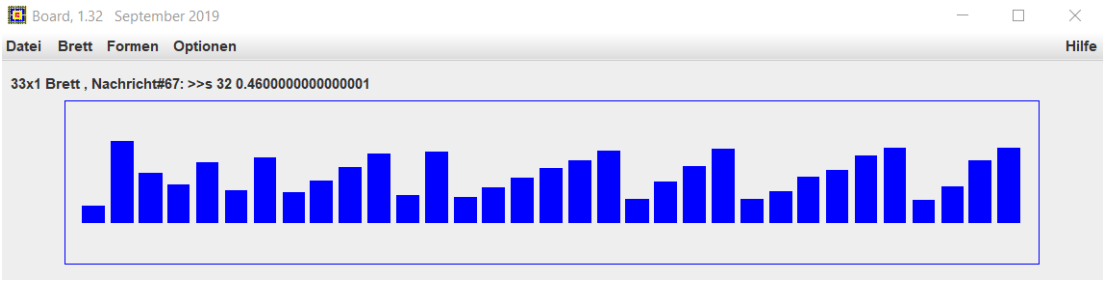


Abbildung 12.1: Unsortiertes Feld mit Brüchen, $N = 10$

```
}
```

Um auch bei kleinen Werte noch ein gut erkennbares Symbol zu erhalten, wird eine Mindestgröße von 0,1 verwendet. Bild 12.1 zeigt das Ergebnis für den Fall $N_F = 10$. Wir finden 33 verschiedene Brüche. Bedingt durch die Konstruktionsweise ergeben sich dabei Gruppen von ansteigenden Werten. Die beiden ersten Werte sind 0 und 1.

12.2 Selectionsort

Angesichts der großen praktischen Bedeutung von Sortierverfahren wundert es nicht, dass im Laufe der Zeit viele Algorithmen entwickelt und verfeinert wurden. Wir beginnen mit einem einfachen und anschaulichen Verfahren. Dabei greifen wir auf die Suche nach dem größten Wert in einem Feld zurück (Aufgabe 6.4.1). Haben wir den größten Wert gefunden, so wird er mit dem letzten Wert im Feld getauscht. Dieser Wert befindet sich dann schon an der richtigen Stelle. Der Rest – das Feld bis zum vorletzten Wert – ist nach wie vor unsortiert. In einem weiteren Durchlauf wird dann das Maximum im verbleiben Rest gesucht und an die vorletzte Stelle getauscht. Dieses Verfahren wird wiederholt, bis der unsortierte Rest auf nur noch einen Wert geschrumpft ist. Da es im wesentlich auf der Auswahl des jeweils größten Wertes beruht, wird es als *Selectionsort* bezeichnet.

12.2.1 Maximum-Suche

Zur Suche nach dem Maximum in einem Feld verwenden wir folgendes Verfahren:

- Nehme das erste Element (`feld[0]`) als größtes an. Setze einen Merker auf dieses Element.
- Prüfe für alle weiteren Elemente, ob sie größer sind als das bisher ermittelte Maximum.
- Falls ein größeres Element gefunden wird, den Merker aktualisieren.

Eine entsprechende Methode in `Sortierer` ist:

```
int findeMax() {
    int maxIndex = 0;
    for (int i = 1; i < brueche.length; i++) {
       if (brueche[i].groesser(brueche[maxIndex])) {
           maxIndex = i;
        }
    }
    return maxIndex;
}
```

Sie durchsucht das Feld `brueche` und gibt den Index des gefundenen Maximums zurück. Dabei übernimmt die Variable `maxIndex` die Funktion des Merkers. Der Vergleich erfolgt mit einer neuen Methode `groesser()` in `Bruch`. Wie der Name sagt, gibt sie `true` zurück, falls das entsprechende Bruch-Objekt größer als das übergebene Objekt ist. Diese Methode ist zusammen mit einigen weiteren in einer erweiterten Version der Klasse enthalten. Zur Visualisierung ergänzen wir die Methode um BoSym-Befehle. Das bisherige Maximum wird rot markiert und der aktuelle Vergleichswert gelb. Außerdem wird der Vergleichswert in der Statuszeile ausgegeben. Vor jedem Vergleich hält das Programm für eine vorgegebene Wartezeit an. Damit ergibt sich der etwas umfangreichere Code:

```
int findeMax() {
    int maxIndex = 0;
    xsend.farbe(0, XSendAdapter.RED);
    for (int i = 1; i < brueche.length; i++) {
        xsend.farbe(i, XSendAdapter.YELLOW);
        xsend.statusText(String.format("%d. %s = %.3f",
            i, brueche[i].toString(), brueche[i].toDouble()));
        Sleep.sleep(warteZeit);
        eingabePruefen();
        if (brueche[i].groesser(brueche[maxIndex])) {
            xsend.farbe(i, XSendAdapter.RED);
            xsend.farbe(maxIndex, XSendAdapter.BLUE);
            maxIndex = i;
        } else {
            xsend.farbe(i, XSendAdapter.BLUE);
        }
    }
    return maxIndex;
}
```

Einen Zwischenstand zeigt Bild 12.2.

Frage 12.2. Wozu wird die Methode `eingabePruefen()` benötigt?

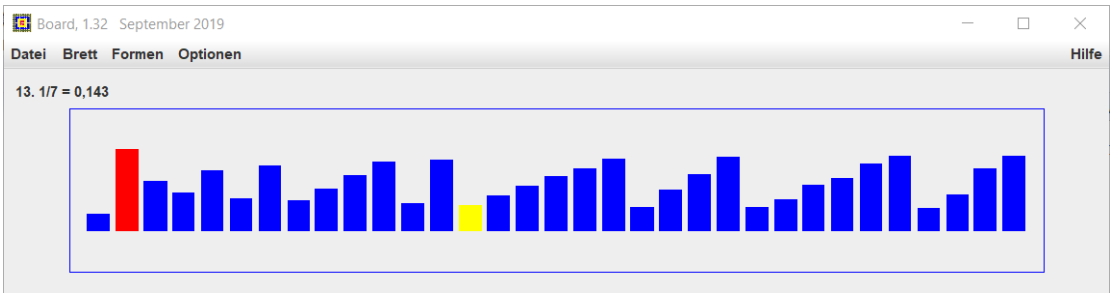


Abbildung 12.2: Zwischenstand bei der Suche nach dem Maximum

Diese Methode realisiert einen einfache Interaktion mit BoSym. Beispielsweise kann man durch Klick auf eines der Symbole die Animation anhalten und später weiter laufen lassen. Da die Details an dieser Stelle zu weit führen würden, sei auf den Anhang A verwiesen.

12.2.2 Umsetzung

Wie beschrieben führt Selectionsort die Aufgabe in mehreren Durchgängen aus. Jeder Durchgang besteht dabei aus drei Schritten:

1. Suche den größten Bruch im unsortierten Bereich.
2. Vertausche den gefundenen größten Bruch mit dem letzten Bruch.
3. Verkleinere den Suchbereich für den nächsten Durchgang.

Mit der Maximumsuche haben wir bereits den wichtigsten Baustein. Wir müssen nur in der Methode `findeMax()` einen Parameter für die Länge des Suchbereichs einführen:

```
int findeMax( int ende) {
...
    for (int i = 1; i < ende; i++) {
...
```

Dann lässt sich der Algorithmus wie folgt implementieren:

```
void selectionSort() {
    for (int ende = brueche.length; ende > 1; ende--) {
        int maxIndex = findeMax( ende );

        // Tauschen
        Bruch tmp = brueche[maxIndex];
        brueche[maxIndex] = brueche[ende - 1];
        brueche[ende - 1] = tmp;
```

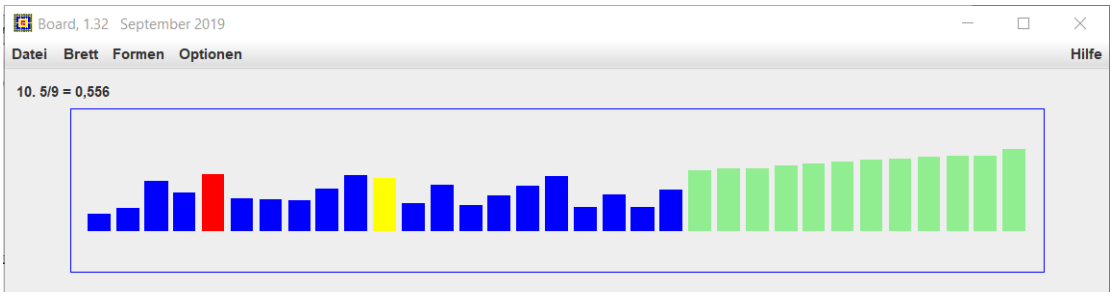


Abbildung 12.3: Zwischenstand bei Selectionsort

```
        // Symbole anpassen
        xsend.farbe(maxIndex, XSendAdapter.BLUE);
        setzeGroesse(maxIndex, brueche[maxIndex]);
        xsend.farbe(ende - 1, XSendAdapter.LIGHTGREEN);
        setzeGroesse(ende - 1, brueche[ende - 1]);
    }
    xsend.farbe(0, XSendAdapter.LIGHTGREEN);
}
```

In den vier Zeilen wird die Darstellung nach jeder Tauschoperation aktualisiert. Die Symbole werden auf die entsprechende Größe gesetzt und der nun sortierte Werte wird hellgrün gefärbt.

Im folgendem Abschnitt wird die Sortierung durchgeführt. Die Klasse Bruch enthält einen Zähler für die durchgeführten Vergleiche. Damit wird die Anzahl der benötigten Vergleiche ermittelt und ausgegeben.

```
Sortierer t = new Sortierer();
t.fuelleFeld(10);
t.anzeigen();

Bruch.resetAnzahlVergleiche();
t.selectionSort();
int N = t.anzahlBrueche();
System.out.println("N= " + N + ", "
    + Bruch.getAnzahlVergleiche() + " Vergleiche");
```

Bild 12.3 zeigt einen Zwischenstand. Rechts sind die bereits fertig sortierten Wert (hellgrün). Im linken, unsortierten Bereich läuft die nächste Maximumsuche. Nach Abschluss erfolgt die Ausgabe

```
N= 33, 528 Vergleiche
```

Die Vergleiche benötigen bei Weitem den größten Teil der Rechenzeit. Die Tausch-Operationen erfordern deutlich weniger Aufwand. Für ein gegebenes n

lässt sich die Zahl der Vergleiche auch leicht berechnen. Zunächst gilt für die Anzahl der benötigten Vergleichsoperationen in jedem Durchgang:

1. Durchgang	$n-1$ Vergleiche
2. Durchgang	$n-2$ Vergleiche
...	
k. Durchgang	$n-k$ Vergleiche
...	
letzter Durchgang	1 Vergleich

Für die Gesamtzahl $G_S(n)$ der Vergleichsoperationen folgt dann die Summe

$$G_S(n) = 1 + 2 + \ldots (n-1) = 1/2 \cdot n(n-1) = 1/2n^2 - 1/2n \tag{12.1}$$

12.2.3 Komplexitätsklassen

Um Algorithmen besser vergleichen zu können, ordnet man sie Komplexitätsklassen zu. Dazu betrachtet man den Aufwand in Abhängigkeit von einem Parameter (in unserem Fall der Feldgröße n). Diesen Parameter lässt man immer größer werden. Wir suchen dann als Abschätzung eine Funktion $f(n)$, die ab einem Wert n_0 schneller steigt. Mit einer positiven Konstanten c gilt demnach

$$G(n) < c \cdot f(n) \text{ für alle } n > n_0 \tag{12.2}$$

Für Selectionsort gilt offensichtlich

$$G_S(n) < 1/2n^2 \tag{12.3}$$

Man schreibt dann

$$G_S(n) = O(n^2) \tag{12.4}$$

mit der Bedeutung: Die Funktion $G_S(n)$ liegt in der Komplexitätsklasse $O(n^2)$. Betrachten wir als zweites Beispiel

$$G(n) = n^2 + 2n + 10 \tag{12.5}$$

Auch diese Funktion erfüllt die Bedingung

$$G(n) < c \cdot n^2 \text{ für alle } n > n_0 \tag{12.6}$$

Wählen wir $c = 3$ so folgt

$$\begin{aligned} n^2 + 2n + 10 &< 3n^2 \mid -n^2 \\ 2n + 10 &< 2n^2 \mid /2 \\ n + 5 &< n^2 \end{aligned}$$

was ab $n = 3$ erfüllt ist. Die Betrachtungsweise abstrahiert von Details und betont das grundlegende Verhalten für wachsendes n. Zwei Algorithmen in $O(n^2)$ sind im Allgemeinen unterschiedlich aufwändig. Aber sie teilen die Gemeinsamkeit, dass der Aufwand bei wachsendem n quadratisch ansteigt.

Frage 12.3. Liegt $G_S(n)$ nicht auch in $O(n^3)$?

In der Tat ist $O(n^2)$ eine Teilmenge von $O(n^3)$ und damit liegen alle Funktionen in der Teilmenge auch in der Obermenge. Eine Funktion wird aber am genauesten beschrieben, wenn wir die kleinste Menge angeben. Für mathematisch Interessierte: das Kriterium ist der Grenzwert

$$\lim_{n\to\infty} \frac{G(n)}{f(n)} \tag{12.7}$$

Existiert der Grenzwert, so ist $G(n)$ in $f(n)$. Ist der Grenzwert 0, dann wächst $f(n)$ schneller als $G(n)$ und ist eine zu große Abschätzung. In unser Fall haben wir beispielsweise einerseits für $f(n) = n^2$ gemäß

$$\lim_{n\to\infty} \frac{1}{2} \cdot \frac{n^2 - n}{n^2} = \frac{1}{2} \cdot \lim_{n\to\infty} \frac{n^2}{n^2} - \frac{n}{n^2} = \frac{1}{2} \tag{12.8}$$

den Grenzwert 1/2 und anderseits für $f(n) = n^3$

$$\lim_{n\to\infty} \frac{1}{2} \cdot \frac{n^2 - n}{n^3} = \frac{1}{2} \cdot \lim_{n\to\infty} \frac{n^2}{n^3} - \frac{n}{n^3} = 0 \tag{12.9}$$

den Wert 0. Eine Untermenge von $O(n^2)$ ist $O(n)$ – Funktionen mit linearer Laufzeit. Beispiele sind die Maximumsuche oder die Berechnung der Summe 1 bis n mittels Schleife (Kapitel 7.5). Die Summe lässt sich allerdings schneller mit der Summenformel berechnen. Dann hängt die Berechnung überhaupt nicht mehr von n ab. Der Algorithmus hat konstante Laufzeit und liegt in $O(1)$.

12.2.4 Stabilität

Durch den Tausch des jeweiligen Maximalwertes mit dem Wert am Ende ändert sich die Reihenfolge im unsortierten Bereich. Enthält die Eingabe gleiche Werte mehrfach, so bleibt dadurch deren Reihenfolge im Allgemeinen nicht erhalten. Betrachten wir ein Beispiel mit Zahlen: Sortiert werden sollen die Werte

14	72	24	52	19	82	11	29	<u>52</u>	26

wobei die Zahl 52 zweimal enthalten ist. Zur Unterscheidung ist der zweite Werte unterstrichen. Im ersten Durchgang wird der größte Wert 82 durch Tausch mit der 26 ans Ende gebracht:

14	72	24	52	19	26	11	29	<u>52</u>	82

Anschließend wird der zweitgrößte Wert 72 ermittelt und mit dem vorletzten Wert <u>52</u> getauscht

14	<u>52</u>	24	52	19	26	11	29	72	82

Dadurch kommt $\underline{52}$ vor die 52, die Reihenfolge dieser beiden Elemente hat sich umgekehrt.

Frage 12.4. Wird nicht später bei der Maximumssuche der Wert $\underline{52}$ zuerst gefunden und dann nach hinten getauscht?

Ja, in unserer Implementierung – die Maximumssuche beginnt links und nimmt bei mehreren Elementen mit dem gleichen Wert das erste davon – würde die Reihenfolge wiederhergestellt werden. Aber dann würden andererseits Elemente, die (noch oder wieder) in der ursprüngliche Reihenfolge sind, beim Einsortieren vertauscht werden. Spätestens bei mehr als zwei Dubletten wird die Reihenfolge unkontrollierbar.

Bei Zahlen oder Brüchen spielt dieses Verhalten keine Rolle. Aber nehmen wir die Teilnehmer der Übungen zu einer Vorlesung. Ausgehend von einer nach Namen alphabetisch sortierten Liste werden die Teilnehmer nach ihren Gruppen A, B, ... sortiert. Bei Selectionsort werden dann innerhalb der Gruppen die Teilnehmer nicht mehr alphabetisch sortiert sein. Solche Verfahren bezeichnet man als instabil.

12.2.5 Bewertung

Selectionsort endet für jede Eingabe mit einem korrekten Ergebnis. Bei der gleichen Eingabe führt er stets zum gleichen Ergebnis, er ist determiniert. Nach jedem Durchgang steht am Ende die Gruppe der bisher gefundenen Maximalwerte in ihrer richtigen Reihenfolge. Der Algorithmus ist daher auch gut geeignet, wenn man nur die M größten oder – entsprechend modifiziert – kleinsten Werte sucht. Beispielsweise interessiert bei einer Internet-Recherche vielleicht nur die Reihenfolge der 10 besten Treffer aus 100000. Dann führt man lediglich 10 Durchgänge des Selectionsort durch und bricht danach ab.

Selectionsort hat eine Laufzeit von $O(n^2)$. Die Anzahl der Vergleichsoperationen ist dabei unabhängig von den Eingangswerten. Weiterhin muss man in jedem Durchgang zwei Werte tauschen. Nach dem Tausch steht das Element bereits an seiner endgültigen Stelle. Jedes Element wird höchstens einmal in Richtung Zielposition getauscht.

