Leseprobe

Hans-Peter Habelitz zeigt Ihnen, dass der Einstieg in die Programmierung von Java leicht gelingen kann. Er macht Sie in dieser Leseprobe mit den Grundlagen vertraut. Außerdem können Sie einen Blick in das vollständige Inhalts- und Stichwortverzeichnis des Buches werfen.

Hans-Peter Habelitz

Programmieren lernen mit Java

537 Seiten, broschiert, mit DVD, 3. Auflage 2015
19,90 Euro, ISBN 978-3-8362-3517-4

www.rheinwerk-verlag.de/3776

Kapitel 2
Grundbausteine eines Java-Programms

Der große Weg ist sehr einfach, aber die Menschen lieben die Umwege.
(Laotse, Tao Te King, übers. Zensho W. Kopp)

Im vorigen Kapitel haben Sie bereits Java-Programme erstellt. Dabei waren die Quelltexte vorgegeben, weil Sie zunächst die grundlegenden Arbeitsabläufe und die Werkzeuge, die dafür benötigt werden, kennenlernen sollten. In diesem Kapitel stehen nun die Sprachelemente von Java im Mittelpunkt.

2.1 Bezeichner und Schlüsselwörter

Bezeichner sind Namen für Elemente, die im Programm verwendet werden. Sie sind nicht von Java vorgegeben, sondern werden vom Programmierer, also von Ihnen, als Namen für die Elemente festgelegt, die Sie verwenden möchten. Bezeichner können aus beliebig vielen Zeichen und Ziffern bestehen, müssen aber immer mit einem Buchstaben beginnen. Zu den Buchstaben gehören auch Währungszeichen (wie z. B. das Dollarzeichen `$`) und Sonderzeichen wie der Unterstrich `_`. Groß- und Kleinschreibung werden unterschieden. Das heißt, dass `zahl` ein anderer Bezeichner ist als `Zahl`. Bezeichner können frei gewählt werden, dürfen aber nicht mit *Schlüsselwörtern* der Sprache und den *Literalen* `true`, `false` und `null` übereinstimmen, die in Java eine bereits festgelegte Bedeutung haben.

Am Beispiel der Übungsaufgabe 2 des vorigen Kapitels können Sie leicht nachvollziehen, an welchen Stellen im Quellcode Bezeichner und Schlüsselwörter verwendet werden:

```
/* Kreisberechnung: Für einen Kreis werden der Umfang und der
 * Flächeninhalt berechnet.
 * Der Kreisradius wird beim Programmstart als Parameter
 * übergeben.
*/
```



```
public class Kreisberechnung2 {
  public static void main(String[] args) {
    double radius;
    double umfang, inhalt;
    radius = Double.parseDouble(args[0]);
    umfang = 2.0 * 3.1415926 * radius;
    inhalt = 3.1415926 * radius * radius;
    System.out.print("Umfang: ");
    System.out.println(umfang);
    System.out.print("Flaeche: ");
    System.out.println(inhalt);
  }
}
```

Listing 2.1 Quellcode der Aufgabe 2 aus Kapitel 1

In Listing 2.1 werden als Bezeichner `Kreisberechnung2` sowie `radius`, `umfang` und `inhalt` verwendet.

Welche Bezeichner bereits als Schlüsselwörter vergeben sind, sehen Sie in Tabelle 2.1. Sie listet die in Java reservierten Schlüsselwörter auf.

Schlüsselwörter von Java				
abstract	default	if	protected	throws
assert	do	implements	public	transient
boolean	double	import	return	try
break	else	instanceof	short	void
byte	enum	int	static	volatile
case	extends	interface	strictfp	while
catch	final	long	super	
char	finally	native	switch	
class	float	new	synchronized	
const	for	package	this	
continue	goto	private	throw	

Tabelle 2.1 Schlüsselwörter in Java

Die Bedeutung jedes einzelnen Schlüsselwortes soll im Augenblick nicht erläutert werden. Die Erläuterungen werden dort folgen, wo die Schlüsselwörter eingesetzt werden. Die Liste soll hier nur zeigen, welche Bezeichner Sie als Programmierer für eigene Zwecke *nicht* verwenden dürfen.

In Listing 2.1 werden als Schlüsselwörter vor dem Klassenbezeichner `Kreisberechnung2` z. B. `public class` verwendet, und vor den Bezeichnern `radius`, `umfang` und `inhalt` steht das Schlüsselwort `double`.

Die in Kapitel 1 erwähnten *Code Conventions* enthalten auch *Namenskonventionen* (*Naming Conventions*).

Namenskonventionen

- *Bezeichner* werden mit gemischter Groß- und Kleinschreibung geschrieben. Großbuchstaben dienen dem Trennen von Wortstämmen, z. B. `kreisRadius`, `mittlererWert`.
- *Variablennamen* beginnen mit Kleinbuchstaben, z. B. `meinKonto`, `anzahlZeichen`. Namen von Konstanten werden mit Großbuchstaben geschrieben. Einzelne Wörter werden durch den Unterstrich _ getrennt, z. B. `MAX_WERT`.
- *Klassennamen* beginnen mit einem Großbuchstaben, z. B. `ErstesBeispiel`. Da Klassennamen als Teil des Namens der Datei verwendet werden, die die Klasse im Bytecode enthält, unterliegen diese auch den Regeln des jeweiligen Betriebssystems.

Wie bereits erwähnt wurde, handelt es sich bei den genannten Konventionen um freiwillige Vereinbarungen, die keineswegs eingehalten werden müssen. Sie haben sich aber in weiten Bereichen durchgesetzt und sind Zeichen professionellen Programmierens.

2.2 Kommentare

Kommentare im Quellcode sind Texte, die vom Compiler beim Übersetzen nicht beachtet werden. Mit Kommentaren können Sie für sich selbst und für andere Leser Hinweise in den Quellcode einfügen.

In Java können drei unterschiedliche Arten von Kommentaren verwendet werden:

- **Einzeilige Kommentare**
  Sie beginnen mit // und enden automatisch mit dem Ende der Zeile.

  Beispiel:

```
int anzahl; // zählt die gelesenen Zeichen
```


- **Mehrzeilige Kommentare**
  Sie beginnen mit /* und enden mit */. Da für das Ende des Kommentars eine Zeichenfolge eingegeben werden muss, kann sich der Kommentar über mehrere Zeilen erstrecken.

  Achtung: Der Kommentar darf die Zeichenfolge */ nicht enthalten, denn dadurch würde der Kommentar beendet.

  Beispiel:

```
/* Dieser Kommentar ist etwas länger
und erstreckt sich über zwei Zeilen.
*/
```

  Die Zeichenfolge /* und */ muss nicht am Zeilenanfang stehen. Der Kommentar kann an beliebiger Stelle beginnen.
- **Dokumentationskommentare**
  Sie beginnen mit /** und enden mit */ und können sich ebenfalls über mehrere Zeilen erstrecken. Sie werden gesetzt, um vom JDK-Werkzeug *javadoc* automatisch eine Programmdokumentation erstellen zu lassen.

Nach den Code Conventions sollte jedes Programm mit einem beschreibenden Kommentar beginnen. Innerhalb des Programmtextes können weitere Kommentare eingefügt werden, um z. B. Aufgaben von Klassen, Methoden und Variablen zu erläutern.

2.3 Variablen und Datentypen

Sollen in einem Programm Daten zur Be- und Verarbeitung zur Verfügung gestellt werden, so werden *Variablen* als Behälter benötigt. Variablen können als Namen für einen Speicherplatz im Hauptspeicher aufgefasst werden. An diesem Speicherplatz wird der Wert der Variablen abgelegt. Der Wert kann dann im Laufe des Programmablaufs ausgelesen und verändert werden. Wie groß dieser Speicherplatz ist und welche Art von Daten darin abgelegt werden können, wird durch den Datentyp festgelegt. Durch die begrenzte Größe des Speicherbereichs ist auch der Wertebereich der Variablen begrenzt (siehe Abbildung 2.1).

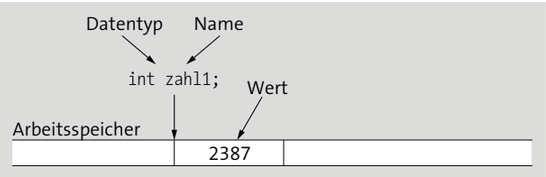

Abbildung 2.1 Variablendefinition

Zur Deklaration geben Sie den Datentyp und – durch ein Leerzeichen getrennt – den Namen der Variablen an. Abgeschlossen wird die Deklaration wie jede Anweisung durch ein Semikolon:

```
Datentyp variablenname;
```

Werden mehrere Variablen des gleichen Typs benötigt, dann kann hinter dem Datentyp auch eine Liste der Variablennamen folgen. Die Liste besteht aus den durch Kommata getrennten Variablennamen:

```
Datentyp variablenname1, variablenname2, ...;
```

Merke

Eine Variablendeklaration besteht aus dem *Datentyp*, gefolgt von einem einzelnen *Variablenbezeichner* oder einer durch Kommata getrennten Liste von Variablenbezeichnern. Sie wird durch ein Semikolon abgeschlossen.

Java kennt acht sogenannte *primitive Datentypen*, die Sie in Tabelle 2.2 aufgelistet finden.

Datentyp	Verwendung	Größe in Byte	Größe in Bit	Wertebereich
`boolean`	Wahrheitswert	1	8	`false`, `true`
`char`	Zeichen	2	16	0 bis 65.535
`byte`	Ganzzahl	1	8	–128 bis 127
`short`	Ganzzahl	2	16	–32 768 bis 32.767
`int`	Ganzzahl	4	32	–2.147.483.648 bis 2.147.483.647
`long`	Ganzzahl	8	64	–9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
`float`	Kommazahl	4	32	Betrag ca. $1{,}4 \times 10^{-45}$ bis $3{,}4 \times 10^{38}$ (Genauigkeit ca. sieben Stellen)
`double`	Kommazahl	8	64	Betrag ca. $4{,}9 \times 10^{-324}$ bis $1{,}7 \times 10^{308}$ (Genauigkeit ca. 15 Stellen)

Tabelle 2.2 Primitive Datentypen

Konkrete Werte wie die Zahlen 13, 28, 1.5 werden als *Literale* bezeichnet. Beachten Sie, dass im Java-Quellcode die englische Notation gilt. Deshalb ist bei Kommazahlen der Punkt als Dezimaltrennzeichen zu verwenden.

Merke

Bevor eine Variable in einem Programm verwendet werden kann, muss sie deklariert werden. Dabei werden der Datentyp und der Name (Bezeichner) festgelegt.

Als Dezimaltrennzeichen wird der Punkt verwendet.

2.3.1 Namenskonventionen für Variablen

Die Namenskonventionen machen zu Variablenbezeichnern folgende Aussagen:

- Variablennamen werden in gemischter Groß-, Kleinschreibung geschrieben, beginnen aber immer mit einem Kleinbuchstaben, z. B. `zahl1`, `mittelwert`, `kleinsteZahl`.
- Setzen sich Variablennamen aus mehreren Wörtern zusammen, werden die internen Wörter mit Großbuchstaben begonnen, z. B. `groessterRadius`, `anzahlSpieler`.
- Variablenbezeichner sollten kurz und dennoch aussagekräftig sein, z. B. `ggT` statt `groessterGemeinsamerTeiler`.
- Variablenbezeichner, die nur aus einem Buchstaben bestehen, sollten vermieden werden. Sie sollten lediglich als kurzlebig verwendete Variablen, z. B. als Schleifenzähler, eingesetzt werden.

Die Variablenbezeichner kurz und aussagekräftig zu halten, ist in der deutschen Sprache nicht immer ganz einfach. Viele Programmierer weichen deshalb oft auch bei Variablenbezeichnern auf die englische Sprache aus.

Beispiele für Variablendeklarationen:

- `boolean gefunden;`
- `char zeichen;`
- `short s1, s2, s3, s4;`
- `int i, j, k;`
- `long grosseZahl;`
- `float ePreis;`
- `double radius, umfang;`

2.3.2 Wertzuweisung

Der Wert einer Variablen wird durch eine *Wertzuweisung* festgelegt. Die Wertzuweisung ist ein Speichervorgang, für den der *Operator* = verwendet wird. Dabei wird der Wert des Ausdrucks, der rechts vom Gleichheitszeichen steht, in der Variablen gespeichert, die links vom Gleichheitszeichen steht.

Durch die Wertzuweisung

```
zahl1 = 1234;
```

wird entsprechend in der Variablen `zahl1` der Wert `1234` gespeichert (siehe Abbildung 2.2). Dabei wird immer der ursprüngliche Wert der Variablen durch den neuen Wert überschrieben (siehe Abbildung 2.3).