12.3 Insertionsort

Bei dem Selection Sort sucht man stets das verbleibende größte Element im unsortierten Bereich. Eine andere Methode ist, jeweils nur ein neues Element zu nehmen und in einen bereits sortierten Bereich einzufügen. Nach dieser Methode verfahren z.B. manche Kartenspieler, wenn sie eine Karte nach der anderen aufnehmen und einsortieren. Das Verfahren beginnt mit dem letzten Wert. Dann gilt ab dem vorletzten Wert:

- Suche Position im bereits sortierten Bereich (Zielposition).
- Verschiebe alle Werte vor der Zielposition.
- Füge den aktuellen Wert an Zielposition ein

12.3.1 Umsetzung

Eine Umsetzung ist die folgende Methode (zur besseren Übersicht ohne BoSym):

```
void insertionSort() {
    for (int i = brueche.length - 2; i >= 0; i--) {
        Bruch naechster = brueche[i];

        int ziel = brueche.length - 1;
        for (int j = i + 1; j < brueche.length; j++) {
            // suche ersten größeren Wert
            if (naechster.kleiner(brueche[j])) {
                ziel = j - 1;
                break;
            }
        }
        for (int k = i; k < ziel; k++) {
            brueche[k] = brueche[k + 1];
        }
        brueche[ziel] = naechster;
    }
}
```

In der äußeren Schleife werde alle Werte beginnend mit dem vorletzten durchlaufen. Die innere Schleife beinhaltet die Suche im bereits sortierten Bereich. Benötigt wird der erste Wert, der kleiner ist als der aktuelle aus der äußeren Schleife. Dann ist der Platz vor diesem Wert die Einfügeposition. Wird kein größerer Wert gefunden, ist der aktuelle Wert der bisher größte und soll ans Ende des Feldes. Mit einer weiteren Schleife werden dann alle Werte bis einschließlich zur Zielposition um einen Stelle nach vorne geschoben. Der aktuelle Wert wurde vorsorglich in der Variable `naechster` gespeichert, da er bei diesem Vorgang überschrieben wird. Schließlich wird der aktuelle Wert in der Zielposition gespeichert. Die innere Schleife wird dann abgebrochen und der nächste Durchgang beginnt.

Frage 12.5. Ist das nicht eine etwas umständliche Implementierung?

Die Suche und das Verschieben kann man auch gut in einem Schritt zusammenfassen. Dann wird der Vergleichswert überschrieben. In der gewählten Implementierung bleibt er während der Suche zur besseren Visualisierung stehen. Bild

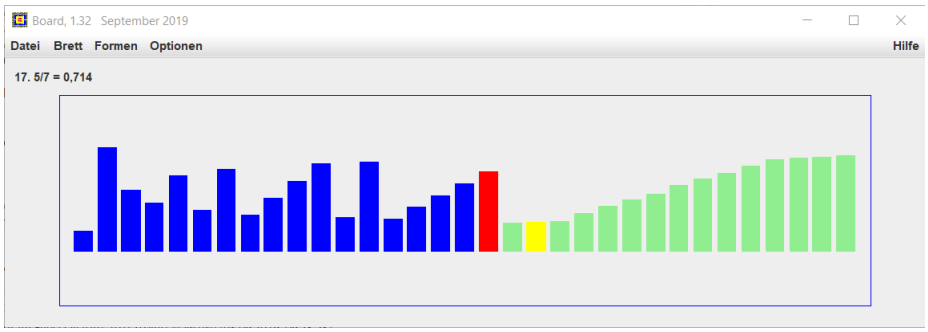


Abbildung 12.4: Zwischenstand bei Insertionsort

12.4 zeigt einen Zwischenstand. Der grüne Bereich enthält die bereits sortierten Werte. Der nächste einzufügende Wert ist rot markiert. Der nächste Vergleichswert auf der Suche nach der Einfügeposition ist schließlich gelb dargestellt. Im unsortierten Bereich stehen die Werte noch auf ihren Ausgangspositionen.

Betrachten wir zur Aufwandsabschätzung einen Durchgang. Bisher sind bereits M Werte sortiert. Um die richtige Position zu finden, muss W_M nicht mit allen bisherigen Werten verglichen werden. Es reicht vielmehr aus – beginnend mit dem ersten Wert – den ersten Wert zu suchen, der kleiner ist als W_M. Im Mittel – bei zufälliger Verteilung – benötigt man also nur $M/2$ Vergleiche. Damit ergibt sich für die mittlere Anzahl der benötigten Vergleiche

1. Durchgang	0,5
2. Durchgang	1
3. Durchgang	1,5
...	
k. Durchgang	$k/2$

bis $n/2$. Der Vergleich mit der entsprechende Form für den Selectionsort zeigt, dass der Insertionsort nur halb so viele Vergleichsoperationen benötigt:

$$G_I(n) = 1/4 \cdot n(n-1) = 1/4n^2 - 1/4n \tag{12.10}$$

und es gilt

$$G_I(n) = O(n^2) \tag{12.11}$$

Nachteil ist, dass man im bereits sortierten Bereich ständig wieder Platz schaffen muss, um das jeweils nächste Elemente einzusortieren. Dies erfordert, dass alle nachfolgenden Elemente um eine Position verschoben werden. Wenn man im Durchschnitt jedes neue Element in der Mitte einfügt, werden wiederum $M/2$ Verschiebeoperationen benötigt.

Die genaue Anzahl der benötigten Tauschoperationen hängt von der Reihenfolge der Eingangswerte ab. Im günstigsten Fall (*best case*) ist das Feld bereits sortiert. In jedem Durchlauf wird gleich beim ersten Vergleich festgestellt, dass

der Wert genau dorthin gehört wo er sich bereits befindet. Damit benötigen wir insgesamt nur n Vergleiche. Umgekehrt dauert die Suche besonders lang, wenn die Werte in umgekehrter Reihenfolge vorliegen (*worst case*). Dann muss jeder Wert an das Ende des sortierten Bereichs geschoben werden. Anstelle der im Mittel $M/2$ Vergleiche brauchen wir dann immer die volle Anzahl M. Damit verdoppelt sich der Aufwand, bleibt aber quadratisch.

12.3.2 Bewertung

Auch Insertionsort terminiert und liefert immer das korrekte Ergebnis. Da Werte direkt von ihrer Ausgangsposition in den sortierten Bereich eingefügt werden, bleibt die ursprüngliche Reihenfolge gleicher Werte erhalten. Insertionsort ist somit stabil.

Insertionsort liegt wie Selectionsort in $O(n^2)$. Im Mittel benötigt er allerdings nur halb so viele Vergleiche, aber mehr Verschiebeoperationen. Günstige Ausgangslagen führen automatisch zu einer Aufwandsreduktion. Umgekehrt ist das Risiko gering: im ungünstigsten Fall benötigt Insertionsort auch nur doppelt so viele Vergleiche wie im Mittel.

In vielen praktischen Fällen muss nicht stets ein komplettes Feld sortiert werden, sondern man kann davon ausgehen, dass bereits ein Teil des Feldes in der gewünschten Reihenfolge vorliegt. Beispielsweise kommen zu einer CD-Sammlung immer wieder neue Titel hinzu. Diese neuen Titel werden dann in die bestehende Reihenfolge einsortiert. Der Insertion Sort profitiert unmittelbar von der vorhandenen Ordnung. Wenn bereits m Werte sortiert sind, entspricht dies genau einer Zwischenstufe im Sortierverfahren. Daher kann in den ersten m Schritten sehr schnell festgestellt werden, dass das jeweils nächste Element an seiner – vorläufig – richtigen Position steht und damit keine Vertauschungen notwendig sind. Nur die $n - m$ neuen Elemente müssen dann einsortiert werden.

Zu Insertionsort gibt es diverse Verbesserungen. Ein Ansatzpunkt ist die Suche nach der Einfügeposition. Da der Suchbereich bereits sortiert ist, kann man anstelle der linearen Suche eine binäre Suche verwenden. Nach dem Prinzip „Teile und Herrsche“ wird dabei in jedem Schritt die Anzahl der Vergleichswerte halbiert, indem man b mit dem Wert in der Mitte des Suchbereichs vergleicht. Je nachdem ob b kleiner oder größer als der Wert in der Mitte ist, braucht man nur noch die linke oder rechte Hälfte des Feldes zu berücksichtigen. Dieses Verfahren setzt man fort, indem man im nächsten Schritt wieder mit dem mittleren Wert im verbleibenden Intervall vergleicht. Damit erreicht man in jedem Schritt eine Halbierung der Anzahl von Vergleichswerte. Dann benötigt man im Mittel statt $M/2$ nur noch $\log_2 M$ Vergleiche. Allerdings bleibt nach wie vor der große Aufwand des Verschiebens beim Einsortieren.

12.4 Aufwandsreduktion

Bei den bisher betrachteten Verfahren wächst der Aufwand quadratisch. Wenn wir beispielsweise die Länge auf $10n$ verzehnfachen, so benötigen wir etwa bei $(10n)^2 = 100n^2$ immerhin 100 mal so viele Vergleiche. Damit wird die Aufwand bei längeren Listen zu groß für den praktischen Einsatz.

Der Aufwand lässt sich reduzieren, indem wir die große Aufgabe $10n$ Werte zu sortieren, in kleinere Aufgaben teilen. Nehmen wir einfach Felder der Länge n. Um zehnmal ein Feld der Länge n zu sortierten, genügen $10n^2$ Vergleiche. Wir sind jetzt um einen Faktor 10 besser. Für die Gesamtlösung müssen die sortierten Teilfelder noch zusammengebaut werden. Das ist aber sicherlich weniger Aufwand als die eingesparten $90n^2$ Vergleiche. Wenn wir also nicht das große Feld als Ganzes sortieren, sondern es in Teilfelder aufspalten, diese sortieren und dann erst zusammenführen sollte der Aufwand deutlich geringer sein. Wir probieren diesen Ansatz mit einem einfachen Verfahren:

- Das Feld wird in zwei Hälften geteilt.
- Jeder Teil wird sortiert (Selectionsort oder Insertionsort).
- Die beiden Teile werden zusammen gesetzt.

Diesen Ansatz können wir im Wesentlichen mit bereits vorhanden Bausteinen realisieren. Zum Sortieren verwenden wir Selectionsort. Wir benötigen lediglich eine Variante, bei der man die Grenzen explizit angeben kann. Das Zusammenfügen erfolgt ähnlich wie das Einfügen bei Insertionsort. Allerdings kann man jetzt ausnutzen, dass die einzufügenden Werte auch sortiert sind. Daher muss die Einfügeposition hinter der letzten liegen und die Suche kann dort weitergehen. Eine entsprechende Methode ist

```
void selectionSortMiddle() {
    selectionSort(0, brueche.length / 2);
    selectionSort(brueche.length / 2, brueche.length);

    int d = 0;
    for ( int up = brueche.length / 2; up<brueche.length; up++ ) {
        // Position zum Einfügen suchen
        while( brueche[d].kleiner( brueche[up]) ){
            ++d;
        }
        // Platz schaffen
        Bruch tmp = brueche[up];
        for (int k = up; k >d; k--) {
            brueche[k] = brueche[k-1];
        }
```

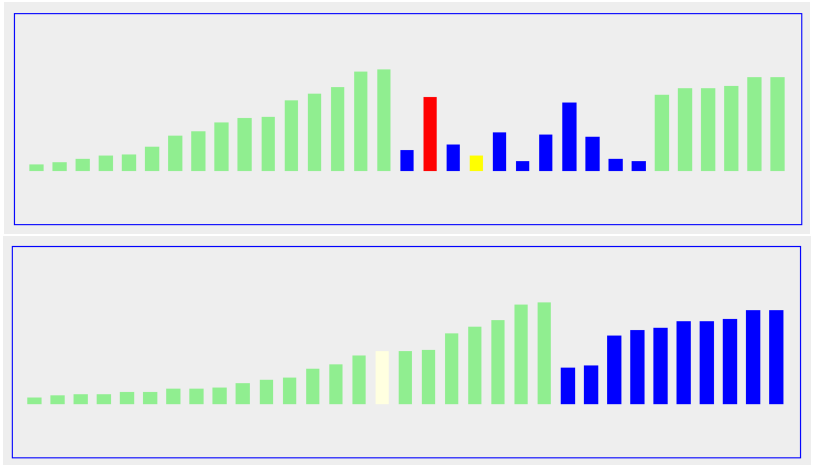


Abbildung 12.5: Sortieren mit zwei Teilfeldern

```
        // Einfügen
        brueche[d] = tmp;
    }
}
```

Bild 12.5 zeigt zwei Zwischenstände beim Ablauf. Im oberen Bild ist die erste Hälfte des Feldes bereits sortiert und Selectionsort arbeitet in der zweiten Hälfte. Im unteren Bild sind beide Hälften sortiert und werden jetzt von links beginnend zusammen gefügt.

In unserem Referenzfall $n = 33$ werden nur noch 305 Vergleiche benötigt gegenüber 528 beim Selectionsort. Dieses einfache Verfahren zeigt bereits das Potential des Ansatzes. Eine Verbesserung erreichen wir durch wiederholte Anwendung des Grundprinzips. Die beiden Hälften werden nicht direkt durch Selectionsort sortiert. Vielmehr unterteilen wir die Hälften und sortieren die kleineren Vierteln. Diese Aufteilung lässt sich im Prinzip weiter fortsetzen. Nach dieser Grundidee – teilen und zusammenfügen – arbeitet das Verfahren Mergesort. Allerdings benötigt man für ein effizientes Zusammenfügen entweder zusätzlichen Speicher oder recht trickreiche Implementierungen.

Eine mögliche Verbesserung ist, das Feld nicht in zwei Hälften sondern in zwei ineinander verzahnte Folgen unterteilen. Entsprechend zu den beiden Hälften betrachten wir die beiden Gruppen

```
brueche[0]  brueche[2]  brueche[4]  ...
```

und

```
brueche[1]  brueche[3]  brueche[5]  ...
```

Wieder werden die beiden Gruppen zunächst getrennt sortiert und dann zusammengefügt. Da in diesem Fall zunächst weiter auseinander liegende Werte sortiert

werden, kommen sie schneller in die Nähe ihrer Zielposition. Dadurch ist das Zusammenfügen weniger aufwändig. Auf dieser Grundidee beruht der Shell-Sort [She59]. Die Übertragung dieses Ansatzes auf Bubblesort führt zum Combsort [Dob80].

12.5 Quicksort

Ebenfalls nach dem Prinzip „Teile und Herrsche“, aber mit einem etwas anderen Ansatz arbeitet Quicksort (C. A. R. Hoare, 1960 [Hoa62]). Das Ziel ist, in einem Durchgang ein Element schnell an die Endposition zu bringen und gleichzeitig einen maximalen Nutzen aus den dazu notwendigen Vergleichen zu ziehen. Das Grundprinzip ist wie folgt:

1. Wähle ein beliebiges Element X aus (Pivotelement).
2. Bringe alle kleineren Werte auf die linke Seite.
3. Bringe alle größeren Werte auf die rechte Seite.
4. Setze dann X an die Stelle zwischen die beiden Bereiche.

Danach steht X selbst bereits an der richtigen Stelle. Die restlichen Werte sind nach ihrer Größe im Vergleich zum Pivotelement in zwei Teilbereiche getrennt. Diese beiden Teile können daher in weiteren Durchgängen unabhängig voneinander weiter sortiert werden.

12.5.1 Umsetzung

Kernstück des Quicksort-Algorithmus ist das Aufteilen des Feldes in die beiden Bereiche mit den kleineren und größeren Werten. Dafür verwenden wir eine eigene Methode `aufteilen()` (Listing 12.2).

Listing 12.2: Methode `aufteilen()`

```
1  private int aufteilen(int links, int rechts) {
2      int p = (links+rechts) / 2;
3      tausche(p, rechts);
4
5      int il = links;
6      int ir = rechts - 1;
7      while (il <= ir) {
8          if (brueche[il].groesser(brueche[rechts])) {
9              tausche(il, ir);
10             ir--;
11         } else
```

```
12              il++;
13      }
14      tausche(il, rechts);
15      return il;
16  }
```

Die Methode erhält als Parameter die beiden Grenzen. Sie gibt die Position zurück, an der das Pivotelement eingefügt wurde. Dies ist gleichzeitig die Grenzen zwischen den beiden noch zu sortierenden Bereichen. Zunächst wird in der Methode der mittlere Wert als Pivotelement ausgewählt. In der vorliegenden Implementierung wird dann das Pivotelement mit dem letzten Element getauscht. Zur besseren Übersicht führen wir dazu eine Methode

```
private void tausche(int i, int j) {
    Bruch tmp = brueche[i];
    brueche[i] = brueche[j];
    brueche[j] = tmp;
}
```

ein. Bild 12.6 zeigt diese beiden Schritte. Zum Tauschen verwenden wir zwei

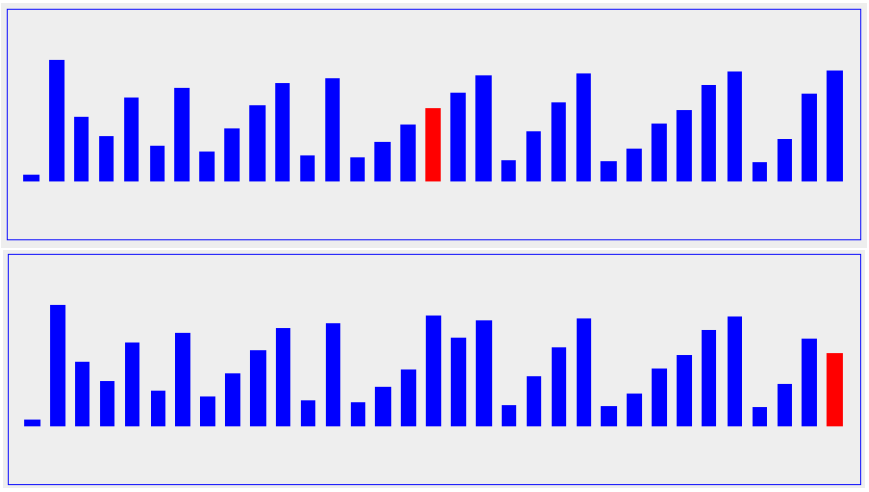


Abbildung 12.6: Quicksort: Auswahl des mittleren Wertes als Pivotelement und Tausch mit dem letzten Element

Merker:

1. `il` beginnt links und markiert den nächsten Vergleichswert.
2. `ir` beginnt rechts vor dem Pivotelement und markiert das Ziel für größere Werte.