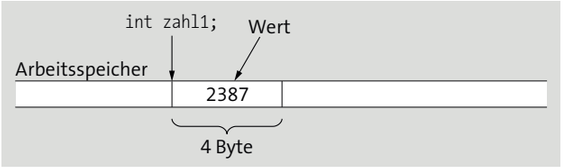

Abbildung 2.2 Variable vor der Wertzuweisung

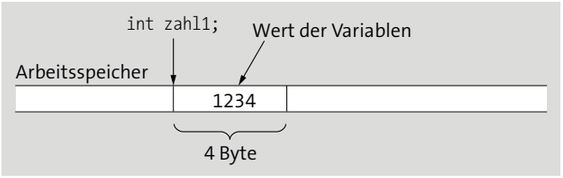

Abbildung 2.3 Variable nach der Wertzuweisung

Eine Wertzuweisung kann auch mit der Definition verbunden werden:

```
int zahl1 = 1234;
```

Dadurch wird bei der Variablendefinition direkt auch der Wert definiert, der in der Variablen gespeichert sein soll. Diese erste Zuweisung eines Wertes an eine Variable wird *Initialisierung* genannt.

2.3.3 Die primitiven Datentypen im Einzelnen

In den folgenden Abschnitten werden die primitiven Datentypen näher erläutert, bevor Sie in einigen Beispielen den Umgang mit diesen Datentypen üben können.

»boolean«

Dieser Datentyp wird als *Wahrheitswert* bezeichnet. Er kann nur einen von zwei Werten (Literalen) annehmen (`true` oder `false`). Er wird überall dort benötigt, wo Entscheidungen zu treffen sind.

»char«

Der *Zeichentyp* `char` dient dazu, ein einzelnes Zeichen des Unicode-Zeichensatzes zu speichern. Literale werden zwischen einfachen Anführungszeichen angegeben (z. B. `'a'` für den Buchstaben *a*). Mithilfe sogenannter Escape-Sequenzen können auch Zeichen mit einer ganz speziellen Bedeutung angegeben werden. Eine Escape-Sequenz beginnt mit dem Backslash-Zeichen (\), dem das eigentliche Zeichen folgt. In der Zeichenfolge `\t` z. B. wird durch das Backslash-Zeichen angegeben, dass der Buchstabe t nicht als einfacher Buchstabe zu verstehen ist, sondern als ein Tabulatorzeichen. Tabelle 2.3 gibt einen Überblick über die wichtigsten Escape-Sequenzen.

Escape-Sequenz	Bedeutung
`\b`	Backspace
`\t`	Tabulator
`\n`	Neue Zeile (Newline)
`\f`	Seitenvorschub (Formfeed)
`\r`	Wagenrücklauf (Carriage return)
`\"`	Doppeltes Anführungszeichen "
`\'`	Einfaches Anführungszeichen '
`\\`	Backslash \

Tabelle 2.3 Escape-Sequenzen

Hinweis für OS X-User

Den Backslash (\) erreichen Sie auf der Mac-Tastatur mit der Tastenkombination ⇧ + alt + 7.

»byte«, »short«, »int« und »long«

Die *Ganzzahlentypen* sind vorzeichenbehaftet. Das heißt, sie können positiv oder negativ sein. Wie in der Mathematik üblich, muss bei positiven Zahlenwerten das Vorzei-

chen nicht angegeben werden. Negative Werte erhalten wie gewohnt das vorangestellte negative Vorzeichen. Die vier unterschiedlichen Datentypen für ganze Zahlen unterscheiden sich lediglich durch den Wertebereich. Wie Sie Tabelle 2.2 entnehmen können, lassen sich im Datentyp `byte` nur Zahlenwerte von –128 bis 127 speichern. Für größere Zahlenwerte müssen Sie auf einen der drei übrigen Ganzzahltypen ausweichen. Je größer der Wertebereich eines Datentyps ist, desto mehr Speicherplatz wird durch ihn belegt. Bei den heute verfügbaren Speichergrößen spielt das Argument, dass man durch eine geschickte Wahl der Datentypen Speicherplatz einsparen kann, nicht mehr eine so große Rolle. Sie sollten deshalb den Standardtyp `int` für ganze Zahlen verwenden und nur dann davon abweichen, wenn Sie sicher sind, dass der Wertebereich nicht ausreicht oder auf jeden Fall unnötig groß gewählt ist.

»float« und »double«

Zur Speicherung von Kommazahlen stehen *Fließkommazahlentypen* zur Verfügung. Wie bei den ganzzahligen Datentypen unterscheiden sich diese beiden Typen durch den Wertebereich (siehe Tabelle 2.2), den die zu speichernden Zahlenwerte umfassen können. Zusätzlich unterscheiden sich die beiden Datentypen durch die Genauigkeit. In einem `float` können die Zahlenwerte auf circa sieben Nachkommastellen genau gespeichert werden. Der Datentyp `double` ermöglicht eine Genauigkeit von circa 15 Nachkommastellen. Als Standardtyp sollten Sie `double` verwenden. Literale von Fließkommazahlen werden in dezimaler Form geschrieben. Sie können aus einem Vorkommateil, einem Dezimalpunkt, einem Nachkommateil, einem Exponenten und einem Suffix bestehen. Es muss mindestens der Dezimalpunkt, der Exponent oder das Suffix vorhanden sein, damit das Literal von einer ganzen Zahl unterschieden werden kann. Wird ein Dezimalpunkt verwendet, so muss vor oder nach dem Dezimalpunkt eine Ziffernfolge stehen. Dem Vorkommateil und dem Exponenten kann ein Vorzeichen (+ oder –) vorangestellt werden. Der Exponent wird durch ein `e` oder `E` eingeleitet und steht für »mal 10 hoch dem Exponenten« ($\times 10^{\text{Exponent}}$). Wird kein optionales Suffix angegeben, wird das Literal als `double` interpretiert. Mit dem Suffix `f` oder `F` wird das Literal ausdrücklich zum `float`, mit dem Suffix `d` oder `D` wird es ausdrücklich zum `double` erklärt.

Beispiele für gültige Fließkommaliterale:

```
2.5       .3       -4.       -1.3e5       56.234f
```

2.3.4 Praxisbeispiel 1 zu Variablen

Die folgenden Darstellungen sollen helfen, die theorielastigen Ausführungen zu Variablen und Datentypen verständlicher zu machen. Wir erstellen dafür ein Java-Projekt mit dem Namen *JavaUebung02*. Legen Sie also in der Arbeitsumgebung neben dem Ordner *JavaUebung01* einen zweiten Ordner mit dem Namen *JavaUebung02* an.

Im Projekt *JavaUebung02* legen Sie zunächst eine Klasse mit dem Namen `Variablen1` an. Dazu erstellen Sie eine neue Textdatei mit dem Namen *Variablen1*, in der Sie die gleichnamige Klasse mit ihrer `main`-Methode anlegen:

```
public class Variablen1 {
    public static void main(String[] args) {
```

In der `main`-Methode dieser Klasse sollen die folgenden Variablen deklariert werden:

- `bZahl` als `byte`
- `sZahl` als `short`
- `iZahl` als `int`
- `lZahl` als `long`
- `fZahl` als `float`
- `dZahl` als `double`
- `bestanden` als `boolean`
- `zeichen` als `char`

Das können Sie bereits selbst. Vergleichen Sie Ihr Ergebnis mit Listing 2.2.

Nun sollen Sie den Variablen die in Tabelle 2.4 vorgegebenen Werte zuweisen. Vergleichen Sie Ihr Ergebnis wieder mit Listing 2.2.

Variable	Wert
`bZahl`	28
`sZahl`	–18453
`iZahl`	4356576
`lZahl`	345236577970
`fZahl`	4.37456678
`dZahl`	3645.564782
`bestanden`	`true`
`zeichen`	%

Tabelle 2.4 Wertzuweisungen

Wir wollen nun aber noch einen Schritt weitergehen und die Variablen mit der Anweisung `System.out.println` bzw. `System.out.print` in der Konsole ausgeben. Dabei sollen

in jeweils einer Zeile der Name der Variablen und der Wert der Variablen, z. B. nach folgendem Muster, stehen:

```
bZahl = 28
```

Sie haben die Anweisung `System.out.print` bzw. `println` bereits in den ersten Übungsaufgaben verwendet. Die genauere Bedeutung der drei durch Punkte getrennten Bezeichner werden Sie in Kapitel 5, »Klassen und Objekte«, und Kapitel 6, »Mit Klassen und Objekten arbeiten«, erfahren. Wie in Abschnitt 1.3.2, »Wie sind Java-Programme aufgebaut?«, erläutert wurde, unterscheiden sich `print` und `println` lediglich dadurch, dass `println` nach der Ausgabe noch einen Zeilenvorschub erzeugt. Dadurch stehen die folgenden Ausgaben in einer neuen Zeile. In den ersten Programmbeispielen wurden die beiden Methoden verwendet, um konstante Texte (Literale) auszugeben. Solche Text-Literale (Stringliterale) erkennen Sie daran, dass sie zwischen Anführungszeichen stehen. Das folgende Beispiel stammt aus dem Hallo-Welt-Programm:

```
System.out.println("Hallo Welt!");
```

Die beiden `print`-Anweisungen sind sehr flexibel und können nicht nur Texte ausgeben. Übergeben Sie in der Klammer einen Variablennamen, so wird von der `print`-Anweisung der Wert der Variablen ausgegeben. Damit können Sie diese Methoden sehr gut nutzen, um zu prüfen, ob Wertzuweisungen an Variablen erfolgreich ausgeführt wurden.

Zur Kontrolle der Wertzuweisungen ergänzen Sie jetzt noch die Ausgabeanweisungen mit `System.out.print` bzw. `System.out.println`. Verwenden Sie für jede Variable einen eigenen `println`-Befehl. Geben Sie jeweils zuerst mit `print` den Namen der Variable, gefolgt von einem Gleichheitszeichen, aus. Für die folgende Ausgabe des Variablenwertes verwenden Sie `println`, damit die folgende Ausgabe des nächsten Variablennamens in einer neuen Zeile steht. Hier nun der vollständige Quelltext:

```
/* Programm zum Testen der Verwendung von Variablen
 * Datum: 2011-11-30
 * Hans-Peter Habelitz
 */

public class Variablen1 {
  public static void main(String[] args) {

    // Variablendeklarationen
    byte bZahl;
    short sZahl;
    int iZahl;
    long lZahl;
    float fZahl;
    double dZahl;
    boolean bestanden;
    char zeichen;

    // Wertzuweisungen
    bZahl = 28;
    sZahl = -18453;
    iZahl = 4356576;
    lZahl = 345236577970;
    fZahl = 4.37456678;
    dZahl = 3645.564782;
    bestanden = true;
    zeichen = '%';

    // Ausgabe der Variablenwerte
    System.out.print("bZahl=");
    System.out.println(bZahl);
    System.out.print("sZahl=");
    System.out.println(sZahl);
    System.out.print("iZahl=");
    System.out.println(iZahl);
    System.out.print("lZahl=");
    System.out.println(lZahl);
    System.out.print("fZahl=");
    System.out.println(fZahl);
    System.out.print("dZahl=");
    System.out.println(dZahl);
    System.out.print("bestanden=");
    System.out.println(bestanden);
    System.out.print("zeichen=");
    System.out.println(zeichen);
  }
}
```

Listing 2.2 Quelltext zu Aufgabe 1

Auch wenn Sie den Quelltext fehlerfrei von oben übernommen haben, werden Sie beim Übersetzen die Fehlermeldung aus Abbildung 2.4 erhalten. Der Compiler meldet: `integer number too large`. Ein ganzzahliger Wert innerhalb des Quellcodes wird vom Java-

Compiler immer als `int`-Wert (Standard für ganzzahlige Zahlenwerte) interpretiert. Das gilt auch, wenn wie hier auf der linken Seite der Wertzuweisung eine Variable vom Typ `long` angegeben ist. Soll ein ganzzahliger Zahlenwert als `long` interpretiert werden, so müssen Sie dies dem Compiler durch Anhängen des Buchstabens `L` (Klein- oder Großbuchstabe) anzeigen. Wegen der besseren Lesbarkeit sollte der Großbuchstabe verwendet werden, da der Kleinbuchstabe sehr leicht mit der Ziffer 1 (eins) verwechselt werden kann:

```
lZahl = 345236577970L;
```

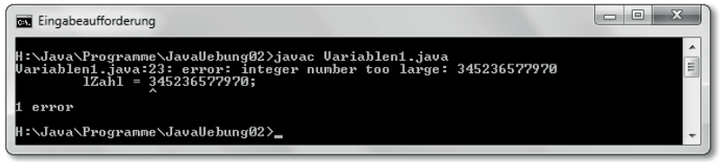

Abbildung 2.4 Fehlermeldung beim ersten Kompilieren

Ergänzen Sie also die Zahlenangabe entsprechend, und starten Sie die Übersetzung erneut. Sie werden eine weitere Fehlermeldung erhalten (siehe Abbildung 2.5).

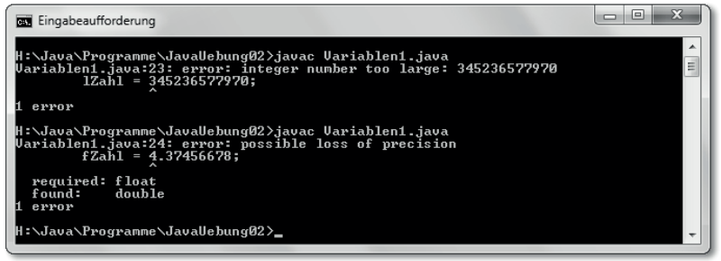

Abbildung 2.5 Fehlermeldung beim zweiten Übersetzungsversuch

Kommazahlen im Quellcode werden standardmäßig als `double`-Werte interpretiert. Der Zahlenwert soll aber einer `float`-Variablen zugewiesen werden. Sie ahnen es wahrscheinlich schon: Der Zahlenwert muss durch Anhängen des Buchstabens `f` oder `F` ausdrücklich als `float`-Typ kenntlich gemacht werden. Nach der Korrektur

```
fZahl = 4.37456678f;
```

ist der Übersetzungsvorgang erfolgreich, und das Programm sollte die in Abbildung 2.6 dargestellte Ausgabe zeigen.

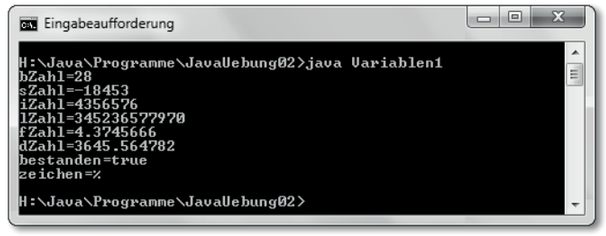

Abbildung 2.6 Ausgabe von Aufgabe 1

2.3.5 Häufiger Fehler bei der Variablendeklaration

Abbildung 2.6 zeigt die Ausgabe der `println`-Anweisungen aus Listing 2.2. Einen Fehler, den Programmieranfänger häufig begehen, möchte ich an dieser Stelle ansprechen. Der Quellcode ist fehlerbereinigt, denn der Compiler erzeugt keine Fehlermeldungen. Die Ausgabeanweisungen werden ausgeführt und zeigen die Variablenwerte an. Programmieranfänger geben sich mit diesen Überprüfungen zufrieden und sehen die Aufgabe als gelöst an. Das Testen eines als fertig angesehenen Programms ist eine der aufwendigsten Aufgaben beim Programmieren. Hier ist sehr große Sorgfalt geboten, d. h., dass die Programmergebnisse sehr genau überprüft werden müssen. In unserem Beispiel, in dem keinerlei Eingaben des Anwenders erfolgen, ist das noch relativ einfach. Ein genauer Blick auf die ausgegebenen Werte zeigt aber auch hier, wie leicht Fehler übersehen werden.

Überprüfen Sie die Ausgabe der Variable `fzahl`, indem Sie den ausgegebenen Wert mit dem zugewiesenen Wert vergleichen. Es wird offensichtlich ein etwas anderer Wert ausgegeben. Wo liegt die Ursache für diese Abweichung? Der zugewiesene Wert umfasst acht Nachkommastellen. In Tabelle 2.2 sind als Genauigkeit für `float`-Werte – und als solchen haben wir den Zahlenwert gekennzeichnet – sieben Nachkommastellen angegeben. Der Compiler war gezwungen, den Wert so anzupassen, dass er in den Speicherplatz passt, der für eine `float`-Variable zur Verfügung steht. Bei dieser Anpassung wird aber nicht ab- oder aufgerundet, sondern es entsteht ein abweichender Wert, der nur schwer vorhersehbar ist. Solche Verfälschungen kommen immer dann vor, wenn Zahlenwerte in Variablen gespeichert werden, für die sie eigentlich zu groß sind. Sie sollten deshalb die gültigen Wertebereiche für die gewählten Datentypen im Auge behalten.

Ich empfehle Ihnen, für Zahlenwerte die Standardtypen `int` und `double` zu verwenden. Sie sind für die meisten Anwendungen ausreichend groß bemessen, und das Argument, dass man mit den Datentypen `byte` und `short` bzw. `float` für Kommawerte Speicherplatz einsparen kann, spielt bei den heute zur Verfügung stehenden Speichergrößen kaum noch eine Rolle. Wenn Sie noch einmal einen Blick auf die Fehlermel-

dung in Abbildung 2.5 werfen, werden Sie feststellen, dass die Fehlermeldung des Compilers sehr präzise auf dieses Problem aufmerksam gemacht hat. Er hat dort gemeldet: `possible loss of precision` (möglicherweise droht ein Verlust an Genauigkeit). Es lohnt sich also, bei jeder Fehlermeldung genau hinzuschauen, was der Compiler meldet.

2.3.6 Praxisbeispiel 2 zu Variablen

Wir erstellen im Projekt *JavaUebung02* eine Klasse mit dem Namen `Variablen2`. In der `main`-Methode dieser Klasse sollen fünf Zeichen-Variablen mit den Namen `z1`, `z2`, `z3`, `z4` und `z5` deklariert werden. Die Variablen sollen mit den folgenden Werten der Tabelle 2.5 initialisiert und dann ausgegeben werden.

Variable	Wert
z1	a
z2	b
z3	A
z4	©
z5	¾

Tabelle 2.5 Variablen und ihre Werte in Aufgabe 2

Die ersten drei Werte dürften keine Probleme verursachen, denn die Zeichen sind direkt über die Tastatur erreichbar, und Sie können sie so eingeben:

```
char z1, z2, z3, z4, z5;
// Wertzuweisungen
z1 = 'a';
z2 = 'b';
z3 = 'A';
```

Listing 2.3 Wertzuweisungen bei »char«-Variablen

Wie aber erreichen Sie die letzten beiden Zeichen? Beachten Sie, dass das letzte Zeichen tatsächlich als ein Zeichen zu verstehen ist, auch wenn Sie zunächst meinen könnten, dass es sich um drei Zeichen (`3`, `/` und `4`) handelt.

Um das Problem zu lösen, müssen Sie auf *Unicode* zurückgreifen. Grundsätzlich müssen Zeichen, die am Bildschirm dargestellt oder von einem anderen Gerät, wie z. B.

einem Drucker, ausgegeben werden sollen, digital codiert werden. Diese Codierung besteht darin, dass jedem Zeichen, das dargestellt werden soll, ein Zahlenwert zugeordnet wird. Diese Zahl als Dualzahl entspricht dann der digitalen Darstellung des Zeichens. Neben dem ASCII-Code, dem ANSI-Code und vielen weiteren beschreibt der Unicode eine mögliche Codierung. Da nun Java Unicode verwendet, sollten wir uns diesen Code etwas genauer anschauen. Im Internet finden Sie eine Vielzahl von Seiten, die den Unicode in Tabellenform darstellen.

Unter der Adresse *http://www.utf8-zeichentabelle.de* finden Sie u. a. die Darstellung aus Tabelle 2.6.

Unicode-Codeposition	Zeichen	Name
U+000A		<control> Steuerzeichen Zeilenwechsel (New Line)
U+000D		<control> Steuerzeichen Wagenrücklauf
U+0020		SPACE
U+0041	A	LATIN CAPITAL LETTER A
U+0042	B	LATIN CAPITAL LETTER B
U+0043	C	LATIN CAPITAL LETTER C
U+0044	D	LATIN CAPITAL LETTER D
U+0045	E	LATIN CAPITAL LETTER E
U+0046	F	LATIN CAPITAL LETTER F
U+00A9	©	COPYRIGHT SIGN
U+00AE	®	REGISTERED SIGN
U+00B2	²	SUPERSCRIPT TWO
U+00BD	½	VULGAR FRACTION ONE HALF
U+00BE	¾	VULGAR FRACTION THREE QUARTERS
U+00C4	Ä	LATIN CAPITAL LETTER A WITH DIAERESIS
U+00D6	Ö	LATIN CAPITAL LETTER O WITH DIAERESIS
U+00DC	Ü	LATIN CAPITAL LETTER U WITH DIAERESIS

Tabelle 2.6 Auszug aus dem Unicode

Unicode-Codeposition	Zeichen	Name
U+00DF	ß	LATIN SMALL LETTER SHARP S
U+00E4	ä	LATIN SMALL LETTER A WITH DIAERESIS
U+00F6	ö	LATIN SMALL LETTER A WITH DIAERESIS
U+00FC	ü	LATIN SMALL LETTER A WITH DIAERESIS

Tabelle 2.6 Auszug aus dem Unicode (Forts.)

Da Unicode mit über 110.000 Zeichen sehr umfangreich ist, ist in Tabelle 2.6 nur ein Auszug dargestellt Die Tabelle beschränkt sich auf die deutschen Umlaute und einige interessante Sonderzeichen. In der ersten Spalte steht die Nummer des jeweiligen Zeichens allerdings in hexadezimaler Schreibweise. Diese Schreibweise ist wesentlich übersichtlicher und kürzer und wird deshalb in der Computertechnik als Kurzform für Dualzahlen verwendet. Die Zeichen mit den Codes von 0 bis 31 sind Steuerzeichen, die in einem Text quasi unsichtbar sind. Als Beispiel sind die beiden Steuerzeichen mit den Codes 10 (U+000A) und 13 (U+000D) aufgeführt. Das Zeichen mit dem Code 10 entspricht z. B. einem Zeilenvorschub. Das Zeichen mit dem Code 32 (U+0020) entspricht der Leerstelle und erscheint in einem Text als Lücke zwischen zwei Wörtern. Wie hilft uns nun diese Tabelle bei der Lösung unseres Problems aus Aufgabe 2?

Auf der Computertastatur kann immer nur ein kleiner Teil des umfangreichen Zeichencodes untergebracht werden. Alle anderen Zeichen können Sie mithilfe des Zeichencodes ansprechen. Aus der Tabelle können Sie für das Zeichen © den Code 00A9 und für das Zeichen ¾ den Code 00BE entnehmen. In einer Wertzuweisung kann der Zeichenvariablen einfach der Zahlencode des betreffenden Zeichens zugewiesen werden. Sie können dabei die dezimale Schreibweise `z4 = 169;` (für ©) ebenso wie die hexadezimale Schreibweise `z4 = 0x00a9;` verwenden. Dem Java-Compiler wird durch `0x` kenntlich gemacht, dass die folgende Zeichenfolge als hexadezimale Zahl zu behandeln ist.

```
/* Programm zum Testen der Verwendung von Variablen
 * Datum: 2011-11-30
 * Hans-Peter Habelitz
 */

public class Variablen2 {
  public static void main(String[] args) {
```



```
    // Variablendeklarationen
    char z1, z2, z3, z4, z5;

    // Wertzuweisungen
    z1 = 'a';
    z2 = 'b';
    z3 = 'A';
    z4 = 169; // alternativ z4 = 0x00a9
    z5 = 190; // alternativ z5 = 0x00be

    // Ausgaben
    System.out.print("z1: ");
    System.out.println(z1);
    System.out.print("z2: ");
    System.out.println(z2);
    System.out.print("z3: ");
    System.out.println(z3);
    System.out.print("z4: ");
    System.out.println(z4);
    System.out.print("z5: ");
    System.out.println(z5);
  }
}
```

Listing 2.4 Quelltext zu Aufgabe 2

Das Programm wird Ihnen wahrscheinlich die Ausgabe aus Abbildung 1.7 liefern.

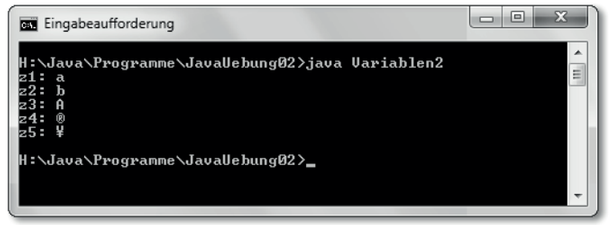

Abbildung 2.7 Ausgabe des Programms »Variablen2«

Sie werden feststellen, dass die letzten beiden Zeichen, die über den Zeichencode angesprochen wurden und Sonderzeichen anzeigen sollten, nicht die erwarteten Zeichen sind.