In der Schleife wird der Werte `brueche[il]` mit dem Pivotelement verglichen. Ist er kleiner, dann befindet sich das Element bereits im richtigen Bereich. Dann wird

lediglich der Marker weiter gezählt. Ein größerer Werte wird demgegenüber nach rechts getauscht auf die mit `ir` markierte Position. Dann wird `ir` um eine Position nach links geschoben. Es kann durchaus sein, dass der getauschte Wert ebenfalls größer als das Pivotelement ist und daher nicht an der neuen Position bleiben soll. Um diesen Fall abzufangen, wird `il` nicht verändert und damit automatisch für den nächsten Vergleich nochmal verwendet. In Bild 12.7 ist das Pivotelement ganz rechts wieder rot gezeichnet. Davor stehen die bereits getauschten Elemente (gelb), die alle größer als das Pivotelement sind. Das einzelne gelb gezeichnete Element ist das aktuelle Vergleichselement `brueche[il]`.

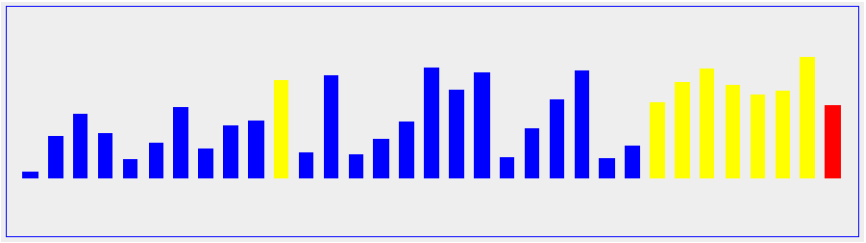


Abbildung 12.7: Quicksort: Zwischenstand im ersten Durchgang

Die Schleife endet, wenn die beiden Marker aufeinander treffen. Dann deutet `il` auf den ersten Wert größer als das Pivotelement. Daher gehört das Pivotelement genau an diese Stelle. Mit einem letzten Tausch wird es hierher gebracht. In Bild 12.8 ist der erreichte Stand dargestellt. Das ehemalige Pivotelement – jetzt grün gezeichnet – befindet sich bereits an seiner finalen Position. Die restlichen Elemente wurden ihrer Größe entsprechend auf die beiden Seiten aufgeteilt.

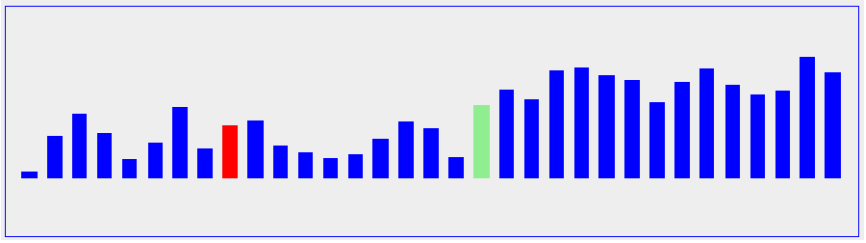


Abbildung 12.8: Quicksort: Nach dem ersten Durchgang

Im nächsten Schritt müssen die beiden unsortierten Bereiche bearbeitet werden. Das ist die gleiche Aufgabe wie zu Beginn – nur jetzt für Teilbereiche. Wir haben unser Aufgabe *Sortieren eines großen Feldes* durch das Aufteilen auf *Sortieren zweier kleinerer Felder* reduziert. Dieses Vorgehen können wir fortsetzen. Um eines der kleinen Felder zu sortieren, teilen wir es in zwei noch kleinere Felder und arbeiten damit weiter. Dieses Prinzip lässt sich durch eine rekursive Methode implementieren:

```
void quickSort(int links, int rechts) {
```

```
    if (links >= rechts) {
        return;
    }

    int i = aufteilen(links, rechts);
    quickSort(links, i - 1);
    quickSort(i + 1, rechts);
}
```

In jedem Aufruf teilen wir das Feld. Als Rückgabewert erhalten wir die Grenze. Damit rufen wir dann die Methode für jedes verbliebene Teilfeld wieder auf. Die Rekursion bricht ab, sobald die linke Grenze die rechte Grenze erreicht hat. Im Bild 12.9 ist ein entsprechender Zwischenstand gezeigt. Die grün gezeichneten Elemente haben bereits ihre Endposition erreicht.

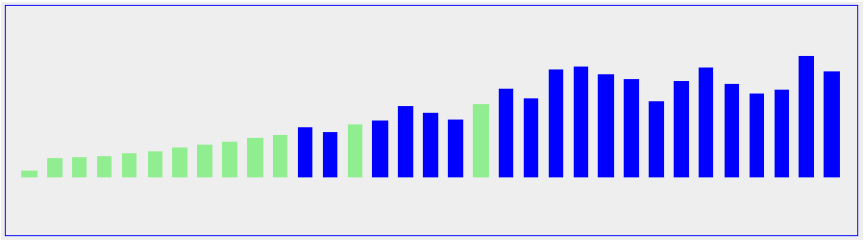


Abbildung 12.9: Quicksort: Nach mehreren Rekursionen

12.5.2 Komplexität

Im ersten Durchgang benötigen wir n Vergleiche zum Aufteilen der Elemente. Nehmen wir den idealen Fall an, dass das Pivotelement genau in der Mitte liegt und damit die beiden Teilbereiche gleich groß sind, so gilt näherungsweise die Rekursion

$$G_Q(n) = n + 2G_Q(n/2) \tag{12.12}$$

Zur weiteren Vereinfachung betrachten wir Feldlängen, die Vielfache von 2 sind. Es gibt also ein ganzzahligen m mit $n = 2^m$. Dann erhalten wir

$$G_Q(2^m) = 2^m + 2G_Q(2^{m-1}) \tag{12.13}$$

Division durch 2^m führt zu

$$\frac{G_Q(2^m)}{2^m} = \frac{2^m}{2^m} + \frac{2G_Q(2^{m-1})}{2^m} = 1 + \frac{G_Q(2^{m-1})}{2^{m-1}} \tag{12.14}$$

Gleichermaßen können wir den Ausdruck $G_Q(2^{m-1})$ weiter auflösen und erhalten dann

$$\frac{G_Q(2^m)}{2^m} = 1 + 1 + \frac{G_Q(2^{m-2})}{2^{m-2}} \tag{12.15}$$

Dies können wir insgesamt m-mal durchführen, so dass auf der rechten Seiten eine Summe vom m Einsen übrig bleibt. Damit gilt

$$\frac{G_Q(2^m)}{2^m} = m \tag{12.16}$$

beziehungsweise

$$G_Q(2^m) = 2^m \cdot m \tag{12.17}$$

Jetzt ersetzen wir den Exponenten m wieder gemäß $n = 2^m$ und $m = \log n$ und kommen zum Ergebnis

$$G_Q(n) = n \cdot \log n \tag{12.18}$$

Quicksort liegt demnach zumindest in unserem idealisiert Fall in $O(n \log n)$. Man kann zeigen, dass dies auch im Mittel gilt. Kritisch ist die Auswahl des Pivotelementes. Es sollte möglichst in der Mitte des Wertebereichs liegen. Besonders ungünstig sind die Randwerte. Bei Minimum oder Maximum als Pivotelement bleibt ein Bereich leer und der andere wird nur von n auf $n - 1$ verkürzt. In diesem Fall (worst case) wählt man in jedem Durchgang immer einen Randwert. Dann ist die Komplexität $O(n^2)$.

Die Verwendung des mittleren Elementes als Pivotelement in unserer Implementierung ist eine Vorsichtsmaßnahme. Würden wir stattdessen immer das letzte Element nehmen, so käme es bei einer schon sortierten Eingabe zum worst case. Andere Varianten von Quicksort wählen das Pivotelement zufällig oder als Median von mehreren Elementen. Die Methode `sort()` in der Klasse `Arrays` der Java-Bibliothek verwendet einen Dual-Pivot Quicksort [Yar09], bei dem in jedem Durchgang 3 Bereiche gebildet werden.

12.5.3 Bewertung

Quicksort benötigt sowohl im besten als auch im durchschnittliche Fall $O(n \log n)$. Im ungünstigsten Fall werden wie bei Selectionsort und Insertionsort $O(n^2)$ Vergleiche durchgeführt. Allerdings ist bei geschickter Wahl des Pivotelementes dieser Fall sehr unwahrscheinlich. Da $\log n$ viel langsamer wächst als n, kommt Quicksort bei wachsendem n mit deutlich weniger Vergleichen als die beiden anderen Verfahren aus. In Tabelle 12.1 sind für einige Beispiele die gezählten Vergleiche eingetragen.

Quicksort benötigt nur wenige Tauschoperationen. Allerdings ist er in der vorgestellten Form nicht stabil. Zum Abschluss sind in Bild 12.10 die drei behandelten Sortierverfahren mit typischen Zwischenständen zusammengestellt. In diesem Fall wurde jeweils ein Feld mit 120 Zufallswerten verwendet.

Tabelle 12.1: Gemessene Anzahl von Vergleichen beim Sortieren von n Elementen für Selectionsort (V_S), Insertionsort (V_I) und Quicksort (V_Q),

n	V_S	V_I	V_Q
33	528	269	146
73	2628	1279	431
129	8256	3977	961
279	38781	18764	2432
491	120295	58540	4429

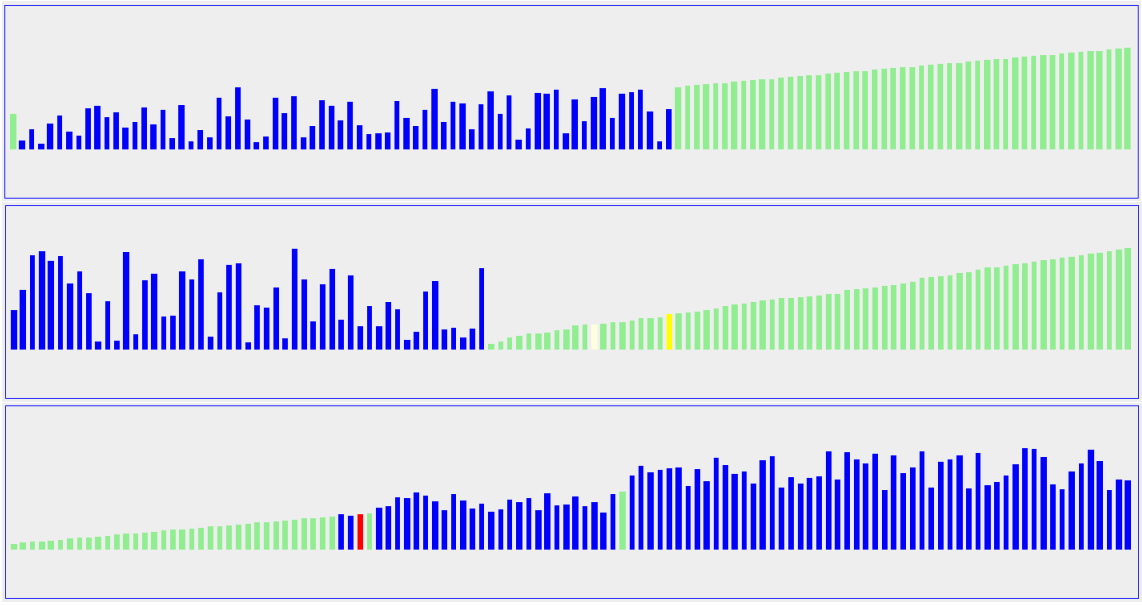


Abbildung 12.10: Vergleich der drei vorgestellten Algorithmen Selectionsort, Insertionsort und Quicksort

12.6 Übungen

Übung 12.6.1. Sortieren
Wolfgang und Sabine sollen ein Feld mit 1000 Werten sortieren. Wolfgang verwendet einfach einen Bubble-Sort. Sabine teilt das Feld in zwei Felder mit je 500 Werten. Dann lässt sie jedes der beiden Felder mit Bubble-Sort sortieren. Anschließend mischt sie die beiden sortierten Teilfelder zu einem großen Feld zusammen. Dieses Zusammenmischen benötigt nochmal 500 Vergleiche. Wie viele Vergleiche benötigen Wolfgang und Sabine jeweils?

Übung 12.6.2. Komplexität
Wie viele Züge benötigt die Lösung des Problems *Türme von Hanoi* in Abschnitt 7.5.1 bei n Scheiben?

Kapitel 13

Datenstrukturen

Bisher haben wir Felder benutzt, um mehrere gleichartige Objekte gemeinsam zu verwalten. In einem Feld stehen die Elemente in einer festen Reihenfolge und werden über ihren Index angesprochen. Am Beispiel Sortieren haben wir zwei Beschränkungen erfahren:

- Felder haben eine feste Größe.
- Das Einfügen von Elementen ist relativ aufwändig, da dann erst durch Verschieben aller folgenden Elemente Platz geschaffen werden muss.

In diesem Kapitel werden wir andere Datenstrukturen kennen lernen, die in vielen Anwendungsfällen besser geeignet sind als Felder.

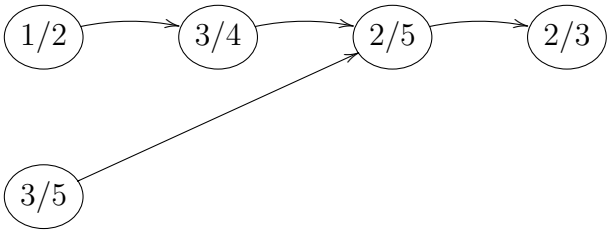
13.1 Verkettete Liste

In einem Feld befinden sich die Elemente an festen, nummerierten Plätzen so wie im anschaulichen Beispiel die Häuser entlang einer Straße. Der Vorteil ist der direkte Zugriff auf ein Element über seinen Index. Weniger günstig ist diese Anordnung beim Einfügen oder Löschen von Elementen. Dann ändern sich alle Indizes hinter der entsprechenden Stelle.

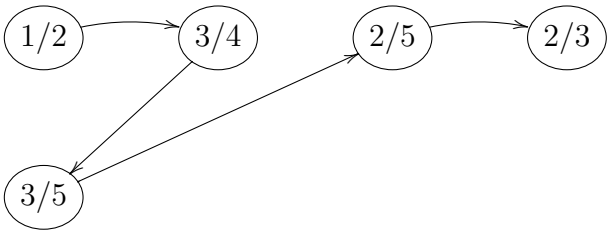
Flexibler ist eine Anordnung ähnlich einer Perlenkette. Jede Perle – außer am Rand – ist mit zwei anderen verbunden. Beim Einfügen oder Entnehmen werden alte Verbindungen gelöst und neue geknüpft. Übertragen auf unsere Brüche erhalten wir beispielsweise mit vier Werten das Bild

Jeder Bruch befindet sich in einem so genannten Knoten, der wiederum mit bis zu zwei Nachbarknoten verbunden ist. In diesem Bild sind die Verbindungen als einseitige Pfeile dargestellt. Dahinter steht die Vorstellung, dass jeder Knoten nur die Verbindung zu seinem rechten Nachbarn kennt. Wir wollen die Kette um den

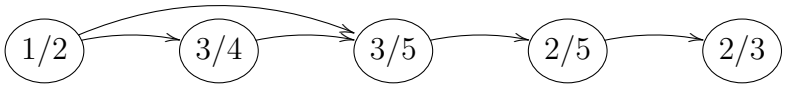
Bruch 3/5 erweitern, wobei er nach 3/4 eingefügt werden soll. Dazu müssen zwei Schritte ausgeführt werden. Zunächst wird der neue Knoten mit seinem rechten Nachbar verknüpft:



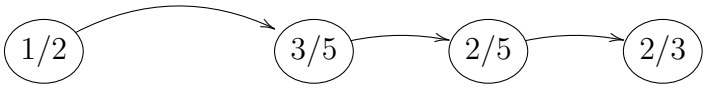
Es ist wichtig, mit dieser zweiten Verknüpfung zu beginnen. Ansonsten würde beim Auftrennen die Verbindung verloren gehen und wir hätten zwei lose Enden. So aber kann gefahrlos die vorhandene Verknüpfung gelöst werden und auf den neuen Nachbarn umgehängt werden:



Ähnlich funktioniert das Löschen von Knoten. Zunächst wird der Knoten, der gelöscht werden soll, durch eine neue Verknüpfung überbrückt:



Dann kann der Knoten entnommen werden:



13.1.1 Umsetzung in Java

Zur Veranschaulichung betrachten wir eine eigene Implementierung einer verketteten Liste. Basis ist eine Klasse für die Knoten. Jeder Knoten enthält einen Bruch und den Verweis auf seinen Nachbarknoten. Wir beginnen dementsprechend mit dem Grundgerüst

```
public class BruchKnoten {
    private Bruch bruch;
    private BruchKnoten naechster;
}
```

für die Klasse. Sie dient lediglich als gemeinsame Hülle um den Wert und die Verknüpfung. Sie benötigt daher nur noch einen Konstruktor sowie die Getter- und Setter-Methoden. Listing 13.1 zeigt die vollständige Implementierung. Das Attribut `naechster` enthält den Verweis auf den Nachbarn. Bei Knoten ohne Nachbarn ist der Wert `null`. Zur besseren Übersicht wurde noch die Methode `hatNachfolger()` eingebaut. Sie übernimmt die Prüfung, ob ein Knoten einen Nachfolger hat und gibt direkt einen Aussagewert zurück.