Wichtiger Hinweis für Windows-Anwender

Sollten Sie einmal unerwartete Zeichenausgaben feststellen, so kann das daran liegen, dass Ihr Betriebssystem nicht den passenden Zeichensatz verwendet. Sie sollten dann prüfen, welche Codepage Ihr System verwendet, und diese eventuell umstellen. Windows verwendet z. B. für die Eingabeaufforderung als Überbleibsel aus den frühen Tagen der Microsoft-Betriebssysteme noch eine Codepage, die Sonderzeichen anders als in der grafischen Oberfläche – und damit auch anders als im Unicode beschrieben – codiert. Mit dem Konsolenbefehl `chcp` (*change codepage*) ohne weitere Parameter können Sie die aktuell von der Eingabeaufforderung verwendete Codepage anzeigen lassen. Wahrscheinlich wird hierbei die Codepage 850 angezeigt. Zwei Umstellungen sind erforderlich, um die Eingabeaufforderung so einzustellen, dass sie den Unicode wie in anderen Umgebungen korrekt anzeigt.

Mit dem Konsolenbefehl `chcp 1252` stellen Sie zunächst die entsprechende Codepage ein (siehe Abbildung 2.8).

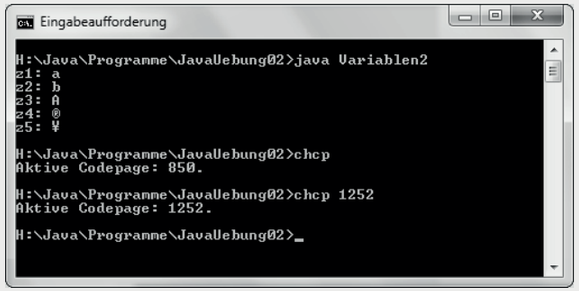

Abbildung 2.8 Umstellen der verwendeten Codepage

Öffnen Sie anschließend die EIGENSCHAFTEN der Eingabeaufforderung durch einen Rechtsklick auf die Titelleiste der Eingabeaufforderung (siehe Abbildung 2.9).

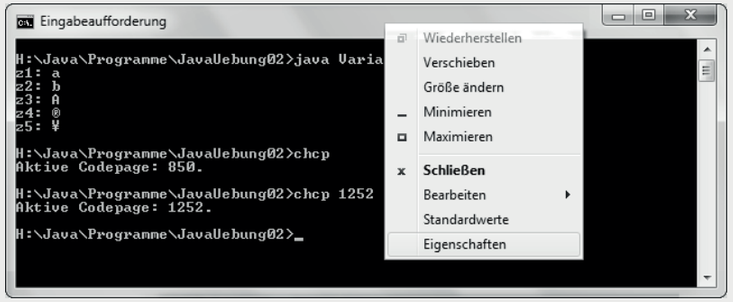

Abbildung 2.9 Kontextmenü zum Öffnen des »Eigenschaften«-Dialogs

Im Dialog in Abbildung 2.10 stellen Sie die Schriftart auf eine der Alternativen zur Rasterschrift (z. B. Lucida) um.

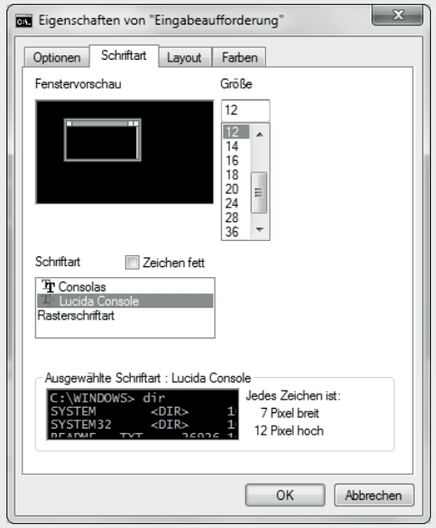

Abbildung 2.10 Ändern der Schriftart für die Eingabeaufforderung

Nach diesen Umstellungen verhält sich die Eingabeaufforderung so wie andere Umgebungen und zeigt auch die Unicodezeichen richtig an (siehe Abbildung 2.11).

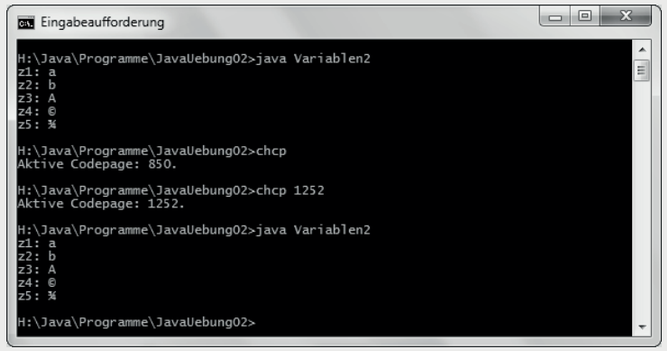

Abbildung 2.11 Ausgabe von »Variablen2« nach den Umstellungen

2.3.7 Der Datentyp »String«

Sie werden sich fragen, wieso der Datentyp `String` bei den Datentypen nicht angesprochen wurde. Der Datentyp `String` gehört nicht zu den primitiven Datentypen. Er gehört

zu den Objekttypen, die wesentlich mächtiger sind und deshalb nicht mit wenigen Worten erschöpfend behandelt werden können. Andererseits ist dieser Datentyp so elementar wichtig, dass man auch in einfachen Programmen kaum ohne ihn auskommt. An dieser Stelle soll der Datentyp `String` deshalb zumindest so weit erläutert werden, dass Sie ihn nutzen können. Eine ausführlichere Beschreibung wird in Kapitel 7, »Grundlegende Klassen«, folgen, sobald die Grundlagen zu Objekten behandelt sind.

Zum Speichern einzelner Zeichen stellt Java den primitiven Datentyp `char` zur Verfügung. Ein ganzes Wort oder sogar ein ganzer Satz bildet eine Zeichenkette. Um eine solche Zeichenkette in einer einzigen Variablen zu speichern, steht kein primitiver Datentyp zur Verfügung. Er kann in einer Variablen vom Datentyp `String` gespeichert werden. Konstante Zeichenketten (Literale) werden in Java zwischen Anführungszeichen gesetzt. Eine Stringvariable wird wie jede Variable eines primitiven Datentyps mit

```
String variablenname;
```

deklariert bzw. mit

```
String variablenname = "Das ist der Wert der Variablen";
```

mit der Deklaration initialisiert.

Erinnern Sie sich noch an unser erstes Programmbeispiel, das Hallo-Welt-Programm? Bereits dort haben wir den Datentyp `String` in Form eines Literals verwendet, als wir mit der Anweisung `System.out.println("Hallo Welt!")` eine Bildschirmausgabe in der Konsole erzeugt haben. Dies unterstreicht die Bedeutung dieses Datentyps.

Die Ausgabe von Text mit `System.out.print` oder `println` ist ein wichtiges Element für den Dialog zwischen Programm und Anwender. Das Programm zeigt dem Anwender so die Ergebnisse seiner Arbeit an oder es gibt dem Anwender Hinweise zu erforderlichen Eingabedaten.

2.3.8 Der Dialog mit dem Anwender

Programme stehen immer im Dialog mit dem Anwender – und wenn es sich dabei nur um die Ausgabe von Fehlermeldungen handelt. Nahezu jedes Programm arbeitet nach dem *EVA-Prinzip* (siehe Abbildung 2.12). Das Kürzel EVA steht dabei für Eingabe–Verarbeitung–Ausgabe. Es besagt, dass dem Programm zunächst über die Eingabe Daten zur Verfügung gestellt werden. Mit diesen Daten arbeitet das Programm in einer Verarbeitungsphase, um dann in der Ausgabe die berechneten Ergebnisse dem Anwender mitzuteilen.

Abbildung 2.12 Das EVA-Prinzip

Den Informationsfluss vom Programm zum Anwender haben wir bisher hauptsächlich mit `System.out.println` bzw. `System.out.print` über die Konsole realisiert. Für die umgekehrte Richtung, d. h. zur Eingabe von Informationen vom Anwender zum Programm, haben wir die Aufrufparameter verwendet (siehe Kapitel 1, Aufgabe 1, Projekt *JavaUebung01*, *Uebergabe.java*).

Für den Fall, dass nach dem Programmstart das Programm dazu auffordern soll, Daten einzugeben, haben wir die Methode `JOptionPane.showInputDialog` verwendet (siehe Kapitel 1, Aufgabe 6, Projekt *JavaUebung01*, *Kreisberechnung4*).

Wir verwenden bei diesem Programm zur Ausgabe nicht wie sonst die Konsole. Früher, als die Betriebssysteme noch keine grafischen Oberflächen verwendeten, waren Konsolenprogramme die einzige Möglichkeit, einen Dialog zwischen Anwendungsprogramm und Anwender zu realisieren. Heute sind die Anwender gewohnt, mit grafischen Oberflächen zu arbeiten. Der Vollständigkeit halber möchte ich Ihnen aber das Einlesen von Benutzereingaben als Konsolenanwendung nicht vorenthalten. Sie werden feststellen, dass die Variante mit dem `InputDialog` sogar noch einfacher ist als diese primitiver anmutende Version:

```
1:    import java.io.BufferedReader;
2:    import java.io.IOException;
3:    import java.io.InputStreamReader;
4:
5:    public class Kreisberechnung4Console {
6:      public static void main(String[] args) throws IOException {
7:        double radius, umfang, flaeche;
8:        String einheit, eingabe;
9:        BufferedReader eingabepuffer = new BufferedReader
                          (new InputStreamReader(System.in));
10:       System.out.print("Geben Sie den Kreisradius ein: ");
11:       eingabe = eingabepuffer.readLine();
12:       radius = Double.parseDouble(eingabe);
13:       System.out.print("Geben Sie die Einheit ein: ");
14:       eingabe = eingabepuffer.readLine();
15:       einheit = eingabe;
16:       umfang = 2.0 * 3.1415926 * radius;
17:       flaeche = 3.1415926 * radius * radius;
18:       System.out.print("Umfang: ");
```

```
19:         System.out.print(umfang);
20:         System.out.println(" " + einheit);
21:         System.out.print("Flaeche: ");
22:         System.out.print(flaeche);
23:         System.out.println(" " + einheit + '\u00b2');
24:       }
25:     }
```

Listing 2.5 »Kreisberechnung4« mit Tastatureingabe in der Konsole

Im Unterschied zu der Version aus Kapitel 1 werden mehrere `import`-Anweisungen (Zeile 1 bis 3) verwendet, damit anstelle der `JOptionPane`-Komponente die Komponenten `IOException`, `BufferedReader` und `StreamInputReader` zur Verfügung stehen. Zur Vorbereitung der Tastatureingabe wird in Zeile 9 als Zwischenspeicher eine zusätzliche Variable `eingabepuffer` vom Typ `BufferedReader` (er wird in Kapitel 11, »Dateien«, näher erläutert) angelegt und gleichzeitig mit der Standardeingabe `System.in` (normalerweise ist das die Tastatur) verbunden.

Nach diesen Vorarbeiten kann das eigentliche Einlesen der Tastatureingabe in Zeile 11 mit der Anweisung `eingabepuffer.readLine()` erfolgen. Dieser Aufruf liefert als Ergebnis eine Zeichenkette zurück, die der Variablen `eingabe` zugewiesen wird. Unmittelbar davor wird mit `System.out.print` eine Textzeile als Aufforderung ausgegeben. Ganz gleich über welche Methode Sie Tastatureingaben programmieren, werden die Eingaben als Zeichen bzw. Zeichenketten zurückgeliefert. Das bedeutet, dass in vielen Fällen, in denen es sich bei den Eingaben um Zahlenwerte handelt, mit denen anschließend gerechnet werden soll, diese Zeichenketten noch umgewandelt werden müssen. In unserem Beispiel soll als erste Eingabe der Kreisradius eingegeben werden. Die Zeichenkette wird in Zeile 12 mit der Anweisung `Double.parseDouble(eingabe)` umgewandelt und der Variablen `radius` zugewiesen.