Listing 13.1: Klasse `BruchKnoten()`

```
public class BruchKnoten {
    private Bruch bruch;
    private BruchKnoten naechster;

    public BruchKnoten(Bruch bruch) {
        this.bruch = bruch;
    }

    @Override
    public String toString() {
        return "BruchKnoten [bruch=" + bruch +
            ", naechster=" + naechster + "]";
    }

    public Bruch getBruch() {
        return bruch;
    }

    public BruchKnoten getNaechster() {
        return naechster;
    }

    public void setNaechster(BruchKnoten naechster) {
        this.naechster = naechster;
    }

    public boolean hatNachfolger() {
        return naechster != null;
    }
}
```

Auf dieser Klasse aufbauend können wir nun die Liste angehen. In der Liste benötigen wir als Startpunkt den ersten Knoten. Von da aus können wir alle weiteren erreichen. Listing 13.2 zeigt eine Implementierung mit zwei Methoden:

1. `anhaengen(Bruch b)` hängt einen Bruch an die Liste an.

2. `anzahl()` gibt die Länge der Liste zurück.

Beide Methoden beginnen mit dem Startknoten und folgen den Verknüpfungen, bis der letzte Knoten erreicht ist. Die weitere Methode `toString()` arbeitet nach dem gleichen Prinzip und ist daher im Listing ausgelassen.

Listing 13.2: Klasse `BruchListe()` (ohne Methode `toString()`)

```
public class BruchListe {
    BruchKnoten start;

    public void anhaengen(Bruch b) {
        if (start == null) {
            start = new BruchKnoten(b);
        } else {
            BruchKnoten bk = start;
            while (bk.hatNachfolger()) {
                bk = bk.getNaechster();
            }
            bk.setNaechster(new BruchKnoten(b));
        }
    }

    public int anzahl() {
        int n = 0;
        BruchKnoten bk = start;
        while (bk != null) {
            ++n;
            bk = bk.getNaechster();
        }
        return n;
    }
}
```

Frage 13.1. Ist das nicht ziemlich mühsam?

Diese Implementierung soll gerade den grundlegenden Mechanismus zeigen. Die gesamte Information steckt in der Liste und kann bei Bedarf daraus gezogen werden. Als Alternative könnte man die Länge der Liste gut in einem eigenen Attribut speichern. Die Abfrage erfordert dann keinen Aufwand. Allerdings muss man bei Einfügen oder Löschen von Knoten auch diese Attribut aktualisieren. Der folgende Code-Abschnitt zeigt das Anlegen und Füllen der Liste:

```
BruchListe bL = new BruchListe();
System.out.println(bL.anzahl() + ": " + bL);

bL.anhaengen(new Bruch(1, 2));
System.out.println(bL.anzahl() + ": " + bL);
bL.anhaengen(new Bruch(2, 3));
System.out.println(bL.anzahl() + ": " + bL);
```

Daraus resultiert die Ausgabe

```
0: ()
1: (1/2)
2: (1/2, 2/3)
```

mit der schrittweise wachsenden Liste.

13.1.2 Beispiel Farey-Folge

Als Alternative zum Sortier-Verfahren können Farey-Folgen auch iterativ konstruiert werden. Um aus der Folge F_N die nächste Folge F_{N+1} zu gewinnen, muss man die neuen Brüche mit dem Nenner $N+1$ einfügen. Eine Möglichkeit ist, diese an die vorhandene Kette anzuhängen und dann die verlängerte Kette zu sortieren. Dabei bietet sich Insertionsort an, der von der vorhandenen Teilordnung profitiert. Allerdings gibt es eine direktere Möglichkeit. Dazu probiert man paarweise die benachbarten Brüche in F_N. Seien dies die Brüche a/b und b/c, dann bildet man den neuen Bruch

$$\frac{a+c}{b+d}$$

(Farey-Addition). Wenn der berechnete Nenner gemäß $a+c = N+1$ der Ordnung der neuen Folge entspricht, so gehört der Bruch als neues Element zwischen die beiden Brüche. Damit ist die Vorschrift für die Konstruktion von Farey-Folgen:

- Beginne mit der Folge F_1 mit den beiden Brüchen 0/1 und 1/1.
- Berechne aus allen Paaren von nebeneinander liegenden Brüchen die Farey-Summe.
- Falls der Nenner der neuen Ordnung entspricht, füge die Summe zwischen den beiden Brüchen ein.

Der erste Schritt ist einfach:

```
BruchListe bL = new BruchListe();

bL.anhaengen(new Bruch(0, 1));
bL.anhaengen(new Bruch(1, 1));
System.out.println(bL);
```

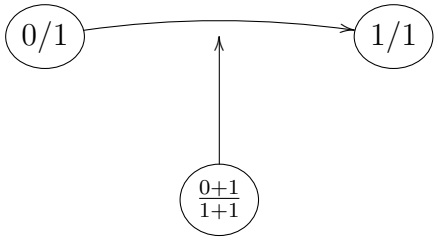
schreibt die beiden Brüche in die Liste. Die Ausgabe dazu ist:

```
(0/1, 1/1)
```

Um von F_1 zu F_2 zu gelangen „addieren“ wir das – in diesem Fall einzige – Paar und erhalten

$$\frac{0+1}{1+1} = \frac{1}{2}$$

Der Nenner stimmt mit der Ordnung überein. Also gehört der Bruch zu F_2 und wird zwischen die beiden anderen eingefügt. Das folgende Bild zeigt den Vorgang:



Dieses Vorgehen lässt sich allerdings mit der bisherigen Klasse `BruchListe` nicht leicht umsetzen. Es fehlen Möglichkeiten zum einfachen Navigieren durch die Liste und zum Einfügen von Brüchen an beliebigen Stellen. Um das Arbeiten mit der Liste zu vereinfachen, führen wir einen zusätzlichen Zeiger ein. Dieser Zeiger deutet anfangs auf den Start der Liste. Dann kann er über entsprechende Methoden zu anderen Knoten bewegt werden. Weitere Methoden ermöglichen die Abfrage und das Einfügen an der aktuellen Position. Zunächst erweitern wir die Klasse mit

```
BruchKnoten zeiger;
```

um ein weiteres Attribut. Weiterhin führen wir die folgenden Methoden ein:

`naechster()`	bewege Zeiger zum nächstem Knoten
`zeigerZurueck()`	Zeiger zurück auf Start
`vonZeiger()`	Bruch von aktueller Position
`nachZeiger()`	Bruch von Nachfolger
`einfuegen(Bruch b)`	Einfügen an der aktuellen Stelle

Beispielhaft zeigt die Methode zum Einfügen an der aktuellen Stelle

```
public void einfuegen(Bruch b) {
    // neuer Knoten
    BruchKnoten bk = new BruchKnoten(b);
    // mit Nachbarn verbinden
    bk.setNaechster( zeiger.getNaechster());
    // aktuellen Knoten mit neuem verbinden
    zeiger.setNaechster(bk);
}
```

die Verwendung des Zeigers. Mit diesen Erweiterungen lässt sich die Generierung der Farey-Folgen von 2 bis 12 folgendermaßen schreiben:

```
for (int N = 2; N <= 12; N++) {
    brueche.zeigerZurueck();
    Bruch b1;
    while ((b1 = brueche.vonZeiger()) != null) {
        Bruch b2 = brueche.nachZeiger();
        if (b2 != null) {
            Bruch t = b1.fareyAdd(b2);
            if (t.getNenner().intValue() == N) {
                brueche.einfuegen(t);
                brueche.naechster();
            }
        }
        brueche.naechster();
    }
    System.out.println(N + ": " + brueche + " " + brueche.anzahl());
}
```

13.1.3 Lösung mit Bibliotheksklassen

Anhand der eigenen Implementierung der `BruchListe` haben wir die grundlegende Funktionsweise einer Liste kennengelernt. In der Praxis ist dies aber gar nicht notwendig, da Java eine umfangreichen Klassen-Bibliothek enthält. Als Teil davon umfasst das Java Collections Framework diverse Interfaces und Klassen für unterschiedliche Datenstrukturen. In diesem Abschnitt sehen wir eine neue Version zur Generierung der Farey-Folge mittels Bibliotheksklassen. Wir verwenden wieder eine Liste. Sie wird wie folgt angelegt:

```
List<Bruch> brueche = new LinkedList<>();
```

Dabei ist `List` ein Interface, das das allgemeine Verhalten einer Liste beschreibt. Die Bibliothek enthält mehrere Klassen, die dieses Interface implementieren. Wir wählen mit `LinkedList` eine verkettete Liste. Eine Alternative wäre `ArrayList`, eine intern durch ein Feld realisierte Liste.

Frage 13.2. Warum definieren wir nicht einfach `brueche` als `LinkedList`?

Die gewählte Schreibweise ist flexibler. Wir sagen: *brueche ist von Typ List* ohne uns schon auf eine konkrete Implementierung festzulegen. Wir können dadurch später leicht die Implementierung wechseln, indem wir den Konstruktor austauschen. In den spitzen Klammern geben wir den Datentyp der Listenelemente an. Unsere Liste darf demnach nur Elemente vom Typ `Bruch` (oder gegebenenfalls

davon abgeleitete Typen) enthalten. Seit Version JDK 7 kann man beim Konstruktor vereinfacht <> (*diamond operator*) schreiben. Der Compiler fügt dann automatisch den passenden Typ ein (*Typ-Inferenz*). Nach diesen Vorbereitungen können wir die Liste mit den ersten beiden Werten füllen:

```
brueche.add(new Bruch(0, 1));
brueche.add(new Bruch(1, 1));
System.out.println("1: " + brueche);
```

In unserer Implementierung hatten wir einen Zeiger benutzt, um durch die einzelnen Knoten der Liste zu gehen. Die Bibliothek bietet eine etwas andere Möglichkeit. Man kann zu der Liste einen Iterator anlegen. Dieser Iterator bietet dann Möglichkeiten, um durch die Liste zu navigieren. Außerdem verfügt er über Methoden, zum Ersetzen, Einzufügen und Löschen von Elementen. Die folgende Schleife übernimmt die Generierung der Farey-Folgen:

```
for (int N = 2; N < 12; N++) {
    ListIterator<Bruch> iter = brueche.listIterator();
    while( iter.hasNext() ) {
        Bruch b1 = iter.next();
        if( ! iter.hasNext() ) {
            break;
        }
        Bruch b2 = iter.next();
        iter.previous();
        Bruch t = b1.fareyAdd(b2);
        if (t.getNenner().intValue() == N) {
            iter.add(t);
        }
    }
    System.out.println(N + ": " + brueche + " " + brueche.size() );
}
```

13.2 Löschen von Elementen

```
public class ListExample {
    List<Integer> liste;

    public static void main(String args[]) {
        ListExample le = new ListExample();
        le.setUp();
    }
```

```
    private void setUp() {
        liste = new ArrayList<>();
        for ( int i=0; i<10; i++ ) {
            liste.add( i );
        }
        System.out.println( liste );
    }
}
```

Direkter Versuch:

```
void loesche(int n) {
    for ( int v : liste ) {
        if( v == n ) {
            System.out.println( v );
            liste.remove( v );
        }
    }
}
```

führt zu Fehlermeldung

```
Exception in thread "main" java.util.ConcurrentModificationException
    at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:937)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:891)
    at swe2_19/daten.ListExample.loesche(ListExample.java:16)
    at swe2_19/daten.ListExample.main(ListExample.java:12)
```

Lösung 1: zu löschenden Werte zwischenspeichern

```
private void loesche1(int n) {
    Integer zuLoeschen = null;
    for ( int v : liste ) {
        if( v == n ) {
            System.out.println( v );
            zuLoeschen = v;
        }
    }
    liste.remove( zuLoeschen );
}
```

Lösung 2: Iterator

```
private void loesche2(int n) {
    Iterator<Integer> iterator = liste.iterator();
    while( iterator.hasNext() ) {
        int v = iterator.next();
        if ( v == n ) {
```

```
                iterator.remove();
        }
    }
}
```

Lösung 3: Methode `removeIf`:

```
private void loesche3(int n) {
    liste.removeIf( i -> (i == n) );
}
```

13.3 Beispiel Kartenspiel

Kinder spielen oft eine Variante von Quartett, bei der immer die höhere Karte gewinnt. Wir simulieren ein ähnliches Spiel nach folgenden Regeln:

- es gibt insgesamt 52 Karten mit den Werten 1 bis 52
- diese Karten werden gemischt und an zwei Spieler/innen verteilt
- in jedem Zug werden die beiden obersten Karten verglichen und der Spieler bzw. die Spielerin mit dem höheren Wert gewinnt beide Karten und legt sie unten wieder an
- das Spiel endet, wenn ein Spieler oder eine Spielerin keine Karten mehr hat oder eine vorgegebene Anzahl von Zügen erreicht wurde (z. B. 100).

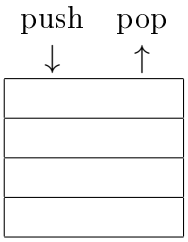
Zur Theorie: [LR10]

```
public class WarGame {
    int N = 26;

    public static void main(String[] args) {
        (new WarGame()).simulate();
    }

    private void simulate() {
        List<Integer> p1 = new ArrayList<Integer>();
        List<Integer> p2 = new ArrayList<Integer>();
        List<Integer> pa = new ArrayList<Integer>();
        for (int i = 0; i < 2 * N; i++) {
            pa.add(i);
        }
        Collections.shuffle(pa);
        p1.addAll(pa.subList(0, N));
```

```
        p2.addAll(pa.subList(N, pa.size()));

        int n = 0;
        while (n < 500 & p1.size() > 0 & p2.size() > 0) {
            ++n;
            int v1 = p1.remove(0);
            int v2 = p2.remove(0);
            if (v1 > v2) {
                p1.add(v1);
                p1.add(v2);
            } else {
                p2.add(v1);
                p2.add(v2);
            }
        }
        System.out.println("Spiel nach " + n + " Zügen zu Ende");
    }
}
```

13.4 Stack und Queue

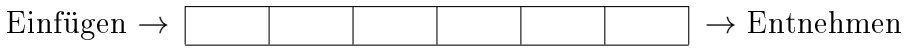
Ein Stapelspeicher (engl. *Stack*) ist ein Speicher, bei dem Werte nur am Anfang angefügt oder entnommen werden können. Anschaulich kann man sich die Werte übereinander liegend wie bei einem Stapel Spielkarten oder Mensatabletts vorstellen. Man kann jeweils eine weitere Karte bzw. ein weiteres Tablett auflegen oder vom Stapel nehmen. Diese beiden Operationen nennt man *push* (Auflegen) und *pop* (Wegnehmen). Charakteristisch für einen Stack ist, dass die Elemente in umgekehrter Reihenfolge entnommen werden (*Last In First Out*, LIFO).



Naheliegend ist die Verwendung von Stapelspeichern für das Problem Türme von Hanoi. In jedem Schritt nimmt man von einem Turm die oberste Scheibe und legt sie auf einen anderen Turm. Implementiert man die Türme jeweils als `Stack`, so lässt sich diese zentrale Operation als

```
tuerme[nach].push(tuerme[von].pop());
```

schreiben. Sollen die Werte in der Reihenfolge ihres Einfügens bearbeitet werden (*First In First Out*, FIFO), so kann man eine Warteschlange (engl. *Queue*) verwenden. Die Werte werden auf einer Seite eingefügt und nach ihrer Reihenfolge am anderen Ende entnommen:



In der Variante Prioritätswarteschlange haben die eingefügten Werte zusätzlich einen Prioritätswert. Entnommen wird dann jeweils der Wert mit der höchsten Priorität.

13.5 Assoziativspeicher

In vielen Anwendungsfällen kann man einen Datensatz als ein Paar von Schlüssel (Suchbegriff) und Wert sehen. Beispiele sind:

- Wörterbücher
- Telefonbücher
- Abürzungsverzeichnisse
- Literaturlisten

Charakteristisch ist, dass der Zugriff in der Regel über einen Schlüssel erfolgt. Die Daten sind dann so organisiert, dass die Suche in den Schlüsseln schnell erfolgen kann. Bei Telefonbüchern sind die Schlüssel die Namen. Die Einträge sind alphabetisch sortiert, so dass man einen Namen rasch findet. Die umgekehrte Suche – der Name zu einer Nummer – ist demgegenüber sehr aufwändig. Da diese Art von Anwendung häufig vorkommt, lohnt sich der Einsatz von dafür optimierten Datenstrukturen.