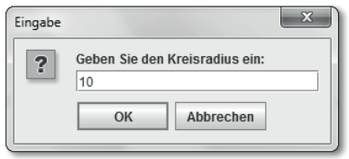

Abbildung 2.13 Eingabe mit »JOptionPane.showInputDialog«

Ein ganz wesentliches Vorhaben, das diesem Buch zugrunde liegt, besteht darin, Ihnen möglichst frühzeitig die Erstellung von grafisch orientierten Programmen zu ermöglichen. Deshalb möchte ich hier bereits auf die Verwendung der Konsole gänzlich verzichten und stattdessen das Programm aus Kapitel 1 so verändern, dass auch für die

Ausgabe der Ergebnisse aus dem Paket `javax.swing` die Klasse `JOptionPane` verwendet wird (siehe Abbildung 2.13). Die Methode `showMessageDialog` können Sie dazu verwenden, ein Meldungsfenster zur Ausgabe einer Information einzublenden.

```
1:   import javax.swing.JOptionPane;
2:
3:   public class Kreisberechnung4JOptionPane {
4:     public static void main(String[] args) {
5:       double radius, umfang, flaeche;
6:       String einheit, eingabe;
7:       eingabe = JOptionPane.showInputDialog(
                          "Geben Sie den Kreisradius ein: ");
8:       radius = Double.parseDouble(eingabe);
9:       eingabe = JOptionPane.showInputDialog(
                            "Geben Sie die Einheit ein: ");
10:       einheit = eingabe;
11:       umfang = 2.0 * 3.1415926 * radius;
12:       flaeche = 3.1415926 * radius * radius;
13:       JOptionPane.showMessageDialog(
                  null,"Umfang: " + umfang + " "
                  + einheit +  "\nFläche: " + flaeche + " "
                  + einheit + '\u00b2');
14:     }
15:   }
```

Listing 2.6 »Kreisberechnung4« ohne Konsole

Die Methode `showMessageDialog` erwartet im Unterschied zu `showInputDialog` zwei durch Komma getrennte Werte. Der erste Wert wird erst in komplexeren Programmen relevant, die zur gleichen Zeit mehrere Programmfenster darstellen. Mit diesem Parameter können Sie den Messagedialog dann einem anderen Fenster unterordnen. Wird wie hier eine solche Unterordnung nicht benötigt, darf der Parameter aber nicht einfach wegfallen. Stattdessen wird der vordefinierte Wert `null` angegeben.

Der zweite Wert muss eine Zeichenkette sein. Sie stellt den Text dar, der als Hinweis ausgegeben wird. Das Beispiel zeigt sehr anschaulich, wie diese Zeichenkette mit dem +-Operator aus mehreren Teilen zusammengesetzt werden kann. Beachten Sie, dass Zeichenkettenliterale in doppelte Anführungszeichen gesetzt werden, einzelne Zeichen dagegen werden zwischen einfache Hochkommata gesetzt. Das Zeichen `'\n'` steht für den Zeilenvorschub und `'\u00b2'` für die hochgestellte 2. Das Ergebnis dieser Bemühungen sehen Sie in Abbildung 2.14.

Abbildung 2.14 Ausgabe mit »JOptionPane.showMessageDialog«

In Bezug auf die hochgestellte 2 ist Ihnen als aufmerksamem Leser vielleicht eine Diskrepanz zwischen der Version aus Kapitel 1 und der hier geänderten Version aufgefallen. In Kapitel 1 wurde für die hochgestellte 2 das Zeichenliteral `'\u00fd'` verwendet. Für den Programmieranfänger wird diese Diskrepanz oft zu einem Stolperstein. Dabei gibt es eine recht einfache Erklärung dafür. Sie haben zu Beginn dieses Kapitels erfahren, dass Java den Unicode verwendet und deshalb eine sehr große Zahl unterschiedlicher Zeichen darstellen kann. Genau genommen muss man sagen, dass Java eine sehr große Zahl unterschiedlicher Zeichen codieren kann. Für die Darstellung ist aber die Umgebung verantwortlich, auf der das Java-Programm ausgeführt wird (siehe Abschnitt 2.3.6, »Praxisbeispiel 2 zu Variablen«). Gespeichert wird das Zeichen immer als Zahlenwert.

In Kapitel 1 wurde das Programm in der Eingabeaufforderung von Windows gestartet. Die Eingabeaufforderung verwendet zum Codieren und entsprechend auch zum Decodieren standardmäßig den erweiterten ASCII-Code. Die Codes der Standardzeichen sind im erweiterten ASCII-Code identisch mit den Codes im Unicode. Bei den Sonderzeichen – und dazu gehört neben den deutschen Umlauten auch die hochgestellte 2 – gibt es zwischen diesen beiden Codetabellen aber Abweichungen. Aus diesem Grund wurde in Kapitel 1 die Codierung der hochgestellten 2 aus der ASCII-Code-Tabelle entnommen.

Wenn Sie Programme in einer Entwicklungsumgebung wie Eclipse starten, die eine eigene Konsolendarstellung in einem Fenster verwendet, dann wird zur Decodierung von Zeichen unter Windows der ANSI-Code verwendet. Dieser Code entspricht auch bei den Sonderzeichen der Darstellung in Unicode. Deshalb konnten wir hier den Code für die hochgestellte 2 aus der Unicode-Tabelle entnehmen. Da auch die deutschen Umlaute der Darstellung in Unicode entsprechen, können auch diese viel unproblematischer verwendet werden.

2.3.9 Übungsaufgaben

An dieser Stelle sollen Sie noch ein bisschen üben, um ein besseres Verständnis für die Verwendung von Variablen zu entwickeln.

Aufgabe 1

Sind die folgenden Deklarationen korrekt und sinnvoll gewählt?

1. `int zahl_der_versuche;`
2. `char z1, z2, z3;`
3. `boolean ist_verheiratet;`
4. `float postleitzahl;`
5. `long kantenlaenge;`
6. `short byte;`
7. `int nummer, anzahl;`
8. `long telefonnummer; hausnummer;`
9. `nummer byte;`
10. `byte i, j;`
11. `boolean false;`
12. `double gehalt, abzuege;`
13. `boolean rund;`
14. `short long;`
15. `long laenge, breite, hoehe;`
16. `pi double;`
17. `char buchstabe, ziffer;`
18. `int summe/anzahl;`
19. `gebraucht boolean;`
20. `long zaehler, durchschnitt;`

Aufgabe 2

Sind die folgenden Wertzuweisungen richtig und sinnvoll? Geben Sie bei Fehlern eine Fehlerbeschreibung an!

1. `int zahl_der_versuche = 15;`
2. `double gehalt = 2645.34€;`
3. `int hausnummer = 24;`
4. `char 'A' = buchstabe;`
5. `byte b = 324;`
6. `short z = 15;`
7. `boolean ist_verheiratet = false;`
8. `double laenge = breite = hoehe;`
9. `long postleitzahl = 02365;`

```
10. float umfang = 64537.34756;
11. long zahl = –23456786;
12. double telefonnummer = 0176.46578675;
13. boolean true = ist_gerade_zahl;
14. short i = 31556;
15. char zeichen = '\u00B1';
16. byte x = –112;
17. char zeichen = 174;
18. long 385799 = lange_zahl;
19. float 1.zahl = 4567.23545f;
20. double verlust = 34567,45;
21. double zahl1 = –1.7e7;
22. char zeichen = '\t';
23. char trenner = '\x2f';
24. float m = .3f;
25. char hk = '\'';
26. double wert = -.e;
27. short zahl13 = 13f;
28. double zahl12 = 24;
```

Die Lösungen zu den Aufgaben 1 und 2 finden Sie in Anhang C, »Musterlösungen«.

2.4 Operatoren und Ausdrücke

Sie haben bereits einen Operator kennengelernt, ohne dass der Begriff *Operator* dafür verwendet wurde. Sie haben mit dem Operator = Variablen Werte zugewiesen. Die Wertzuweisung ist ein Beispiel für eine Operation, die in einem Programm ausgeführt wird. Für Operationen benötigen wir immer Operanden, mit denen eine Operation durchgeführt wird, und Operatoren, die angeben, welche Operation durchgeführt werden soll. Wir kennen z. B. arithmetische Operationen. Dabei dienen Zahlenwerte als Operanden und Rechenzeichen als Operatoren.

In Java gibt es eine Vielzahl von Operatoren. Die wichtigsten Operatoren sind die arithmetischen, logischen und Vergleichsoperatoren. Wie in der Mathematik können Sie mithilfe von Operatoren *Ausdrücke* bilden. Jeder Ausdruck hat einen Wert, der sich nach der Auswertung des Ausdrucks ergibt. Der Wert ergibt sich aus dem Typ der Operanden

und dem Operator, der auf die Operanden angewendet wird. Wenn in einem Ausdruck mehrere Operatoren vorkommen, legen Prioritäten die Reihenfolge für die Anwendung der Operatoren fest. Dies kennen Sie bereits aus der Mathematik, wenn in einem arithmetischen Ausdruck mehrere Rechenoperationen vorzunehmen sind. Es gilt dann z. B. die Regel, dass die Punktrechnungen vor den Strichrechnungen auszuführen sind.

2.4.1 Zuweisungsoperator und Cast-Operator

Bei der *einfachen Zuweisung* (=) wird der rechts stehende Ausdruck ausgewertet, und das Ergebnis wird der links stehenden Variablen zugewiesen. Dabei müssen Sie darauf achten, dass der Typ des rechten Ausdrucks mit dem Typ der links stehenden Variablen kompatibel ist. Das heißt, dass die Typen identisch sein müssen oder aber der Typ des rechts stehenden Ausdrucks muss in den Typ der links stehenden Variablen umgewandelt werden können. Umwandlungen von einem »kleinen« in einen »größeren« Datentyp erfolgen automatisch, umgekehrt gilt das nicht. Umwandlungen von einem »größeren« Datentyp in einen »kleinen« Datentyp müssen explizit erfolgen. Die Größe eines Datentyps können Sie an dem von ihm benötigten Speicherplatz erkennen (siehe Tabelle 2.2).

Beispiel:

```
byte byteZahl;
int intZahl;
float floatZahl;
double doubleZahl;
```

Nach diesen Deklarationen sind folgende Wertzuweisungen möglich:

```
byteZahl = 100;          // keine Umwandlung erforderlich
intZahl = byteZahl;      // Umwandlung von byte nach int
floatZahl = intZahl;     // Umwandlung von int nach float
floatZahl = 23.345f;     // keine Umwandlung erforderlich
doubleZahl = floatZahl;  // Umwandlung von float nach double
```

Folgende Zuweisungen sind nicht möglich:

```
byteZahl = intZahl;
floatZahl = doubleZahl;
```

Sie erhalten bei diesen Zuweisungsversuchen den Fehlerhinweis »Type mismatch – cannot convert from int to byte« bzw. »from double to float«. Ist eine Umwandlung mög-

lich, wird sie jeweils automatisch durchgeführt. Man nennt diese automatische Umwandlung auch *implizite Typumwandlung*.

Operator	Bedeutung	Priorität
=	einfache Zuweisung	13

Tabelle 2.7 Zuweisungsoperatoren

Neben der quasi automatisch ablaufenden impliziten Typumwandlung besteht auch die Möglichkeit, Umwandlungen zu erzwingen. Eine »erzwungene« Typumwandlung nennt man *explizite Typumwandlung*. Für eine solche Typumwandlung wird der Cast-Operator eingesetzt. Der Ausdruck `(type)a` wandelt den Ausdruck `a` in einen Ausdruck des Typs `type` um. Auch hierbei handelt es sich nicht um eine Wertzuweisung. Das bedeutet, dass `a` selbst dabei nicht verändert wird.

Mithilfe des Cast-Operators können Sie durchaus auch »größere« in »kleinere« Datentypen umwandeln. Logischerweise gehen dabei in der Regel aber Informationen verloren. So wird z. B. beim Umwandeln eines `double` in einen `int` der Nachkommateil abgeschnitten (nicht gerundet). Beim Umwandeln eines `short`-Ausdrucks in einen `byte`-Ausdruck wird ein Byte abgeschnitten. Das bedeutet, dass ein Teil verloren geht, weil für ihn in dem neuen Datentyp nicht genügend Speicherplatz zur Verfügung steht. Dabei wird der zu speichernde Wert unter Umständen so verfälscht, dass nur schwer nachzuvollziehende Fehler entstehen.

Beispiel:

```
double x = 3.89;
int y;
y = (int) x; // y wird der Wert 3 zugewiesen
```

So kann der `int`-Variablen `y` der Wert der `double`-Variablen `x` zugewiesen werden. Wie bereits erläutert, gehen dabei die Nachkommastellen verloren.