13.5.1 Hashtable

Die Datenstruktur Hash Table (*hash*, engl. für zerhacken, vermischen) ist für einen schnellen Zugriff über einen Schlüssel gedacht. Die Grundidee ist, aus dem Schlüssel einen Index – d. h. eine Zahl aus einem vorgegebenen Intervall – zu berechnen. Dieser Index dient dann zum schnellen Zugriff auf den Wert. Eine Voraussetzung dabei ist die Eindeutigkeit des Schlüssels. Zu einem Schlüssel darf es nur einen Wert geben. Zum Beispiel darf das Telefonbuch nicht zwei identische Namen mit unterschiedlichen Nummern enthalten. Eine Hash Table besteht aus zwei Komponenten:

- einem Feld mit linearer Adressierung

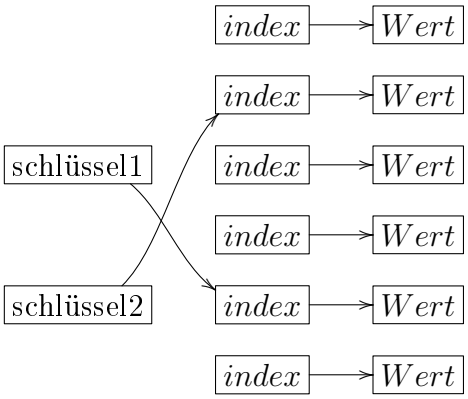


Abbildung 13.1: Aufbau einer Hashtable

- einer Hash-Funktion zur Abbildung der Schlüssel auf Indices

Dieser Aufbau ist in Bild 13.1 dargestellt. Die Hash-Funktion berechnet aus einem Objekt eine Adresse aus dem vorgegebenen Bereich (Hash-Code). Dazu wird zunächst das Objekt in eine Zahl umgewandelt. Beispielsweise kann man bei Zeichenketten die Zahlenwerte der einzelnen Zeichen addieren. Da diese Zahl im Allgemeinen außerhalb des Adressbereichs liegen kann, wird sie noch auf diesen Bereich abgebildet. Dies kann mit der Modulo-Funktion geschehen. Wichtig für die Hash-Funktion ist:

- kurze Rechenzeit
- möglichst gleichmäßige Verteilung der erzeugten Werte

Ein Vorteil ist, dass die Größe des Feldes keinen Einfluss auf die Rechendauer bei der Bestimmung des Hash-Codes hat. Damit ist die Zugriffszeit weitgehend unabhängig von der Größe der Tabelle. Mit anderen Worten: eine Hash Table bietet eine näherungsweise konstante Zugriffszeit $O(1)$. Verschiedene Objekte können allerdings den gleichen Hash-Code ergeben. Daher müssen Strategien zur Behandlung von mehreren Objekten mit identischem Hash-Code implementiert werden. Eine Möglichkeit besteht darin, an jedem Eintrag in dem Feld nicht nur ein einzelnes Objekt sondern eine Liste aller Objekte mit dem zugehörigen Hash-Code einzubauen. Die Hash-Funktion übernimmt dann die Vorauswahl der Listen.

Wichtig für die Performanz einer Hash Table ist die Größe des Feldes. Ist das Feld zu klein, so teilen sich zu viele Objekte den gleichen Hash-Code. Ist umgekehrt das Feld zu groß, so wird Speicherplatz verschwendet. Die Implementierung in Java verwendete eine Standardgröße von 11 beim Anlegen des Feldes. Erreicht die Tabelle einen bestimmten Füllstand, so wird das Feld automatisch vergrößert und die Einträge entsprechend neu angeordnet.

Mit der Methode `Object put(Object key, Object value)` werden Schlüssel-Wert-Paare in eine Tabelle eingetragen. War bereits ein Wert zu diesem Schlüssel

vorhanden, so wird der alte Wert zurück gegeben und gleichzeitig durch den neuen ersetzt. Ansonsten ist der Rückgabewert `null`. Mit `Object get(Object key)` wird der Wert zu einem Schlüssel abgefragt. Die Implementierung ist für einen schnellen Zugriff über einen Schlüssel gedacht. Allerdings stellt die Klasse auch Methoden für den langsameren Zugriff auf die Werte zur Verfügung. So erhält man mit `elements()` einen Iterator über alle Werte. Ein Beispiel für den Einsatz der Klasse zeigt die folgende Implementierung eines Telefonbuchs.

```
import java.util.Hashtable;
import java.util.Scanner;

public class HashTest {

    public static void main(String args[]) {
        Hashtable<String, String> telefonBuch = new Hashtable<>();

        telefonBuch.put("Bob", "223456789");
        telefonBuch.put("Yvonne", "123456789");
        telefonBuch.put("Pizzeria", "23456789");
        telefonBuch.put("Anna", "3456789");
        telefonBuch.put("Jörg", "123456788");
        telefonBuch.put("Dekanat", "123456780");

        Scanner in = new Scanner(System.in);
        for (;;) {
            System.out.print(">");
            String name = in.nextLine();
            if (name.length() == 0)
                break;
            String nummer = (String) telefonBuch.get(name);
            if (nummer == null) {
                System.out.println("Kein Eintrag gefunden");
            } else {
                System.out.println(name + ": " + nummer);
            }
        }
        System.out.println("ENDE");
        in.close();
    }
}
```

Ein anderes Anwendungsbeispiel nutzt eine `Hashtable`, um die Häufigkeit von Wörtern zu zählen:

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

```
Hashtable<String, Integer> ht = new Hashtable<String, Integer>();
for (;;) {
    System.out.print(">");
    String text = in.nextLine();
    if (text.length() == 0) {
        break;
    } else if (text.equals("!")) {
        System.out.println(ht);
    } else {
        String[] words = text.split("\\s+");
        for (String word : words) {
            if (ht.containsKey(word)) {
                int count = ht.get(word) + 1;
                System.out.println(word + ": " + count);
                ht.put(word, count);
            } else {
                System.out.println(word + ": neuer Eintrag");
                ht.put(word, 1);
            }
        }
    }
}
```

Die eingegebenen Zeilen werden anhand der Leerzeichen in einzelne Wörter aufgespalten. Für jedes Wort wird dann geprüft, ob es schon als Schlüssel vorhanden ist. Falls ja, wird der Wert erhöht. Ansonsten wird ein neuer Eintrag mit dem Wert 1 angelegt. Mit der speziellen Eingabe `!` wird der aktuelle Inhalt der Tabelle angezeigt. Bei einer leeren Eingabe wird die Schleife beendet.

13.5.2 Properties

Aus `Hashtable` abgeleitet ist die Klasse `Properties`. Sie ist auf die Aufnahme von Zeichenketten spezialisiert. Wie bei `Hashtable` erfolgt der Zugriff über einen Schlüssel. Mit dem Methodenpaar

```
public Object setProperty(String key, String value)
public String getProperty(String key)
```

werden Eigenschaften gesetzt und gelesen. Im folgenden Beispiel werden einige von der Klasse System gelieferte Eigenschaften dargestellt:

```
import java.util.*;
import java.io.*;

public class TestProp  {
```

```
    public static void main (String args[])
          throws Exception{
        Properties prop = System.getProperties();
        System.out.print( "Running under "
                    + prop.getProperty("os.name") );
        System.out.print( " Version "
                    + prop.getProperty("os.version") );
        System.out.print( " Architecture "
                    + prop.getProperty("os.arch") );
        System.out.println(  );
    }
}
```

Mit der Klasse `Properties` steht eine Möglichkeit zur Verwaltung von Eigenschaften für eine Anwendung zur Verfügung. Dazu dienen insbesondere die Methoden `load` und `save`, um solche Eigenschaften einzulesen oder in eine Datei zu speichern. Die Ergänzung

```
prop.save(new FileOutputStream( "props.txt"),
   "System Properties");
```

speichert die in der Klasse `TestProp` geladenen Systemeigenschaften in die Datei `props.txt`. Das zweite Argument wird als Kopf eingetragen. Die Eigenschaften werden in der Form

Schlüssel=*Wert*

zeilenweise geschrieben. In meinem Beispiel beginnt die Datei mit

```
#System Properties
#Mon Dec 02 11:53:03 CET 2002
java.runtime.name=Java(TM) 2 Runtime Environment, Standard Edition
sun.boot.library.path=C\:\\PROGRA~1\\java\\JDK13~1.1\\jre\\bin
java.vm.version=1.3.1-b24
java.vm.vendor=Sun Microsystems Inc.
java.vendor.url=http\://java.sun.com/
path.separator=;
java.vm.name=Java HotSpot(TM) Client VM
```

13.6 Bäume

Die Suche nach einem bestimmten Element ist bei Listen langwierig. Da man sich nur elementweise durch die Liste bewegen kann, bedingt eine Suche innerhalb einer Liste mit N Elementen im Mittel N/2 Vergleiche. Eine bessere Alternativen

für schnellen Zugriff bieten Bäume. Die Darstellung als Bäume begegnet uns auch im Alltagsleben. Beispiele sind die Spiele eines Turniers nach K.o. System, Hierarchien in Firmen oder der Familienstammbaum. Häufig wird die Terminologie von Familienstammbäumen übernommen. Man spricht dann entsprechend von Geschwistern, Großeltern, etc. Gebräuchlich sind auch die Ausdrücke Ast, Wurzel und Blatt.

13.6.1 Binäre Bäume zur effizienten Suche

Als Beispiel für den Einsatz von Bäumen benutzen wir einen binären Baum – ein Baum bei dem jeder Knoten maximal zwei Äste hat – zum Abspeichern einer Anzahl von Wörtern. Wir werden sehen, wie durch eine geeignete Baumstruktur eine schnelle Suche gewährleistet wird. Betrachten wir dazu folgenden Beispielsatz:

> Hab nun ach Philosophie, Juristerei und Medizin
> und leider auch Theologie durchaus studiert

Der Satz wird Wort für Wort gelesen und jedes Wort wird an die passende Stelle in einem Baum sortiert. Dabei interessiert nur, welche Wörter in dem Text auftreten. Eventuell mehrfach vorkommende Wörter werden nur beim ersten Auftreten in die Struktur aufgenommen. Das erste Wort des Textes bildet die Wurzel des Baumes:

Hab

Das zweite Wort wird nach der alphabetischen Reihenfolge an den rechten Ast gehängt:

Hab

Nun

Entsprechend gilt für das dritte Wort:

Hab

ach nun

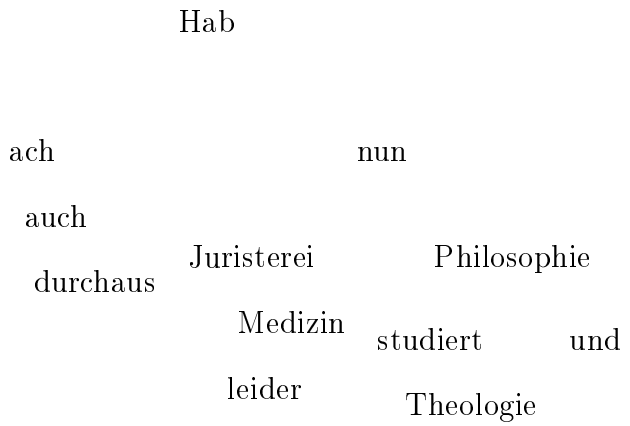
Damit ist der Wurzelknoten vollständig. Weitere Wörter werden in den angelegten Ästen angehängt. Das nächste Wort – Philosophie – folgt nach Hab und auch nach nun:

Hab

ach nun

Philosophie

Insgesamt erhält man für den Satz den Baum



Die Wörtern im Text wurden in Form eines binären Baums angeordnet. Um ein Wort zu finden, vergleicht man es zunächst mit dem Wort an der Wurzel. Falls die beiden nicht überein stimmen, folgt man je nach der Reihenfolge entweder dem linken oder rechten Ast. Erreicht man einen Endknoten ohne eine Übereinstimmung zu finden, ist das Wort nicht im Baum enthalten. Der erreichte Endknoten ist gleichzeitig die richtige Stelle um das neue Wort einzufügen. An diesem einfachen Beispiel erkennt man bereits:

- Jeder Knoten kann gleichzeitig Wurzel für einen weiteren Baum sein
- Zu jedem Knoten führt nur ein einziger Weg
- Bei einem gleichmäßig besetzten Baum wird mit jedem Vergleich die Anzahl der verbleibenden Knoten annähernd halbiert. Die Suche nach einem Wort ist daher von $O(\log_2 N)$.
- Bei beliebiger Reihenfolge der Wörter wächst der Baum ungleichmäßig
- Die Baumstruktur verbindet schnellen Zugriff auf die einzelnen Elemente mit der Möglichkeit zum dynamisches Wachstum

Kapitel 14

Grafische Benutzeroberfläche GUI

Dieses Kapitel gibt einen ersten Einblick in die Programmierung von GUIs in Java. Der Schwerpunkt liegt auf wesentlichen Grundprinzipien. Gleichzeitig bietet BoSym einen leichten Einstieg, so dass einfachere Anwendungen schnell realisiert werden können. Die aktuelle Technologie für grafische Anwendungen ist JavaFX. BoSym verwendet allerdings noch die etwas ältere, aber immer noch weit verbreitete Bibliothek Swing.

14.1 Allgemeine Architektur

Die BoSym-Anwendung beruht im wesentlichen auf drei Klassen:

- `Plotter`
- `Graphic`
- `Board`

Die Klasse `Plotter` bietet einfache Möglichkeiten zur graphischen Darstellung. Sie ist im zunächst dazu gedacht, durch Eingabe von Wertepaaren Kurven zu zeichnen. Daneben kann sie auch Texte sowie Bilder darstellen. Man kann einen Plotter auch in eigene Anwendungen einbauen. Aber zur Vereinfachung übernimmt in unserem Fall ein `Graphic`-Objekt diese Aufgabe.

Die Klasse `Graphic` ist als einfacher Rahmen um einen Plotter gedacht. Sie ist von `JFrame` aus der Swing-Bibliothek abgeleitet. Sie erstellt ein Hauptfenster und darin eingebettet ein `Plotter`-Objekt sowie eine Statuszeile. Das einzige Menü bietet die Auswahl zwischen Speichern in einer Bilddatei, Export der Daten in eine Textdatei oder direktem Ausdruck.

`Board` übernimmt schließlich die Verwaltung der Symbole. Eine `Board`-Instanz enthält eine `Graphic`-Instanz zur Darstellung. Beim Anlegen werden weitere Menüs wie die Auswahl der Farben und Formen zu dem Standard-Menü hinzugefügt.

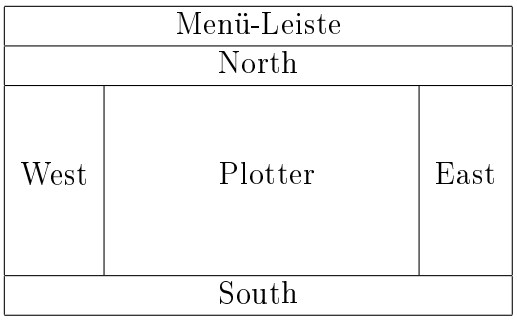


Abbildung 14.1: Aufbau der Darstellung bei Graphic

Die Klasse `Graphic` soll einen leichten Einstieg bieten, so dass ein Plotter auch ohne Kenntnis der Java-Klassen für grafische Benutzeroberflächen genutzt werden kann. Für anspruchsvollere Anwendungen ist es in der Regel sinnvoll, eine eigenständige Anwendung mit passender Benutzeroberfläche zu entwickeln. Als Mittelweg für eine schnelle Umsetzung ohne grosse Ansprüche bietet die Graphic-Klasse zwei Erweiterungsmöglichkeiten. Zum einen kann man die Menüleiste um weitere Menüs ergänzen:

```
void addExternMenu( JMenu menu )
```

Zum anderen kann man Elemente in die Bereiche rund um den zentralen Plotter einfügen. Bild 14.1 zeigt den Aufbau. Mit den Methoden

```
void addNorthComponent(Component comp)
void addSouthComponent(Component comp)
void addEastComponent(Component comp)
void addWestComponent(Component comp)
```

können Komponenten in die jeweiligen Bereiche eingefügt werden. Im Weiteren soll als Beispiel-Anwendung ein Mal-Programm entwickelt werden. Als Start dient die Klasse `BoardPainter` (Listing 14.1). Im weiteren werden entsprechende Code-Abschnitt behandelt. Eine vollständige Version steht auf GitHub[1] zur Verfügung.

Listing 14.1: Erste Version `BoardPainter`

```
1 public class BoardPainter {
2     private Board board;
3     private Graphic graphic;
4     private XSendAdapter xsend;
5
6     public static void main(String[] args) {
7         BoardPainter e = new BoardPainter();
8         e.starten();
```

[1] `https://github.com/stephaneuler/board-of-symbols/blob/master/demos/BoardPainter.java`

```
 9      }
10
11      void starten() {
12          board = new Board();
13          graphic = board.getGraphic();
14          xsend = new XSendAdapter(board);
15
16          // Standardfarbe
17          xsend.farben(XSendAdapter.LIGHTGRAY);
18
19          graphic.pack();
20          graphic.repaint();
21      }
22  }
```

14.2 Hinzufügen von Komponenten

Im ersten Schritt wird ein Schaltknopf angelegt und im unteren Bereich eingefügt:

```
JButton faerbeButton = new JButton("färben");
graphic.addSouthComponent(faerbeButton);
```

Das Klicken auf den neu eingeführten Knopf bewirkt allerdings noch nichts. Vielmehr ist es erforderlich, eine Aktion mit diesem Ereignis zu verbinden. Swing verwendet eine Ereignis-basierte Architektur (*Delegation Event Model*). Eine Aktion des Anwenders – Klicken auf einen Schalter, Eingabe von Text, Auswahl aus einem Menü, u.s.w. – löst ein entsprechendes Ereignis (Event) aus. Spezielle Objekte – Listener – warten auf Ereignisse. Sobald ein Ereignis eintritt, wird jeder dafür registrierte Listener aktiviert. Er führt dann die gewünschte Aktion aus. Der Knopf soll dazu dienen, ein Feld einzufärben. Die Koordinate soll dabei aus einem Textfeld entnommen werden. Mit

```
private JTextField indexField = new JTextField();
...
graphic.addSouthComponent(new JLabel("pos:"));
graphic.addSouthComponent(indexField);
```

wird ein solches Feld angelegt und ebenfalls in den unteren Bereich eingebaut. Zusätzlich wird mit dem `JLabel` eine Beschriftung angebracht.

Zur Behandlung des Ereignisses wird ein Objekt benötigt, das das Interface `ActionListener` implementiert. Um einen Listener für einem `JButton` zu registrieren, wird die Methode `addActionListener` verwendet. Mit dieser Methode wird das als Parameter angegebene Listener-Objekt für den Empfang von Events

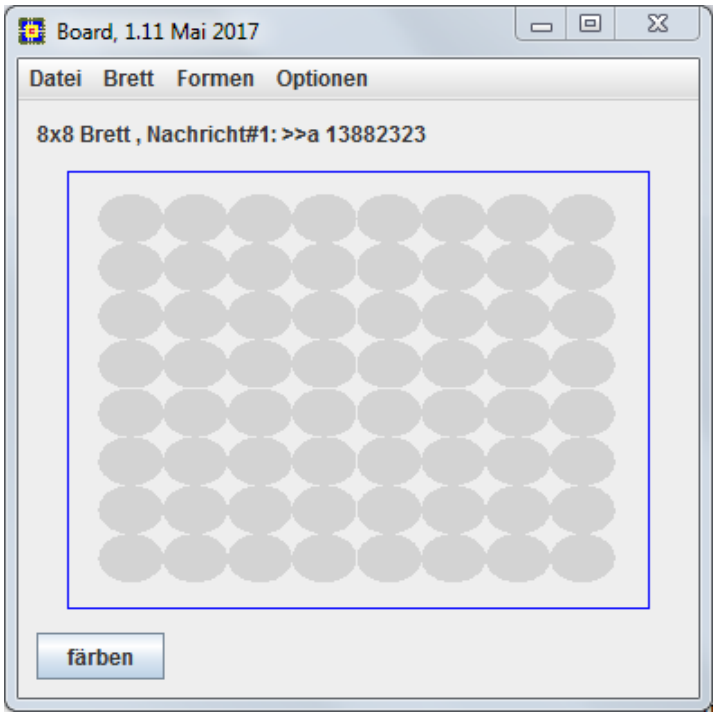


Abbildung 14.2: `BoardPainter` mit Knopf

von diesem Schalter eingetragen. Es ist im übrigen durchaus möglich, mehrere Listener für eine Komponente anzugeben. Dann werden beim Eintreten eines Ereignisses alle Listener informiert. Genauso kann ein Listener bei mehreren Komponenten angemeldet werden. Wir verwenden in unserem Beispiel als Listener eine innere anonyme Klasse. Dabei wird die Definition der Klasse und das Erzeugen ihrer einzigen Instanz kombiniert. Die Syntax dazu ist

```
faerbeButton.addActionListener(new ActionListener() {

    @Override
    public void actionPerformed(ActionEvent e) {
        String text = indexField.getText();
        int index = Integer.parseInt(text);
        xsend.farbe(index, XSendAdapter.ORANGE);
    }
});
```

Dabei wird – gekennzeichnet durch die Annotation `@Override` – die Methode `actionPerformed` überschrieben. Diese Methode wird aufgerufen, sobald der Knopf gedrückt wird. Im Parameter werden Information zum Ereignis übermittelt. In unserem Fall benötigen wir keine weiteren Informationen. Aber man beispielsweise bei einen `ActionListener` für mehrere Komponenten aus dem Para-

meter die Quelle ermitteln.

In unserer Implementierung wird der Inhalt des Eingabefeldes gelesen. Aus der dabei erhaltenen Zeichenkette wird ein `int`-Wert ermittelt. Dann wird schließlich das entsprechende Feld gefärbt. In solchen Fällen ist eine Überprüfung auf sinnvolle Eingaben hilfreich. Beispielsweise würde ein leeres Eingabefeld zu einem Fehler führen. Diesen Fall kann man wie folgt abfangen:

```
if( text.isEmpty() ) {
    JOptionPane.showMessageDialog(graphic,
            "Bitte Wert eintragen", "Leeres Feld",
            JOptionPane.ERROR_MESSAGE);
    return;
}
```

Bei einem leeren Feld wird ein Dialogfenster mit einem entsprechenden Hinweis geöffnet.

14.2.1 Radiobutton

Die Auswahl aus mehreren Optionen kann über Radiobuttons erfolgen. Wir erweitern beispielhaft unsere Anwendung um eine Farbauswahl. Zur besseren Übersicht legen wir zunächst Felder mit den Farbwerten und den Farbnamen an. In `zeichenFarbe` steht die aktuelle Farbe.

```
private int farben[] = {XSendAdapter.GREEN, XSendAdapter.YELLOW,
                        XSendAdapter.RED };
private String[] farbNamen = {"Grün", "Gelb", "Rot" };
private int zeichenFarbe = XSendAdapter.BLUE;
```

Dann werden in einer Schleife über allen Farben entsprechend viele `JRadioButton`-Elemente angelegt.

```
graphic.addEastComponent(new JLabel("Click-Farben"));
ButtonGroup group = new ButtonGroup();
for ( int f=0; f<farben.length; f++ ) {
    JRadioButton farbWaehler = new JRadioButton( farbNamen[f]);
    farbWaehler.setActionCommand(""+farben[f]);
    group.add(farbWaehler);
    farbWaehler.addActionListener(new ActionListener() {

        @Override
        public void actionPerformed(ActionEvent e) {
            zeichenFarbe = Integer.parseInt(e.getActionCommand());
            xsend.statusText("Farbe gewechselt: "
                 + Integer.toHexString(zeichenFarbe ) );
        }
```

```
    });
    graphic.addEastComponent(farbWaehler);
}
```

Angezeigt wird bei jedem `JRadioButton` der Farbname. Benötigt wird aber der Farbwert. Daher wird der Farbwert als `ActionCommand` eingetragen. In der Methode *`actionPerformed`* wird dann wiederum daraus der Farbwert ermittelt und in `zeichenFarbe` gespeichert. Zusätzlich wird bei jedem Aufruf eine entsprechende Information in die Statuszeile geschrieben.

14.2.2 Maus-Ereignisse

Die beiden Interfaces `MouseListener` und `MouseMotionListener` definieren Methoden, um auf Maus-Aktionen wie Bewegungen oder Klicks zu reagieren. Die Methoden erhalten ein `MouseEvent`-Objekt, aus dem man die Koordinaten und den Typ eines Mausklicks erhält. Zur Vereinfachung übernimmt die Klasse `Board` die Umrechnung der Koordinaten in Zeilen und Spalten-Informationen. Die entsprechende Methode ist in dem Interface `BoardClickListener` definiert. In diesem Fall soll keine weitere anonyme Klasse verwendet werden. Vielmehr soll Board selbst das Interface einbinden:

```
class BoardPainter implements BoardClickListener {
```

Dazu muss die einzige Methode in diesem Interface implementiert werden:

```
@Override
public void boardClick(BoardClickEvent info) {
    System.out.println(info);
    xsend.farbe2( info.getX(), info.getY(), zeichenFarbe );
}
```

In diesem Fall wird das angeklickte Symbol mit der zuletzt ausgewählten Farbe gefärbt. Schließlich wird mit

```
board.addClickListener(this);
```

die `Board`-Instanz als Abonnent angemeldet.

14.2.3 Menü-Einträge

Graphische Anwendungen haben oft eine Menüleiste, über die einzelne Befehle ausgeführt werden können. Eine Menüleiste besteht aus einem oder mehren Menüs mit Einträgen. Diese Hierarchie wird durch entsprechende Klassen in Java nachgebildet. Als Beispiel wird in dem Beispielprogramm ein Menü *Hilfe* mit einem Eintrag *Über* eingefügt.

```
JMenu hilfeMenu = new JMenu("Hilfe");
graphic.addExternMenu(hilfeMenu);
```

```
JMenuItem ueber = new JMenuItem("Über");
ueber.addActionListener(new ActionListener() {

        @Override
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(graphic,
                    "Testanwendung Version 0.0", "Über",
                    JOptionPane.INFORMATION_MESSAGE);
        }
    });
hilfeMenu.add(ueber);
```

14.3 Simulation Zufallsbewegung

Als zweites Beispiel realisieren wir einen Simulator für eine zufällige Bewegung (auch als *Random Walk* oder Irrfahrt bezeichnet). Bild 14.3 zeigt die Oberfläche. Diesmal soll die Bearbeitung der Ereignisse in einer gemeinsamen Methode erfolgen. Diese Methode soll in unserer Hauptklasse – wir nennen sie `Walker` – sein. Dazu muss die Klasse das entsprechende Interface `ActionListener` implementieren. Wir schreiben bei der Definition der Klasse

```
public class Walker implements ActionListener
```

und fügen dann die Methode

```
@Override
public void actionPerformed(ActionEvent e) {
}
```

ein. Damit können wir unsere `Walker`-Instanz z. B. bei einem JButton als Empfänger anmelden:

```
JButton startKnopf = new JButton("Start");
startKnopf.addActionListener(this);
```

Um passend zu reagieren, muss in der Methode nun das auslösende Ereignis ermittelt werden. Informationen dazu stehen als Zeichenkette (*action command*) im übergebenen `ActionEvent`-Objekt. In

```
public void actionPerformed(ActionEvent e) {
    String cmd = e.getActionCommand();
    System.out.println(cmd);
    if (cmd.equals("Start")) {
...
```

wird das Kommando geholt und zur Information ausgegebenen. Dann folgt die Auswahl der Aktionen. Bei einem `JButton` ist standardmäßig der angezeigte Text auch der Befehl.

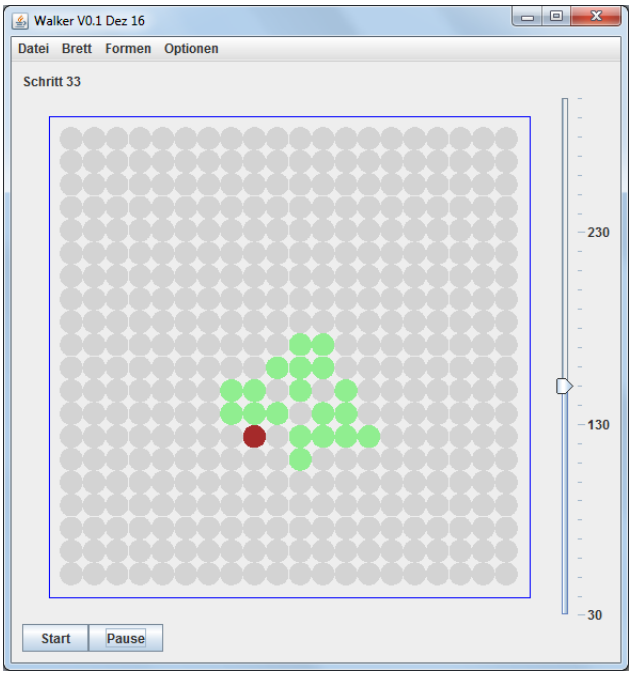


Abbildung 14.3: Simulator für zufällige Bewegung

Frage 14.1. Ist die String-Konstante nicht eine unschöne Code-Wiederholung?

Besser wäre in der Tat eine Konstante für den Befehl. Dann müssten eventuelle Änderungen nur noch an einer Stelle vorgenommen werden. Auch eine gegebenenfalls erforderliche Internationalisierung wird einfacher, wenn solche Zeichenketten gemeinsam zentral an einer Stelle definiert werden. In der Anwendung soll sich ein Punkt zufällig bewegen. Mit `x` und `y` für Spalte und Zeile sowie einer Methode `bewege(int n)` für die zufällige Bewegung können wir die Simulation als Endlosschleife schreiben:

```
if (cmd.equals("Start")) {
    for (;;) {
        xsend.farbe2(x, y, besucht);
        x = bewege(x);
        y = bewege(y);
        xsend.farbe2(x, y, farbe);
        System.out.println( x + " " + y);
        Sleep.sleep(sleepTime);
    }
```

Allerdings funktioniert dies nicht richtig. Zwar werden die Koordinaten in der Konsole ausgegeben, aber die grafische Darstellung wird nicht aktualisiert. Schlimmer noch – die Anwendung reagiert überhaupt nicht mehr. Das Problem resultiert aus der Architektur von Swing. In Swing gibt es einen Teilprozess (*Event Dispatch*

Thread), der auf GUI-Ereignisse wartet und dann die entsprechenden Aktionen ausführt. Dabei werden die Aktionen nacheinander ausgeführt. Erst wenn eine Aktion beendet ist, kann die nächste bearbeitet werden. Bei Aktionen mit kurzer Bearbeitungszeit ist dies relativ unkritisch. In unserem Fall haben wir aber eine im Prinzip unendlich lange Bearbeitungszeit. Daher ist der Thread blockiert und reagiert nicht mehr auf weitere GUI-Ereignisse. Um diese Blockade zu verhindern, lagern wir die Simulation in einen anderen Thread aus.

```
new Thread() {
    public void run() {
        for (int n = 1; ; ++n) {
            xsend.farbe2(x, y, besucht);
...
        }
    }
}.start();
```

Frage 14.2. Was passiert, wenn man mehrfach den Start-Knopf betätigt.

Mit jedem Klick wird ein weiterer Thread gestartet. In unserem Fall werden mehrere Threads gleichzeitig den Punkt bewegen. Diese eher unerwünschte Verhalten lässt sich leicht verhindern. Dazu wird beim ersten Klick mit

```
if (cmd.equals("Start")) {
    startKnopf.setEnabled(false);
```

der Start-Knopf deaktiviert. In der Anwendung ist ein zweiter JButton eingebaut, um die Simulation anzuhalten. Dazu wird in einer Variablen `pause` der aktuelle Status gespeichert. Mit einem Klick auf den Pause-Knopf wechselt dann der Status:

```
private boolean pause = false;
...
} else if (cmd.equals("Pause")) {
    pause = !pause;
}
```

In der Simulation wird dann mit

```
for (int n = 1; ; ++n) {
    if (!pause) {
        ...
```

auf den Pause-Status geprüft.

14.3.1 ActionListener

In den Beispielen hatten wir anfangs Instanzen von anonymen Klassen als ActionListener angegeben. Später hatten wir eine Klasse verwendet, die das Interface implementiert. Im ersten Fall resultierte Code in der Art

```
loescheButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent event) {
        xsend.farben(startFarbe);
    }
});
```

Ein großer Teil des Codes ist dabei vorgegeben und wird auch bei weiteren ActionListener wiederholt werden. In den meisten Fällen wird sich lediglich der Rumpf der Methode `actionPerformed` ändern. Der Rest ist bleibt gleich. Solcher Code, der mehr oder weniger identisch mehrfach vorkommt, wird als Boilerplate-Code (oder einfach nur Boilerplate) bezeichnet. Er kann zwar häufig schnell und einfach in der Entwicklungsumgebung generiert werden. Trotzdem bedeutet er einen gewissen Aufwand und die Lesbarkeit des Codes leidet. Im vorliegenden Fall kann der Code durch einen so genannten Lambda-Ausdruck vereinfacht werden:

```
loescheButton.addActionListener(event -> xsend.farben(startFarbe));
```

Der Compiler kann aus der Kenntnis der Methoden und Klassen den weiteren Code ergänzen: `addActionListener` erwartet als Argument eine `ActionListener`-Instanz. Ein `ActionListener` wiederum benötigt genau eine einzige Methode `actionPerformed`, die einen einzigen Parameter hat. Im vorliegenden Code bezeichnet `event` diesen Parameter und nach dem Pfeil folgt der auszuführende Code (der in unserem Fall den Parameter aber gar nicht verwendet). Längere Code-Blöcke werden dabei in Klammern angegeben:

```
zufallButton.addActionListener(event -> {
    for (int i = 0; i < groesse * groesse; i++) {
        xsend.farbe(i, (int) (Math.random() * 256 * 256 * 256));
    }
});
```

Kapitel 15

Spiele mit Zuständen

In Abschnitt 6.3 hatten wir bereits die Darstellung eines Dame-Spiel behandelt. Die Umsetzung des Code-Snippets führt zur Klasse 15.1.

Wir registrieren zunächst unsere Klasse als Listener für Maus-Klicks:

```
public Dame19() {
    send.groesse(brett.length, brett.length);
    send.getBoard().addClickListener(this);
}
```

Dazu müssen wir angeben, dass die Klasse das Interface implementiert

```
public class Dame19 implements BoardClickListener {
```

und die notwendige Methode implementieren:

```
@Override
public void boardClick(BoardClickEvent info) {
    System.out.println( info );
}
```

Nun wollen wir im nächsten Schritt eine rudimentäre Spiellogik einbauen: zwei SpielerInnen sollen abwechselnd ziehen können. Zug bedeutet dabei: man wählt einen eigenen Stein aus und setzt ihn auf ein freies Feld. Die Spielregeln von Dame werden dabei noch nicht berücksichtigt. Trotzdem suchen wir auch jetzt schon eine systematische und übersichtliche Lösung, die später leicht zu erweitern ist. Betrachten wir dazu einen Zeitpunkt während des Spiels. Dann stellen sich die Fragen

- in welcher Spielsituation befinden wir uns?
- welche Aktionen sind möglich?
- wohin (zu welchen anderen Spielsituationen) führen die Aktionen?