2.4.2 Vergleiche und Bedingungen

Relationale Operatoren vergleichen Ausdrücke anhand ihrer numerischen Werte miteinander. Als Ergebnis liefert ein solcher Vergleich einen Wert vom Typ `boolean`. Vergleichsoperatoren werden vorwiegend zur Formulierung von Bedingungen verwendet. Von solchen Bedingungen können Sie z. B. die Ausführung von Anweisungen abhängig machen.

Operator	Bedeutung	Priorität
<	kleiner	5
<=	kleiner oder gleich	5
>	größer	5
>=	größer oder gleich	5
==	gleich	6
!=	ungleich	6

Tabelle 2.8 Vergleichsoperatoren

Fließkommazahlen sollten Sie nicht auf exakte Gleichheit oder Ungleichheit hin überprüfen, da Rundungsfehler oftmals eine exakte Gleichheit verhindern. Stattdessen sollten Sie mit den Operatoren < oder > auf eine bestimmte Fehlertoleranz hin prüfen.

Beispiel:

```
boolean test;
test = (2.05-0.05) == 2.0;
```

Man sollte erwarten, dass der Klammerausdruck den Wert 2.0 ergibt. Der Vergleich des Klammerausdrucks mithilfe des `==`-Operators auf Gleichheit sollte also `true` ergeben. Testen Sie das Resultat mit folgendem Quellcode:

```
public static void main(String[] args) {
  double a = 2.05;
  double b = 0.05;
  System.out.println(a);
  System.out.println(b);
  System.out.println(a-b);
  boolean test;
  test = (2.05-0.05) == 2.0;
  System.out.println(test);
  System.out.println(2.05-0.05);
  System.out.println(2.0);
}
```

Listing 2.7 Rundungsfehler beim Rechnen mit Fließkommawerten

Sie erhalten die in Abbildung 2.15 angezeigte Ausgabe in der Konsole.

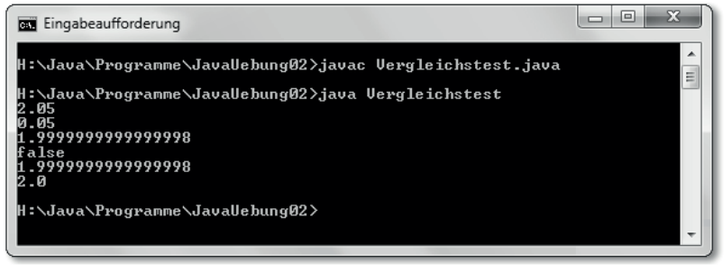

Abbildung 2.15 Rundungsfehler beim Rechnen mit Fließkommazahlen

Solche Rundungsfehler sind keine Seltenheit. Sie sollten deshalb immer daran denken, dass solche Fehler beim Rechnen mit Fließkommazahlen auftreten können. Nicht ohne Grund wird für diese Datentypen immer eine maximale Genauigkeit angegeben.

2.4.3 Arithmetische Operatoren

Die arithmetischen Operatoren haben numerische Operanden und liefern auch numerische Ergebnisse. Werden unterschiedliche Datentypen mit arithmetischen Operanden verknüpft, so erfolgt eine automatische Typumwandlung. Dabei wird grundsätzlich der kleinere Typ in den größeren Typ umgewandelt. Für die Größe des Datentyps ist der benötigte Speicherplatz entscheidend. Der Ergebnistyp entspricht dann immer dem größeren Typ. Tabelle 2.9 zeigt die in Java verfügbaren arithmetischen Operatoren.

Operator	Bedeutung	Priorität
+	positives Vorzeichen	1
-	negatives Vorzeichen	1
++	Inkrementierung	1
--	Dekrementierung	1
*	Multiplikation	2
/	Division	2
%	Modulo (Rest)	2
+	Addition	3
-	Subtraktion	3

Tabelle 2.9 Arithmetische Operatoren von Java

Hinweis

Bei der Verknüpfung zweier Ganzzahlen ist auch das Ergebnis ganzzahlig. Bei der Division ist dabei zu beachten, dass der Nachkommateil abgeschnitten wird. Es erfolgt keine Rundung des Ergebnisses. Möchten Sie als Ergebnis den tatsächlichen Kommawert haben, so müssen Sie dafür sorgen, dass zumindest einer der Operanden eine Kommazahl ist. Man schreibt z. B. statt `8/3` (das Ergebnis hätte den ganzzahligen Wert 2) dann `8./3` oder `8/3.`, damit das Ergebnis zu einem Kommawert wird.

Der Inkrement- und der Dekrement-Operator können nur auf Variablen angewendet werden. Sie erhöhen (inkrementieren) bzw. verringern (dekrementieren) den Wert einer Variablen um eins. Man unterscheidet hierbei die Postfix- und die Präfixform. Bei der Postfixform steht der Operator hinter der Variablen, bei der Präfixform steht er vor der Variablen. Der Unterschied zwischen beiden wird nur relevant, wenn der Operator innerhalb eines Ausdrucks verwendet wird. Beim Postfix wird die Variable erst nach dem Zugriff in- bzw. dekrementiert. Beim Präfix wird bereits vor dem Zugriff herauf- bzw. heruntergezählt. Dieser Sachverhalt wird an einem Beispiel verdeutlicht:

```
int a = 5;
System.out.println(a++);
System.out.print(a);
```

Hier wird das Inkrement von `a` als Postfix innerhalb der Ausgabeanweisung verwendet. Deshalb greift der Ausgabebefehl noch auf das nicht inkrementierte `a` zu und gibt den Wert 5 aus. Unmittelbar nach dem Zugriff durch `System.out.println` wird `a` dann um 1 erhöht. Dadurch wird beim nächsten Ausgabebefehl der Wert 6 ausgegeben.

```
int a = 5;
System.out.println(++a);
System.out.print(a);
```

Wählen Sie den Inkrementoperator als Präfix, so wird bereits vor dem ersten Zugriff mit der `print`-Anweisung die Erhöhung vorgenommen, und Sie erhalten jedes Mal den Wert 6 als Ausgabe.

Der Modulo-Operator `%` berechnet den Rest, der bei einer Division entsteht. Im Allgemeinen wird der Operator bei ganzzahligen Operatoren verwendet. So liefert `18% 5` als Ergebnis 3, denn teilt man 18 ganzzahlig durch 5, so bleibt ein Rest von 3. Der Operator kann in Java auch auf Kommazahlen angewendet werden. Damit liefert `12.6% 2.5` als Ergebnis `0.1`.

Ich möchte Sie an dieser Stelle noch auf eine Besonderheit des +-Operators hinweisen. Sie besteht darin, dass der +-Operator auch Texte als Operanden akzeptiert. Als Ergebnis entsteht dabei immer ein neuer Text. Werden zwei Texte mit dem +-Operator verknüpft, wird als Ergebnis ein Text geliefert, der aus den beiden aneinandergehängten Texten besteht. Wird der +-Operator zur Verknüpfung zweier Zahlenwerte verwendet, so bezeichnen wir die Operation als Addition. Die Verknüpfung zweier Texte mit dem +-Operator kann nicht als Addition bezeichnet werden, da sie keinen numerischen Wert liefert. Sie wird stattdessen als *Konkatenation* (Verkettung) bezeichnet.

Im folgenden Beispiel werden die beiden Variablen `nachname` und `vorname` zu einer einzigen Zeichenkette verkettet, die dann mit `System.out.println` in der Konsole ausgegeben wird:

```
String nachname = "Habelitz";
String vorname = "Hans-Peter";
System.out.println(vorname + " " + nachname);
```

Listing 2.8 Verketten von Strings

Das Beispiel zeigt, dass die Konkatenation wie die arithmetische Addition beliebig oft hintereinandergeschaltet werden kann. Hier wird das Stringliteral, das nur aus einem Leerzeichen besteht, als Trennzeichen zwischen Vor- und Nachname gesetzt.

Wird ein Text mit einem numerischen Wert verknüpft, dann wandelt der Compiler den numerischen Wert in einen Textwert um und setzt dann die beiden Texte zum Ergebnistext zusammen.

Beispiel:

```
int a = 2;
System.out.println("Die Variable a hat den Wert " + a);
```

Die `println`-Anweisung gibt den folgenden Text aus:

```
Die Variable a hat den Wert 2.
```

2.4.4 Priorität

Bildet man Ausdrücke mit mehreren Operatoren, so bestimmt die Priorität die Reihenfolge, in der die Operatoren angewendet werden. Die Prioritäten entsprechen der Rangfolge, die von der Mathematik her bekannt ist. Mithilfe von runden Klammern kann die Reihenfolge der Auswertung wie in der Mathematik beliebig verändert werden. Die Klammern können dabei beliebig tief geschachtelt werden.


```
int a = 2;
int b = 3;
int c = 5;
int ergebnis = a+b*c;
System.out.print("a+b*c=");
System.out.println(ergebnis);     // liefert 17
ergebnis = (a+b)*c;
System.out.print("(a+b)*c=");
System.out.println(ergebnis);     // liefert 25
```

Listing 2.9 Klammern in Ausdrücken

Das Listing 2.9 kann kürzer formuliert werden, wenn Sie eine Ausgabezeile mit einer einzigen `System.out.println` erzeugen:

```
int a = 2;
int b = 3;
int c = 5;
int ergebnis = a+b*c;
System.out.println("a+b*c=" + ergebnis);
ergebnis = (a+b)*c;
System.out.print("(a+b)*c=" + ergebnis);
```

Listing 2.10 Text und Zahlenwert wurden mit »+« verknüpft.

Sie können den Quellcode noch weiter verkürzen, indem Sie die Berechnung auch noch in die `System.out.println`-Anweisung integrieren.

Aber Achtung! Komplexe Ausdrücke bergen die Gefahr, dass man den Überblick über die Art und Weise verliert, wie der Compiler Ausdrücke auswertet. Testen Sie folgenden Quellcode:

```
int a = 2;
int b = 3;
int c = 5;
System.out.println("a+b*c = " + a + b * c);
System.out.println("(a+b)*c = " + (a + b) * c);
```

Listing 2.11 Fehlerhafter Verkürzungsversuch

Das Programm liefert die in Abbildung 2.16 gezeigte Ausgabe.

Abbildung 2.16 Fehlerhafte Ergebnisausgabe

Das Programm scheint falsch zu rechnen! Weshalb liefert die erste Berechnung nicht den Wert 17? Die Antwort gibt ein genaues Nachvollziehen der Vorgehensweise des Compilers. Alle Informationen, die Sie dazu brauchen, haben Sie in diesem Kapitel erhalten. Die Frage ist, wie wird der folgende Ausdruck vom Compiler ausgewertet?

```
"a+b*c = " + a + b * c
```

Der Ausdruck enthält drei Operatoren. Ein Blick auf die Prioritäten in Tabelle 2.9 bestätigt, dass wie in der Mathematik die Multiplikation (Priorität 2) vor der Addition (Priorität 3) auszuführen ist. Es gilt: Je kleiner der Zahlenwert der Priorität ist, desto höher ist die Priorität der Operation. Zuerst wird also die Multiplikation `b*c` mit dem Ergebnis 15 ausgeführt. Bleiben noch zwei +-Operationen auszuführen. Da beide die gleiche Priorität haben, werden die Operationen von links beginnend ausgeführt. Zuerst wird entsprechend die Verknüpfung des Textes `"a+b*c ="` mit dem Zahlenwert der Variablen `a` (2) als Konkatenation vorgenommen. Dabei entsteht wie oben erläutert der Textwert `"a+b*c = 2"`, der mit dem Ergebnis der Multiplikation (15) verknüpft wird. Es wird also nochmals ein Text mit einem Zahlenwert verknüpft. Der Zahlenwert 15 wird in einen Text umgewandelt, und die Verknüpfung der beiden Textelemente `"a+b*c = 2"` und `"15"` liefert ganz konsequent als Ergebnis `"a+b*c = 215"`. Wir lösen das Problem dadurch, dass wir die gesamte numerische Berechnung in Klammern einschließen, damit auf jeden Fall zuerst die komplette numerische Berechnung erfolgt, bevor das Zusammensetzen des Ausgabetextes erfolgt:

```
int a = 2;
int b = 3;
int c = 5;
System.out.println("a+b*c = " + (a + b * c));
System.out.println("(a+b)*c = " + (a + b) * c);
```

Listing 2.12 Korrigierte Ergebnisausgabe mit Klammern

An diesem Beispiel sehen Sie, dass es für jedes auf den ersten Blick auch noch so merkwürdige Programmergebnis einen nachvollziehbaren Grund gibt.