Beginnen wir mit der Anfangssituation:

Listing 15.1: Dame-Spiel

```
import jserver.XSendAdapter;

public class Dame {
    int[][] brett = {
           { 1, 0, 1, 0, 0, 0, 2, 0 }, { 0, 1, 0, 0, 0, 2, 0, 2 },
           { 1, 0, 1, 0, 0, 0, 2, 0 }, { 0, 1, 0, 0, 0, 2, 0, 2 },
           { 1, 0, 1, 0, 0, 0, 2, 0 }, { 0, 1, 0, 0, 0, 2, 0, 2 },
           { 1, 0, 1, 0, 0, 0, 2, 0 }, { 0, 1, 0, 0, 0, 2, 0, 2 } };
    XSendAdapter send = new XSendAdapter();

    public static void main(String[] args) {
        new Dame();
    }

    public Dame() {
        setUp();
    }

    private void setUp() {
        send.groesse(brett.length, brett.length);
        send.formen("none");

        for (int x = 0; x < brett.length; x++) {
            for (int y = 0; y < brett.length; y++) {
                if (brett[x][y] == 1) {
                    zeichneStein(x, y, XSendAdapter.BLUE);
                } else if (brett[x][y] == 2) {
                    zeichneStein(x, y, XSendAdapter.YELLOW);
                }
                if ((x + y) % 2 == 0) {
                    send.hintergrund2(x, y, XSendAdapter.LIGHTGRAY);
                } else {
                    send.hintergrund2(x, y, XSendAdapter.WHITE);
                }
            }
        }
    }

    private void zeichneStein(int x, int y, int farbe) {
        send.form2(x, y, "c");
        send.symbolGroesse2(x, y, 0.4);
        send.farbe2(x, y, farbe);
    }
}
```

Tabelle 15.1: Anfangszustand

Nr	Zustand	Klick	Folge
10	S1 am Zug	eigenen Stein	11
		anders	10

Tabelle 15.2: Zustände und Übergänge im Dame-Spiel

Nr	Zustand	Klick	Folge
10	S1 am Zug	eigenen Stein	11
		anders	10
11	S1 Zug ausführen	freies Feld	20
		anders	11
20	S2 am Zug	eigenen Stein	21
		anders	20
21	S2 Zug ausführen	freies Feld	10
		anders	21

- SpielerIn 1 ist am Zug
- SpielerIn 1 kann einen eigenen Stein zum ziehen auswählen
- dies führt zu *Zielfeld auswählen*

Übersichtlich kann man dies als Tabelle 15.1 schreiben. Wir verwenden dabei die allgemeinere Bezeichnung Zustand für eine Spielsituation und haben eine Nummerierung begonnen. Die Nummerierung der Zustände ist frei wählbar. Die 10 wurde gewählt um SpielerIn 1 (S1) und Anfang (0) zu symbolisieren. Die Tabelle besagt, dass man in Zustand 10 durch Anklicken eines eigenen Steines in einen noch zu beschreibenden Folgezustand 11 gelangt. Alle anderen Klicks werden ignoriert und man verbleibt im Zustand 10.

Im folgenden Zustand 11 soll ein freies Zielfeld ausgewählt werden. Damit ist der Zug beendet und SpielerIn 2 am Zug. Dafür sehen wir einen weiteren Zustand 20 vor. Mit noch einem Zustand, in dem SpielerIn 2 den Zug beendet, erhalten wir die Tabelle 15.2. Diese Zustandsübergangstabelle beschreibt alle möglichen Zustände und Übergänge zwischen den Zuständen. Alternativ kann man diese Information auch grafisch darstellen. Bild 15.1 zeigt eine solche Darstellung (Zustandsdiagramm). Die Zustände werden durch Kreise symbolisiert und Pfeile markieren die möglichen Übergänge.

Frage 15.1. Nach dieser Tabelle muss man einen ausgewählten Stein auch ziehen. Wie könnte man dies ändern?

Unser Modell hält sich an die *berührt – geführt* Regel aus dem Turnierschach. Wir könnten es aber leicht anpassen, indem wir im Zustand 11 (und entsprechend

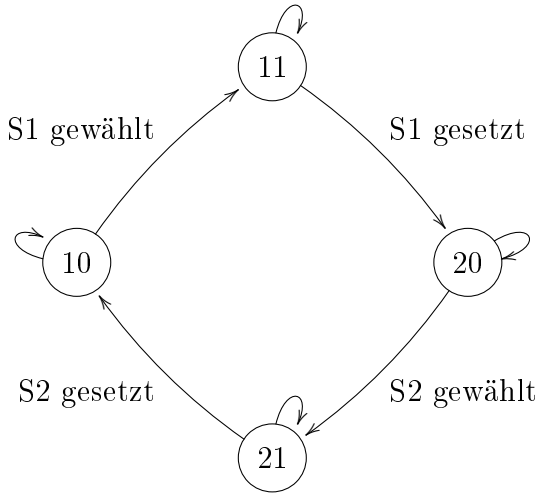


Abbildung 15.1: Zustandsdiagramm für Dame-Spiel

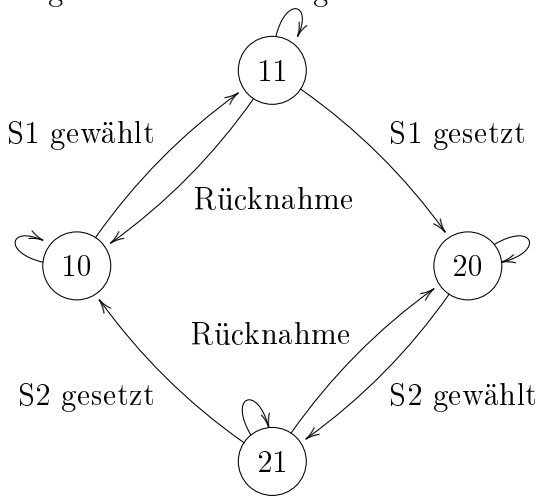


Abbildung 15.2: Zustandsdiagramm mit Rücknahme des berührten Steins

21) einen weiteren Fall *ausgewählte Figur nochmals anklicken* unterscheiden. Dies würde direkt wieder zurück zu Zustand 10 führen. Solche Fragen lassen sich leicht anhand der Übergangstabelle oder des Diagramms klären. Bild 15.2 zeigt ein entsprechend erweitertes Zustandsdiagramm.

Frage 15.2. Und wie kommen wir jetzt zum Java-Code?

Die Umsetzung von Zustandsdiagramm oder Übergangstabelle in Programmcode ist recht einfach. Zumindest für den Aufbau der Struktur muss man dazu nicht einmal das Problem verstanden haben. Daher lässt sich die Umsetzung gut automatisieren und es gibt diverse Programme, um aus einem Zustandsmodell Code zu generieren. Ausgangspunkt ist eine Variable für den aktuellen Zustand. Setzen wir für unseren Fall mit

```
int zustand = 10;
```

den Startzustand. Dann warten wir beispielsweise in einer Schleife auf Aktionen. Sobald eine Aktion auftritt, gehen wir zum aktuellen Zustand, werten die Aktion aus und legen den Folgezustand fest. Für die Auswahl des Zustandes bietet sich eine Konstruktion mit if und else-if oder ein switch-Element an. Beides funktioniert gut, wobei das switch-Element etwas übersichtlicher ist. Für jeden Zustand steht ein case-Block. Wir erhalten dann folgende allgemeine Form

```
for (;;) {
    // auf Aktion warten
    switch( zustand ) {
    case 10:
        Aktion auswerten
        Folgezustand setzen
        break;
    case 11:
        Aktion auswerten
        Folgezustand setzen
        break;
        ...
    }
}
```

Auf diese Art und Weise erhalten wir ein Grundgerüst, in das wir nur noch die Auswertung der Aktionen integrieren müssen. den vollständigen Code zeigt Listing 15.2. Die Details:

- Die Koordinaten werden aus dem Event gelesen und in `x` und `y` geschrieben
- Nach der Auswahl eines Steins werden dessen Koordinaten in `xalt` und `yalt` gepeichert
- Ein ausgewählter Stein wird vergrößert (Wert in `radG`)
- Die Ausführung eines Zugs ist zur besseren Übersicht in eine eigene Funktion ausgelagert (15.3)
- Mit der Methode `statusText()` werden Informationen zum Spielverlauf in der Statuszeile von BoSym (oberhalb des Spielbretts) angezeigt

Listing 15.2: Umsetzung der Übergangstabelle

```
public void boardClick(BoardClickEvent info) {
    int x = info.getX();
    int y = info.getY();
    switch (zustand) {
    case 10:
        if (brett[x][y] == 1) {
            send.symbolGroesse2(x, y, radG);
            xalt = x; yalt = y;
            zustand = 11;
        }
        break;
    case 11:
        if (x == xalt && y == yalt) {
            send.symbolGroesse2(x, y, rad);
            zustand = 10;
        } else if (brett[x][y] == 0) {
            zug(xalt, yalt, x, y, XSendAdapter.BLUE);
            send.statusText("SpielerIn 2 am Zug");
            zustand = 20;
        }
        break;
    case 20:
        if (brett[x][y] == 2) {
            send.symbolGroesse2(x, y, radG);
            xalt = x; yalt = y;
            zustand = 21;
        }
        break;
    case 21:
        if (x == xalt && y == yalt) {
            send.symbolGroesse2(x, y, rad);
            zustand = 20;
        } else if (brett[x][y] == 0) {
            zug(xalt, yalt, x, y, XSendAdapter.YELLOW);
            send.statusText("SpielerIn 1 am Zug");
            zustand = 10;
        }
    }
}
```

Listing 15.3: Funktion zug

```
private void zug(int xalt, int yalt, int ix, int iy, int farbe) {
    brett[ix][iy] = brett[xalt][yalt];
    brett[xalt][yalt] = 0;
    send.form2(xalt, yalt, "none");
    send.symbolGroesse2(xalt, yalt, rad);
    send.farbe2(ix, iy, farbe);
    send.form2(ix, iy, "c");
    send.symbolGroesse2(ix, iy, rad);
}
```

Kapitel 16

Exceptions

16.1 Einleitung

Bei der Ausführung einer Anwendung kann es zu Fehlern kommen. Neben Fehlern im Programm können auch äußere Umstände zu einem Fehlverhalten oder Programmabsturz führen. Häufige Ursachen sind:

- Fehlbedienungen (z. B. falsche Eingaben)
- unzureichende Ressourcen (z. B. zu wenig Speicher, unterbrochene Netzverbindung)

Um ein Programm gegen derartige Fehler zu sichern, muss entsprechender Code eingefügt werden. Diese Aufgabe ist schwierig und aufwändig. Im konkreten Fall ist zu entscheiden:

- kann ein Fehler repariert werden oder ist es sinnvoll, die Anwendung zu beenden?
- wo sollte der Fehler behandelt werden?
- an welcher Stelle soll das Programm fortgesetzt werden?

Generell kann man zwischen zwei Strategien unterscheiden.

1. Der Fehler wird „an Ort und Stelle“ mit normalen Programmiermethoden behandelt. Ein Beispiel ist die Prüfung, ob eine Datei mit dem angegebenen Namen vorhanden ist. Falls nicht, wird der Benutzer erneut nach dem Namen gefragt.
2. Ein Fehler wird festgestellt, kann aber nicht behoben werden (z. B. Zugriff außerhalb der Grenzen eines Feldes). Daher wird der Fehler gemeldet und kann dann unabhängig vom normalen, linearen Programmfluss behandelt werden.

Java verfügt über ein flexibles Konzept zur Fehlerbehandlung. Damit ist es möglich, Fehler zu erkennen und entweder selbst zu behandeln oder an die aufrufende Instanz weiter zu melden.

16.2 Beispiel

Im folgenden Code soll eine Bild-Datei eingelesen werden und das Bild dann als Hintergrund verwendet werden:

```
private static void test() {
    Graphic graphic = new Graphic();
    Plotter plotter = graphic.getPlotter();
    BufferedImage img = ImageIO.read(new File("bild.jpg"));
    plotter.setBackgroundImage(img);
}
```

Allerdings scheitert die Kompilierung mit der Meldung

```
Unhandled exception type IOException
```

Das Lesen einer Datei ist eine prinzipiell fehleranfällige Aktion. Vielleicht existiert die Datei gar nicht. Oder der Prozess hat keine Leserechte. Im Beispiel soll eine Bilddatei gelesen werden. Die Datei muss also von einem bestimmten Typ sein.

Derartige Fehler zur Laufzeit lösen in Java *Exceptions* aus. Wie der Name andeutet, stellen sie Ausnahmen vom normalen Ablauf dar. Ein Methode muss die eventuell auftretenden Fehler nicht unbedingt selbst behandeln. Mit der Angabe `throws Exception` wird dem Compiler mitgeteilt, dass in der Methode ein Ausnahmefehler auftreten kann. Dieser Fehler wird an die aufrufende Methode gemeldet. Auf diese Art und Weise können die Fehler von allen übergeordneten Methoden einschließlich `main` weiter gereicht werden. Dann übernimmt die Java-Maschine die Behandlung. Aus dem Beispiel wird:

```
public static void main(String[] args) throws IOException {
    test();
}

private static void test() throws IOException {
```

Die normale Ausführung wird dann bei einem Fehler unterbrochen und die Exception bis zur Java-Maschine durchgereicht. Diese gibt dann eine Fehlermeldung mit der Ursache und dem Ort aus:

```
Exception in thread "main" javax.imageio.IIOException: Can't read input file!
    at java.desktop/javax.imageio.ImageIO.read(ImageIO.java:1308)
    at swe2_19/fehler.Fehler.test(Fehler.java:26)
    at swe2_19/fehler.Fehler.main(Fehler.java:19)
```

16.3 try - catch Anweisung

Alternativ zu der Weitergabe kann man selbst Code zur Behandlung solcher Fehler bereit stellen. Dies geschieht mittels einer try - catch Anweisung. Die Anweisung muss die kritische Stelle umfassen, kann aber ansonsten frei platziert werden. Entweder sie steht in der Methode selbst oder in einer der aufrufenden Methoden. Generell gilt, dass Fehler behandelt werden müssen. Entweder die Methode kümmert sich selbst darum oder gibt den Fehler weiter (*catch-or-throw* Regel). Tritt zur Laufzeit ein Fehler auf, so sucht die Java-Maschine nach einem entsprechenden catch-Teil. Die Suche beginnt in der aktuellen Methode. Wird dort nichts gefunden, so wird die Suche in der aufrufenden Methode fortgesetzt. Die Suche wird weitergeführt, bis die oberste Ebene main erreicht ist. Ist auch dort keine Behandlung vorgesehen, wird die Anwendung beendet und eine Meldung ausgegeben.

Kriterium für die Platzierung sollte sein: „*Kann man an dieser Stelle den Fehler angemessen behandeln?*“ Mit folgendem Code wird der Fehler vor Ort behandelt:

```
BufferedImage img;
try {
    img = ImageIO.read(new File("bild.jpg"));
    plotter.setBackgroundImage(img);
} catch (IOException e) {
    System.out.println( e );
}
```

Zunächst kommt ein try-Block, der alle kritischen Anweisungen enthält. Eventuelle Fehler werden in einem direkt anschließenden catch-Block aufgefangen. Genauer gesagt unterbricht ein Fehler die normale Ausführung in dem try-Block und die Anwendung springt an den Anfang des catch-Blocks. Der Block wird durch das Schlüsselwort `catch` sowie einen formalen Parameter eingeleitet. In dem Beispiel ist als formaler Parameter eine allgemeine Exception angegeben. Wie bei einem Methodenaufruf übernimmt der Block bei der Ausführung ein konkretes Objekt. In diesem Exception-Objekt sind Informationen zu dem aufgetretenen Fehler gespeichert. Die Methode `toString` (hier implizit durch die Verkettung der Zeichenketten aufgerufen) liefert eine Textinformation zu der Art des Fehlers.

Nach dem Ende des catch-Blocks wird die Anwendung an dieser Stelle („hinter“ der try - catch Anweisung) fortgesetzt. Es gibt keine Möglichkeit, an die Stelle zurück zu gehen, an der der Fehler auftrat. Für weitergehende Informationen kann auch die genaue Position des Fehlers angezeigt werden. Die Methode `printStackTrace()` gibt die entsprechende Information auf dem Standard Error Stream aus.

Angezeigt wird der komplette Stapel (*Stack*) mit Methodenaufrufen, wie er

zum Zeitpunkt des Fehlers vorlag. Hilfreich ist die Angabe der Zeilennummern, mit deren Hilfe die kritische Stelle im Quellcode sofort nachgeschaut werden kann. Die Syntax der try - catch Anweisung erlaubt eine Trennung zwischen „normalem“ Code und Code zur Fehlerbehandlung. Es ist möglich, längere Code-Stücke zusammenhängend zu schreiben und die Fehlerbehandlung für den gesamten Abschnitt daran anzuschließen. Damit wird die Lesbarkeit stark verbessert. Den catch-Block bezeichnet man nach seiner Funktion auch als Exception Handler.

16.4 Hierarchie von Ausnahmefehlern

In einer Anwendung können verschiedenste Fehler auftreten. So kann zum Beispiel die Methode `readLine` der Klasse `LineNumberReader` eine `IOException` verursachen. Die Dokumentation spezifiziert für jede Methode, welche Arten von Fehlern auftreten können. Die Fehler - genauer gesagt die Klassen zur Beschreibung der Fehler - sind hierarchisch organisiert. Die Basisklasse für alle Fehler ist `Throwable`. Nur Instanzen dieser Klasse oder daraus abgeleiteter Klassen können als Parameter an den catch-Block übergeben werden. Danach verzweigt sich der Ableitungsbaum zu den zwei Klassen `Error` und `Exception`.

16.4.1 Die Klasse Error

In dieser Klasse sind alle schwerwiegenden Fehler enthalten. Normalerweise wird eine Anwendung diese Fehler nicht behandeln. Beispiele sind `VirtualMachineError` oder `StackOverflowError`. In diesen Fällen treten schwerwiegende Problem mit der Java-Maschine oder den benötigten Klassen auf. Eine Anwendung hat in solchen Fällen wenig Möglichkeiten zur Reaktion. Fehler der Klasse `Error` brauchen nicht mit `throws` deklariert zu werden.