2.4.5 Logische Operatoren

Logische Operatoren verknüpfen Wahrheitswerte miteinander. In Java stehen die Operatoren UND, ODER, NICHT und Exklusives ODER zur Verfügung.

Operator	Bedeutung	Priorität
`!`	NICHT	1
`&`	UND mit vollständiger Auswertung	7
`^`	Exklusives ODER (XOR)	8
`|`	ODER mit vollständiger Auswertung	9
`&&`	UND mit kurzer Auswertung	10
`||`	ODER mit kurzer Auswertung	11

Tabelle 2.10 Logische Operatoren

Der NICHT-Operator `!` kehrt den logischen Operanden ins Gegenteil um. Hat `a` den Wert `true`, so hat `!a` den Wert `false`. Hat `a` den Wert `false`, dann hat `!a` den Wert `true`.

Tabelle 2.11 zeigt die möglichen Verknüpfungen mit den Ergebnissen der übrigen Operatoren.

a	b	a & b a && b	a ^ b	a | b a || b
`true`	`true`	`true`	`false`	`true`
`true`	`false`	`false`	`true`	`true`
`false`	`true`	`false`	`true`	`true`
`false`	`false`	`false`	`false`	`false`

Tabelle 2.11 Logische Verknüpfungen zweier Wahrheitswerte

UND und ODER gibt es in zwei Varianten. Die *kurze Auswertung* (`&&` bzw. `||`) führt dazu, dass die Auswertung des Gesamtausdrucks abgebrochen wird, sobald das Ergebnis feststeht. Eine *vollständige Auswertung* (`&` bzw. `|`) bewirkt, dass grundsätzlich immer der gesamte Ausdruck abgearbeitet wird. Im folgenden Beispiel wird der Unterschied gezeigt:

```
int a = 1;
boolean x = true;
boolean y = false;
System.out.println(y && (++a == 2));
System.out.println("a = " + a);
```

Die verkürzte Auswertung der UND-Verknüpfung in der ersten `System.out.println`-Anweisung sorgt dafür, dass der Klammerausdruck nicht mehr ausgewertet werden muss. Da `y` den Wert `false` hat, kann das Ergebnis der Verknüpfung nur `false` lauten. Dadurch, dass der Klammerausdruck nicht ausgewertet wird, entfällt auch das Inkrementieren (`++`) von `a`. Die Variable `a` behält ihren Wert. Testen Sie die Anweisungsfolge mit dem `&`-Operator für die vollständige Auswertung, und Sie werden feststellen, dass `a` inkrementiert wird und entsprechend den Wert 2 annimmt.

2.4.6 Sonstige Operatoren

Sie haben bis hierher die wichtigsten Operatoren kennengelernt. Java stellt aber noch einige weitere Operatoren zur Verfügung, die in einigen Situationen sehr hilfreich sein können. Es handelt sich dabei um die *Bit-* und *Bedingungsoperatoren*. Da diese Operatoren seltener Anwendung finden und für den Einstieg in die Programmierung keine große Bedeutung haben, werden sie hier nicht weiter behandelt.

Einen Operator möchte ich Ihnen aber noch vorstellen, weil Sie in fremden Java-Programmen durchaus öfter darauf stoßen werden. In Abschnitt 2.4.1, »Zuweisungsoperator und Cast-Operator«, war von der einfachen Zuweisung die Rede. Diese Formulierung hat bereits angedeutet, dass es neben der einfachen Zuweisung noch eine andere gibt. Es handelt sich dabei um die *kombinierte Zuweisung*, die die Wertzuweisung mit einem arithmetischen Operator oder einem der hier nicht behandelten Bitoperatoren kombiniert. Zum Beispiel bedeutet `a += 1` das Gleiche wie `a = a + 1`, also wird `a` um 1 erhöht und als neuer Wert der Variablen `a` wieder zugewiesen. Kurz gesagt: `a` wird um 1 erhöht.

Operator	Bedeutung	Priorität
`op=`	Kombinierte Zuweisung; `op` steht für `*`, `/`, `%`, `+`, `–` oder einen Bitoperator	13

Tabelle 2.12 Kombinierte Zuweisung

`a op= b` entspricht der Schreibweise `a = a op b`. Dabei können Sie für `op` einen der in Tabelle 2.12 angegebenen arithmetischen oder bitweisen Operatoren einsetzen.

2.5 Übungsaufgaben

Für die folgenden Aufgaben wird vorausgesetzt, dass die folgenden Variablen deklariert wurden:

```
int a = 3;
int b = 5;
int c = 6;
double x = 1.5;
double y = 2.3;
int int_ergebnis;
double double_ergebnis;
```

Aufgabe 1

Welche Werte liefern die folgenden Ausdrücke rechts des `=`-Zeichens, und ist die Wertzuweisung möglich?

```
int_ergebnis = a * b + c;
int_ergebnis = c + a * b;
int_ergebnis = c - a * 3;
int_ergebnis = c / a;
int_ergebnis = c / b;
int_ergebnis = a + b / c;
double_ergebnis = c / b;
double_ergebnis = c + a / b;
double_ergebnis = x + y * b;
double_ergebnis = (x + y) * b;
double_ergebnis = y - x * b;
```

Aufgabe 2

Welche Ausgaben werden von folgendem Quellcode erzeugt?

```
System.out.println("b + c * 6 = " + b + c * 6);
System.out.println("b - c * 6 = " + b - c * 6);
System.out.println("(x * c - a) = " + (x * c - a));
System.out.println("x + c * 6 = " + x + c * 6);
System.out.println("y - c / a = " + (y - c / a));
System.out.println("b + a * x + y = " + b + a * x + y);
System.out.println("b + a * x * y = " + b + a * x * y);
System.out.println("b + a * x - y = " + b + a * x - y);
```

Aufgabe 3

Welche Ausgaben werden von folgendem Quellcode erzeugt?

```
System.out.println("a++: " + a++);
System.out.println("a: " + a);
System.out.println("++a: " + ++a);
System.out.println("a: " + a);
System.out.println("b + a--: " + b + a--);
System.out.println("a: " + a + " b: " + b);
System.out.println("b + a--: " + (b + a--));
System.out.println("a: " + a + " b: " + b);
System.out.println("b + --a: " + (b + --a));
System.out.println("a: " + a + " b: " + b);
System.out.println("a**: " + a**);
```

Aufgabe 4

Welche Ausgaben werden von folgendem Quellcode erzeugt?

```
System.out.println("c > b = " + c > b);
System.out.println("c > b = " + (c > b));
System.out.println("b < a = " + (b < a));
System.out.println("c == b = " + (c == b));
System.out.println("c > a < b = " + (c > a < b));
System.out.println("a = b = " + (a = b));
System.out.println("a = " + a + " b = " + b);
System.out.println("x > y = " + (x > y));
y = y + 0.1;
y = y + 0.1;
System.out.println("y == 2.5 = " + (y == 2.5));
System.out.println("y = " + y);
double z = 1.0;
z = z + 0.1;
z = z + 0.1;
System.out.println("z == 1.2 = " + (z == 1.2));
System.out.println("z = " + z);
```

Aufgabe 5

Welche Ausgaben werden durch folgende Ausgabebefehle erzeugt?


```
boolean b_wert;
b_wert = a == c;
System.out.println("a == b = " + (a == c));
System.out.println(b_wert);
System.out.println(!b_wert);
b_wert = a == b && c > b;
System.out.println("a == b && c > b = " + b_wert);
b_wert = b < c & a++ == 4;
System.out.println("b < c & a++ == 4 = " + b_wert);
b_wert = b < c & ++a == 5;
System.out.println("b < c & ++a == 5 = " + b_wert);
a = 3;
b_wert = b < c & ++a == 4;
System.out.println("b < c & ++a == 4 = " + b_wert);
a = 3;
b_wert = a > b && c++ == 6;
System.out.println("a > b && c++ == 6 = " + b_wert);
System.out.println("c = " + c);
b_wert = !y > x;
System.out.println("!y > x = " + !y > x);
b_wert = !(y > x);
System.out.println("!(y > x) = " + !(y > x));
b_wert = a > b & c++ == 6;
System.out.println("a > b & c++ == 6 = " + b_wert);
System.out.println("c = " + c);
c = 6;
b_wert = a < b || c++ == 6;
System.out.println("a < b || c++ == 6 = " + b_wert);
System.out.println("c = " + c);
b_wert = a < b | c++ == 6;
System.out.println("a < b | c++ == 6 = " + b_wert);
System.out.println("c = " + c);
c = 6;
b_wert = a > b | c++;
System.out.println("a > b | c++ = " + b_wert);
```

Die Musterlösungen zu den Aufgaben 1 bis 5 finden Sie in Anhang C, »Musterlösungen«.

2.6 Ausblick

In diesem Kapitel haben Sie wesentliche Sprachelemente von Java kennengelernt. Sie kennen die einfachen Datentypen und die Operatoren, die auf diese Datentypen angewendet werden können. Sie haben mit Ausdrücken in eigenen Programmen gearbeitet und wissen jetzt, wie Java diese Ausdrücke auswertet.

Im folgenden Kapitel werden Sie erfahren, welche Sprachmittel Java zur Verfügung stellt, um den Programmablauf zu steuern. Sie werden Kontrollstrukturen kennenlernen, mit deren Hilfe Sie dafür sorgen können, dass Programmteile nur unter bestimmten Bedingungen ausgeführt werden. Auch das mehrfache Wiederholen von Programmteilen ist ein wesentliches Instrument für die Erstellung leistungsfähiger Programme.

Sie werden darüber hinaus einiges über die Gültigkeitsbereiche von definierten Variablen und über mögliche Namenskonflikte erfahren.

Kapitel 5
Klassen und Objekte

Sich den Objekten in der Breite gleichstellen, heißt lernen;
die Objekte in ihrer Tiefe auffassen, heißt erfinden.
(Johann Wolfgang von Goethe, 1749–1832)

Ein wesentliches Merkmal der Programmiersprache Java ist ihre *Objektorientierung*. Auch andere moderne Programmiersprachen – wie Delphi, C++, C# oder Visual Basic – sind objektorientiert. Die Objektorientierung ist heute aus der Programmierung nicht mehr wegzudenken. Hier soll zunächst geklärt werden, wodurch sich dieses Merkmal auszeichnet.

5.1 Struktur von Java-Programmen

Im Zusammenhang mit der objektorientierten Programmierung haben Begriffe wie *Klasse*, *Objekt*, *Attribut*, *Methode*, *Vererbung* und *Interface* eine besondere Bedeutung.

5.1.1 Klassen

Die bisher verwendeten Datentypen `byte`, `short`, `int`, `long`, `float`, `double`, `char` und `boolean` sind in Java vordefiniert. Sie werden auch als *primitive Typen* bezeichnet. Sie erfordern einen sehr geringen Aufwand für Compiler und Interpreter und bringen damit Geschwindigkeitsvorteile. Sie repräsentieren einfache Werte (Zahlenwerte oder Zeichen) und benötigen nur wenig Speicherplatz. Deshalb wurde in Java nicht wie in einigen anderen objektorientierten Programmiersprachen (z. B. Smalltalk) komplett auf sie verzichtet. Klassen definieren neue Typen, die Sie als Programmierer komplett an die eigenen Bedürfnisse zuschneiden können. Sie sind wesentlich leistungsfähiger als primitive Typen, denn sie können nicht nur einen, sondern auch eine Vielzahl von Werten speichern, die ihren Zustand als Eigenschaften beschreiben. Zusätzlich können sie auf Botschaften reagieren und selbst aktiv werden.

Anmerkung

Neue Typen können auch mit älteren, nicht objektorientierten Programmiersprachen gebildet werden. Diese Typen beschränken sich dann aber darauf, mehrere, auch unterschiedliche primitive Typen zu einem größeren Verbund zusammenzufassen. Eine Klasse im Sinne der objektorientierten Programmierung geht weit darüber hinaus.

Wir wollen uns nicht lange mit grauer Theorie aufhalten, sondern definieren gleich mal eine eigene Klasse, und zwar einen Zahlentyp, den es in der Programmiersprache Java ebenso wenig wie in vielen anderen Programmiersprachen gibt. An diesem Beispiel werden wir dann einige Besonderheiten der Objektorientierung kennenlernen.