16.4.2 Die Klasse Exception

Unter dieser Oberklasse finden sich alle Fehler, die eine Anwendung sinnvollerweise behandeln kann. Die Fehler sind in mehreren Ebenen in Unterklassen organisiert. Als Beispiel hat `FileNotFoundException` folgenden Ableitungsbaum:

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.io.IOException
                java.io.FileNotFoundException
```

Für die Exception greift der Mechanismus Polymorphismus. Eine speziellere Exception kann einer allgemeineren Exception zugewiesen werden. Wir haben bisher dieses Verhalten ausgenutzt, indem wir stets die allgemeinste Klasse `Exception`

spezifiziert hatten. Damit wurden die verschiedenen speziellen Fehler aufgefangen. Auch wenn z. B. die angegebene Datei nicht existiert und damit eine `FileNotFoundException` ausgelöst wurde, so wurde dieser Fehler in dem allgemeinen catch-Block aufgefangen. Wir können den catch-Block spezialisieren, indem wir gezielt eine Unterklasse angeben. Mit

```
} catch ( NoSuchElementException ex ) {
...
}
```

wird nur noch diese spezielle Fehlerklasse (und eventuelle Unterklassen) behandelt. In einer try - catch Anweisung können mehrere catch-Blöcke für verschieden Fehlerklassen stehen. Beispiel:

```
try {
   System.out.println( "Datei <" + datei + "> lesen" );
   v.ausDatei( datei );
}
catch ( NoSuchElementException ex ) {
   System.out.println("Fehler beim Lesen: <"+ datei +">" );
   System.exit(0);
}
catch ( FileNotFoundException ex ) {
   System.out.println("Datei <"+datei+"> nicht gefunden" );
   System.exit(0);
}
```

Dabei gelten folgende Regeln:

- Ein `catch`-Block kann mehrere Klasse behandeln.
- Speziellere Handler müssen vor allgemeineren stehen (wird durch Compiler geprüft).
- Wenn die Methode dies mit `throws ...` deklariert, brauchen nicht alle Fehler behandelt zu werden.

Entsprechendes gilt für die Deklaration `throws` bei einer Methode. Anstelle des allgemeinen `throws Exception` können die möglichen Fehler einzeln benannt werden. Mehrere Fehlerklassen werden durch Komma getrennt.

16.4.3 Die Klasse RuntimeException

Eine besondere Rolle spielt die Fehlerklasse `RuntimeException`. Dies sind Fehler, die nahezu überall auftreten können. Darunter fallen

- nicht initialisierte Referenzen: `NullPointerException`

- Zugriffe außerhalb eines Feldes: `IndexOutOfBoundsException`
- arithmetische Fehler wie Division durch Null: `ArithmeticException`

Es wäre sehr aufwendig und wenig hilfreich überall diese Fehler als möglich zu deklarieren. Daher sind alle Fehlerklassen, die auf `RuntimeException` basieren, von der catch-or-throw Regel ausgenommen. Sie können allerdings auch explizit behandelt werden. Ansonsten werden sie von der Java-Maschine angezeigt.

Beispiel 16.1. Zugriff außerhalb eines Feldes

```
int[] feld = new int[20];

try {
   feld[40] = 4;
}
catch( Exception ex ) {
   System.out.println( "Exception : " + ex );
}
```

ergibt

```
Exception : java.lang.ArrayIndexOutOfBoundsException
```

16.5 Eigene Exceptions

In den Exception-Mechanismus können eigene Fehlertypen integriert werden. Ein Ausnahmefehler wird mit dem Befehl `throw` ausgelöst. Im einfachsten Fall benutzt man eine der vorhandenen Fehlerklassen und trägt einen eigenen Meldungstext ein. So könnte eine Sicherheitsabfrage über die korrekte Preiseingabe in der Form

```
if( kaufPreis < 0 ) throw new Exception("Falscher Preis");
```

eingebaut werden. Im Fehlerfall würde die Exception von dem allgemeinen catch-Block gefangen werden:

```
Allgemeiner Fehler bei Datei <autos.txt>
Exception : java.lang.Exception: Falscher Preis
```

Weiterhin ist es möglich, eigene Fehlerklassen zu erzeugen. Ein Beispiel ist

```
public class FalschesJahrException
         extends java.lang.Exception {

    public FalschesJahrException() { }
```

```
    public FalschesJahrException(String msg) {
        super(msg);
    }
}
```

und

```
if( kaufJahr < 1900 | kaufJahr > Calendar.getInstance().get(Calendar.YEAR) ) {
    throw new FalschesJahrException( "Jahr " + kaufJahr );
}
```

mit dem Ergebnis

```
Allgemeiner Fehler bei Datei <autos.txt>
Exception : FalschesJahrException: Jahr 12000
```

16.6 finally-Block

Eine try - catch Anweisung kann mit einen optionalen `finally`-Block abgeschlossen werden. Der Block steht nach dem letzten catch-Block. Unabhängig davon, wie die Anweisung verlassen wird – normal oder durch einen Ausnahmefehler – wird der `finally`-Block ausgeführt. Dies gilt selbst dann, wenn der `try`-Block durch `return`, `break` oder `continue` vorzeitig beendet wird oder die Exception nur weiter gereicht wird. Damit ist der `finally`-Block der ideale Platz für allgemeine „Aufräumarbeiten“. Betrachten wir folgendes Beispiel:

```
try {
   lnr = new LineNumberReader( new FileReader( datei ) );
   for ( int i=0; i<autos.length; i++ ) {
      autos[i] = new Auto();
      autos[i].parseLine( lnr.readLine() );
   }
}
catch (Exception ex ) {
   System.out.println( "Fehler beim Lesen <" +datei+ ">" );
   return -1;
}
```

Angenommen die Datei wird zwar erfolgreich geöffnet, aber beim Lesen der Zeilen kommt es zu einem Fehler. In diesem Fall wird der `try`-Block abgebrochen und der `catch`-Block ausgeführt. Die Datei würde aber geöffnet bleiben. Durch eine Block in der Art

```
finally {
   System.out.println( "Datei schliessen" );
```

```
   lnr.close();
}
```

kann sichergestellt werden, dass die Datei geschlossen wird. Der `finally`-Block wird in jedem Fall ausgeführt. Durch diese Konstruktion braucht der Code zum gesicherten Freigeben von Ressourcen nur einmal geschrieben zu werden. Bei umfangreichem Code wird das Programm dadurch übersichtlicher und eine Fehlerquelle wird vermieden.

16.7 Assertions

Assertions sind ein Möglichkeit, Annahmen während der Ausführung eines Programms zu überprüfen. Die Syntax ist

```
assert logischer Ausdruck : Meldung;
```

wobei der Meldungsteil optional ist. Ein Beispiel ist

```
assert prozent >= 0 : "Es gibt keine negativen Prozente";
```

Ist die Bedingung nicht erfüllt, bricht das Programm ab. Die Anweisung ist kompakter als eine entsprechende if-Anweisung. Weiterhin können Sie durch eine Option bei der Ausführung an- oder ausgeschaltet werden. Damit ist es möglich, Assertions in der Entwicklungs- und Testphase einzuschalten, um eventuelle Programmierfehler aufzudecken. Später in der Produktivumgebung können Sie ausgeschaltet werden, um den damit verbundenen Aufwand zu vermeiden. Die folgende Methode zeigt ein Beispiel für den Einsatz von Assertions.

```
private double note(int prozent) {
        // Precondition
        assert prozent >= 0 : "Es gibt keine negativen Prozente";
        double note;
        if( prozent < 50 ) {
                note = 5.0;
        } else {
                note = 4 - 3 * ( prozent - 50. ) / 45;
        }
        // Postcondition
        assert 1. <= note & note <= 5.0 : "Ungueltige Note";
        return note;
}
```

Dabei wird sowohl beim Argument als auch beim Ergebnis der Wertebereich überprüft. Die Prüfung von Vorbedingungen (*preconditions*) und Nachbedingungen (*postconditions*) ist angelehnt an das Konzept *Design by contract* (kurz DbC,

englisch für *Entwurf gemäß Vertrag*), das mit der Programmiersprache Eiffel eingeführt wurde. Bei dieser Implementierung würde man feststellen, dass für Prozentwerte zwischen 95 und 100 Noten kleiner als 1 resultieren. Für solche Werte müsste noch eine weitere Abfrage eingebaut werden.

Es handelt sich im Beispiel um eine private – also interne – Methode, bei der man davon ausgehen kann, dass sie nur mit sinnvollen Werten aufgerufen wird. Bei einer öffentlichen Methode wäre es eher sinnvoll, eine Exception auszulösen.

16.8 Übungen

Übung 16.8.1. Gegeben sei der folgende Code:

```
// Klasse ExTest
import java.util.*;
import java.io.*;

public class ExTest {

    public static void main(String[] args) throws Exception{
        BufferedReader din = new BufferedReader(
            new InputStreamReader( System.in ) );

        for ( ;; ) {
            System.out.print( "> " );
            // Zeile einlesen
            String text = din.readLine();
            test( text );
        }
   }
    static void test( String zeile ) {
        String[] teile = zeile.split(" ");

        for (int i = 0; i<teile.length; i++ ) {
            int wert = Integer.parseInt( teile[i] );
            System.out.println( i + ". wert: " + wert );
        }
    }
}
```

- Welche Fehler können bei geänderter Eingabe auftreten?
- Behandeln Sie diese Fehler mittels try-catch Blocks in main.

- Falls eine Zahl den Wert 0 annimmt, soll ein neuer Ausnahmefehler angezeigt werden. Implementieren Sie dazu eine Klasse `WertNullException`.

Anhang A

BoSym Details

A.1 BoSL

Intern werden die Befehle für BoSym in BoSym-Language (BoSL) übertragen. Spezielle Klassen übernehmen intern die Umsetzung der Funktionen in entsprechende Befehle. Beispielsweise wird der Aufruf `farbe2(1, 2, 0xff)` in den Befehl `#color2 1, 2, 0xff` übersetzt. Die gesendeten Befehle kann man im Menüpunkt *Datei* → *Erhaltene Befehle* einsehen. Dort wird auch auf Befehle hingewiesen, die mehrfach mit den gleichen Argumenten verwendet wurden. Dies ist oft ein Indikator für suboptimalen Code. Zu den verschiedenen Methoden gehören folgende BoSL-Befehle:

- `clear` löscht alle Farben
- `resize` n m setzt die Anzahl der Spalten und Zeilen.
- `color` m f (zwei Zahlen), die Nummer des Elementes und der RGB-Wert für die Farbe. RGB-Werte können auch als Hex-Werte eingegeben werden. Beispiel: `color 5 0xff0000` färbt die 5. Anzeige rot.
- `color2` n m f das Gleiche, aber mit xy-Koordinaten
- `colors` f färbt alle Symbole um.
- `forms [scd*r]` stellt auf entsprechende Symbole um. Dei eckigen Klammern stehen für die Auswahl eines der Zeichen aus der Liste. Einige Möglichkeiten:
  - `s` Quadrat (Square)
  - `c` Kreis (Circle)
  - `d` Raute (Diamond)
  - `*` Stern

  - `r` Zufällig (Random)

  Beispiel: `forms c` wechselt alle Elemente zu Kreisen.

- `form` n `[scd*r]` stellt nur das angegebene Element um
- `fomr2` n m `[scd*r]` (`fi` mit xy-Koordinaten)
- `symbolSize` n r verändert die Größe das angegebene Element
- `symbolSize2` n m r (`s` mit xy-Koordinaten)
- `symbolSizes` r verändert die Größe aller Symbole.
- `ba` f allgemeine Hintergrundfarbe
- `border` *border* Rahmenfarbe
- `background` n f Hintergrundfarbe für das angegebene Element
- `background2` n m f (`b` mit xy-Koordinaten)
- `text` n *text* Text für das angegebene Element
- `text2` n m *text* (`T` mit xy-Koordinaten)
- `textColor` n f Textfarbe für das angegebene Symbol
- `textColor2` n m *text* (`TC` mit xy-Koordinaten)
- `statusText` *text* schreibt den übergebenen Text in das Statusfeld
- `p` holt die älteste Eingabe (*poll*). Dazu kann über den Menüpunkt *interaktiv* die Eingabezeile eingeblendet werden. Auch Maus-Klicks werden als Eingaben behandelt.

Üblicherweise wird man BoSL nicht direkt verwenden sondern auf die deutlich besser zu lesenden Methoden zurückgreifen. Um die Anzahl der Methoden überschaubar zu halten, sind allerdings nicht für alle BoSL-Befehle eigene Methoden implementiert. In diesem Fall muss man auf eine allgemeine Methode zurückgreifen, um die Befehle direkt an eine `Board`-Instanz zu schicken:

```
board.receiveMessage( befehl );
```

Bei Anwendungen in der Sprache C entspricht dem der Aufruf der Funktion `sendMessage(zeile)`. Auf diese Art und Weise können auch die folgenden zusätzlichen BoSL-Befehle geschickt werden:

- `n` schaltet die Nummerierung der Symbole ein oder wieder aus.

- `s` gibt die Größe (Anzahl Zeilen und Spalten) zurück.
- `sleep n` Board wartet für die angegebene Zeit (Wert in Millisekunden).
- `image` x y *dateiname* das Bild aus der angegebenen Datei wird an der Stelle (x, y) angezeigt. Dabei wird es auf die Größe des Symbols skaliert. Ein eventuell an dieser Stelle bereits vorhandenes Bild wird gelöscht.
- `image` x y `-` das Bild an der Stelle (x, y) wird gelöscht.
- `bgImage` *dateiname* das Bild wird als Hintergrund eingefügt.
- `fontsize` n setzt die Font-Größe für alle weiteren Texte.
- `borderwidth` w Rahmenstärke (float)
- `backgrounds` f Hintergrundfarbe für alle Symbole
- `graphicBorder` f Farbe für Randbereich
- `clearAllText` löscht alle Symbol-Texte.
- `statusfontsize` n Setzt die Font-Größe für die Statuszeile.
- `statuscolor` f Setzt die Farbe Statuszeile.
- `fonttype` *name* Wählt einen Font für alle Texte aus.
- *m1 f1 m2 f2 m3 f3 ...* (eine Folge von Paaren Feldnummer-Farbe), die angegebenen Felder werden in der jeweiligen Farbe gezeichnet.
- `clearCommands` löscht alle ausstehenden Eingaben.
- `button` *text region* legt einen Knopf (Button) in der angegebenen Region (*east*, *south*, *west*, *north*) an. Beim Betätigen wird der angegebene Text als Eingabe geschickt.
- `removeAllButtons` entfernt alle angelegten Eingabe-Knöpfe.
- `toggleInput` blendet das Eingabefeld ein oder aus.

Wichtig: Aus einem Snippet müssen die Befehle mit `>>` beginnen. Ansonsten müssen sie mit einem Zeilenende (`\n`) abgeschlossen werden.

A.2 Interaktion

Die Hilfs-Klasse `Dialogs` enthält diverse Methoden für Ein- und Ausgabe. Einige Methoden davon sind:

- `String askString(String title)`: Abfrage nach einer Zeichenkette mit dem angegebenen Text als Titel im Dialogfenster.
- `String askStringMessage(String message)`: Variante, bei der der Abfragetext übergeben wird.
- `String askInteger(String title))`: Abfrage nach einer ganzen Zahl.
- `String askIntegerMessage(String message)`: Variante, bei der der Abfragetext übergeben wird.
- `String askInteger(String title, int lower, int upper))`: Abfrage nach einer ganzen Zahl aus dem Bereich $l \leq n < u$.
- `String askIntegerMessage(String message, int lower, int upper)`: Variante, bei der der Abfragetext übergeben wird.
- `double askDouble(String title))`: Abfrage nach einer Gleitkommazahl.
- `Double askDDouble(String title))`: Der Rückgabewert ist eine Instanz der Klasse `Double`. Bei Dialogabbruch wird der Wert `null` zurückgegeben. Damit kann die Anwendung auf einen Abbruch reagieren.
- `void showString( String text )`: Anzeige einer Zeichenkette.
- `void showStrings( String[] text )`: Anzeige eines Feldes mit Zeichenketten.

Die Liste kann leicht nach Bedarf erweitert werden.

Literaturverzeichnis

[Boo94] Grady Booch. *Objektorientierte Analyse und Design: Mit praktischen Anwendungsbeispielen*. Addison Wesley, 1994.

[C+12] Unicode Consortium et al. The unicode standard version 6.2–core specification. *Unicode Inc., Mountain View, California, USA*, 2012.

[Dij82] Edsger W. Dijkstra. Why numbering should start at zero. August 1982.

[Dob80] Wlodzimierz Dobosiewicz. An efficient variation of bubble sort. *Information Processing Letters*, 11(1):5–6, 1980.

[Eul15] S. Euler. Java plotter. TH Mittelhessen, 2015.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *Computing Surveys*, 1991.

[Hoa62] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[Knu98] Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[LR10] Evgeny Lakshtanov and Vera Roshchina. Finiteness in the card game of war. *Arxiv preprint arXiv10071371*, pages 1–11, 2010.

[R+00] Achut Reddy et al. Java™ coding style guide. *Sun MicroSystems*, 2000.

[Sed02] Robert Sedgewick. *Algorithms in Java, Parts 1–4 (Fundamental Algorithms, Data Structures, Sorting, Searching)*. Addison-Wesley, 2002.

[She59] Donald L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32, 1959.

[Yar09] Vladimir Yaroslavskiy. Dual-pivot quicksort. *Research Disclosure*, 2009.

Index