Der neue Zahlentyp, den wir erzeugen, soll einen Bruch darstellen. Ein Bruch besteht aus einem Zähler und einem Nenner, die beide ganzzahlig sind. Da Zähler und Nenner durch ihren Zahlenwert komplett beschrieben sind, können wir für die Erzeugung entsprechend auf den primitiven Datentyp `int` für die Bestandteile unseres neuen Typs zurückgreifen. Wie eine Variable müssen wir den neuen Typ mit einem eindeutigen Namen (Bezeichner) ausstatten. Es hat sich eingebürgert, als Typnamen englische Begriffe zu verwenden und den ersten Buchstaben immer großzuschreiben. Wir weichen von dieser Vereinbarung insofern ab, als wir statt des englischen Ausdrucks den deutschen Ausdruck »Bruch« verwenden.

Allgemein besteht eine Klassendefinition aus folgender Konstruktion:

```
class Bezeichner {
    ... Einzelheiten der Definition ...
}
```

Listing 5.1 Allgemeine Beschreibung einer Klassendefinition

Entsprechend sieht die Klassendefinition unseres Typs folgendermaßen aus:

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

Listing 5.2 Definition der Klasse »Bruch«

Merke

Eine Klasse beschreibt den Aufbau eines komplexen Datentyps. Eine Klasse wird durch *Eigenschaften* (Datenelemente oder Attribute) und ihre *Fähigkeiten* (Methoden) beschrieben.

5.1.2 Attribute

Unsere Klasse `Bruch` besteht aus zwei *Datenelementen*. Wie Sie in Listing 5.2 sehen, werden diese Datenelemente wie Variablen definiert. In unserem Fall sind beide Datenelemente vom primitiven Typ `int`.

Merke

Für Attribute können Sie sowohl primitive Datentypen als auch Klassen verwenden.

Attribute sind fester Bestandteil einer Klasse und werden deshalb innerhalb der Klassendefinition festgelegt. Die Schreibweise unterscheidet sich nicht von der Definition der bisher verwendeten lokalen Variablen. Die Attribute müssen innerhalb einer Klasse eindeutig benannt sein. Es gelten die bekannten Regeln für die Vergabe von Bezeichnern (siehe Abschnitt 2.1, »Bezeichner und Schlüsselwörter«). Es werden in der Regel kleingeschriebene englische Substantive verwendet.

Unsere Klasse `Bruch` kann als neuer Typ angesehen werden, der gleichberechtigt neben den primitiven Typen `short`, `int`, `double` usw. steht. Im Unterschied zu den primitiven Typen besteht `Bruch` aus mehreren Bestandteilen, die einzeln angesprochen werden können.

Typen, die mehrere Bestandteile haben, werden auch *Referenztypen* genannt. Ein großer Vorteil einer Klasse besteht darin, dass ihre Bestandteile untrennbar miteinander verbunden sind. In unserem Beispiel sind Zähler und Nenner in der Klasse `Bruch` zusammengefasst. Der Zugriff auf die einzelnen Bestandteile ist nur über die Klasse, zu der sie gehören, möglich.

5.1.3 Packages

Jedes Java-Programm ist selbst immer als eine Klasse realisiert. Wie in Abschnitt 1.3.2, »Wie sind Java-Programme aufgebaut?«, erläutert wurde, besteht das gesamte Programm im einfachsten Fall aus dieser einen Klasse. Dass eine einzige Klasse ausreicht, ist aber sehr selten der Fall. Normalerweise werden in einem Programm mehrere Klassen verwendet. In der Regel wird jede einzelne Klasse in einer eigenen Quellcodedatei

definiert. Daraus folgt, dass ein Java-Programm dann aus mehreren Quellcodedateien bestehen kann.

Merke

Packages dienen dazu, mehrere logisch zusammengehörige Klassen zusammenzufassen und damit die Verwaltung größerer Programme zu vereinfachen.

In einer einzelnen Quellcodedatei können zwar theoretisch mehrere Klassen definiert werden, aber spätestens beim Kompilieren erstellt der Java-Compiler für jede Klasse eine eigene Bytecodedatei. Es macht deshalb durchaus Sinn, bereits beim Erstellen des Quellcodes darauf zu achten, dass in einer Quellcodedatei auch nur eine Klasse definiert wird.

Klassen müssen nur innerhalb eines Packages eindeutige Namen haben. Gleichnamige Klassen in anderen Packages erzeugen keine Namenskonflikte. Das erleichtert die Namensgebung für Klassen sehr. Da in ein Programm mehrere Packages eingebunden werden können, schränkt diese Namensgleichheit die Verwendbarkeit aber keineswegs ein.

Wenn wir davon ausgehen, dass jede Klasse in einer Datei abgelegt wird und ein Package mehrere Klassen organisatorisch zusammenfasst, dann sind Packages sehr gut mit Ordnern innerhalb des Dateisystems vergleichbar. Sie werden auch entsprechend in gleichnamigen Ordnern angelegt. Sie können auch wie Ordner geschachtelt werden. Dadurch wird eine hierarchische Struktur mit Packages und Subpackages erzeugt (siehe Abbildung 5.1).

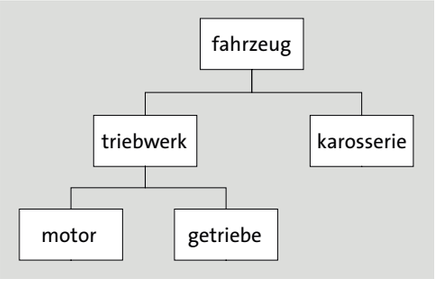

Abbildung 5.1 Beispiel für geschachtelte Packages

Zur Verdeutlichung erstellen wir als neues Projekt *JavaUebung05*. In diesem Projekt legen Sie als neue Klasse (File • New • Class) die Definition unserer Klasse `Bruch` an.

Solange Sie wie bisher das Textfeld Package leer lassen, weist Eclipse Sie darauf hin, dass es nicht gutgeheißen wird, das *default package* zu verwenden (siehe Abbildung 5.2).

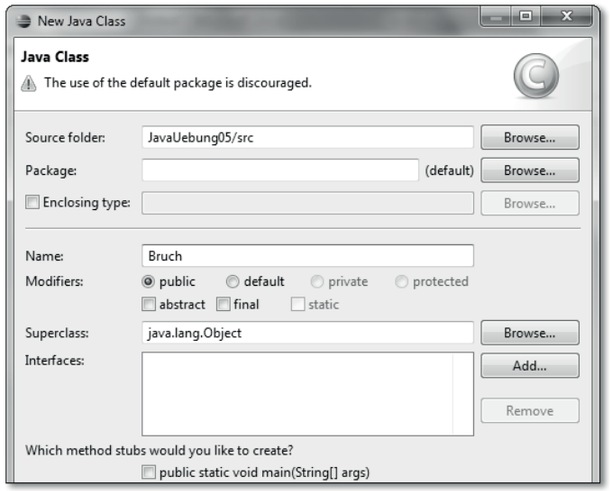

Abbildung 5.2 Erstellen der Klasse »Bruch«

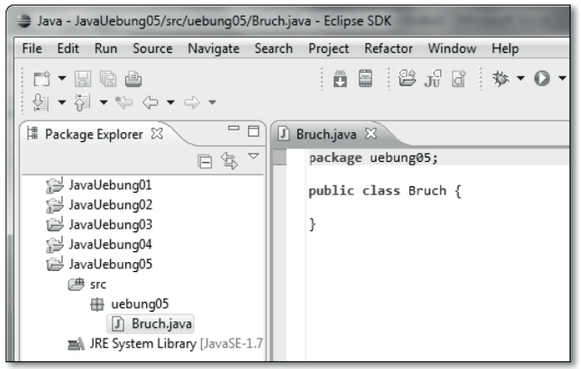

Abbildung 5.3 Projektansicht im Package Explorer

Sie sollten sich diesen Einwurf von Eclipse zu Herzen nehmen und für das Package einen Namen angeben. Auch diese Namenseingabe wird von Eclipse überwacht. Wenn Sie z. B. den Package-Namen mit einem Großbuchstaben beginnen, erscheint der Hinweis, dass Package-Namen mit einem Kleinbuchstaben beginnen sollten. Für die Bezeichnung von Packages gelten die gleichen Regeln wie für Variablen und Klassen. Nennen Sie das Package *uebung05*. Sie sollten diesmal auf die Erstellung der `main`-Methode verzichten, denn die Klasse `Bruch` soll kein eigenständiges Programm sein, sondern lediglich eine Klasse, die in einem Programm verwendet werden kann.

Wie Sie in Abbildung 5.3 sehen, wird die Quellcodedatei der Klasse `Bruch` im Package *uebung05* eingeordnet. Die im Quellcode erforderliche Anweisung `package uebung05;` wird von Eclipse automatisch eingetragen.

Ergänzen Sie den Quellcode um die Definition der Attribute:

```
package uebung05;
class Bruch {
    int zaehler;
    int nenner;
}
```

Listing 5.3 Definition der Klasse »Bruch«

Diese Quellcodedatei kann nicht als Anwendung gestartet werden. Was sollte sie auch ausführen? Beim Versuch, die Datei als Java-Anwendung zu starten, werden Sie feststellen, dass Eclipse im Menü RUN AS keine Option JAVA APPLICATION anbietet und dass beim Versuch, direkt mit RUN zu starten, nur eine Fehlermeldung erscheint (siehe Abbildung 5.4).

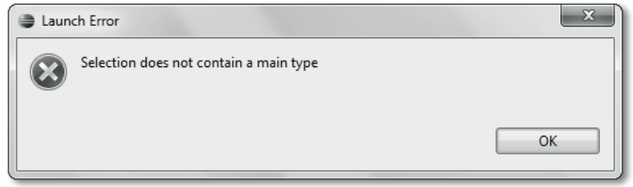

Abbildung 5.4 Fehlermeldung beim Versuch, die Klasse »Bruch« als Anwendung zu starten

Die Klasse `Bruch` selbst ist kein Programm, sondern eine Klasse, die in einem Programm verwendet werden kann. Man kann die Klasse `Bruch` mit einem Datentyp wie `int` vergleichen. Auch der Datentyp `int` ist keine Anwendung, die ausgeführt werden kann, sondern ein Element, das von einer Anwendung zur Speicherung von Informationen verwendet wird. Um die Klasse `Bruch` zu testen, müssen Sie eine Anwendung erstellen, die diese Klasse verwendet.

Innerhalb des Projekts *JavaUebung05* erstellen Sie eine neue Klasse mit dem Namen `Bruchtest1` (siehe Abbildung 5.5). Achten Sie dabei darauf, dass als Package-Name *uebung05* eingetragen ist. Diesen Eintrag nimmt Eclipse automatisch vor, wenn Sie im Package Explorer das Package selbst oder einen dem Package untergeordneten Eintrag markiert haben. Sie können den Eintrag aber auch von Hand vornehmen. Zusätzlich können Sie sich die Arbeit erleichtern, indem Sie unter der Frage WHICH METHOD STUBS WOULD YOU LIKE TO CREATE? das Häkchen vor dem Eintrag PUBLIC STATIC VOID

MAIN(STRING[] ARGS) setzen. Diese Methode macht eine Klasse zu einem ausführbaren Programm. Haben Sie die Absicht, eine als Programm ausführbare Klasse zu erstellen, dann müssen Sie diese `main`-Methode erstellen. Sie bildet den Startpunkt des Programmablaufs. Der Java-Interpreter sucht beim Aufruf in der ihm übergebenen Binärdatei nach der Methode mit dem Namen `main` und beginnt dort mit der Abarbeitung der Anweisungen.

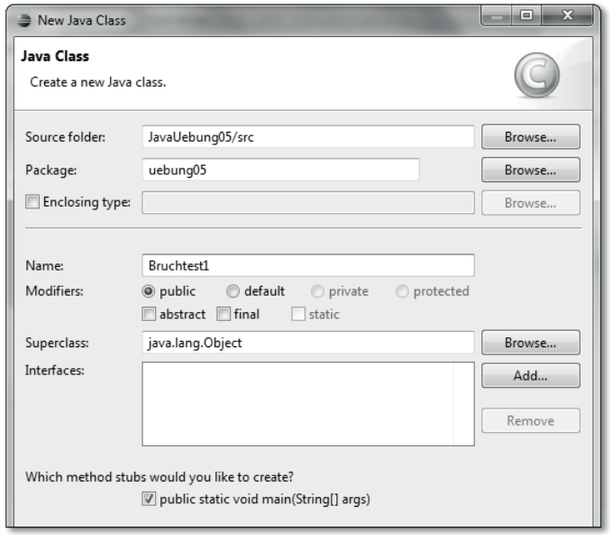

Abbildung 5.5 Erstellen der Anwendung »Bruchtest1«

Den von Eclipse erstellten Quellcode zeigt Abbildung 5.6.

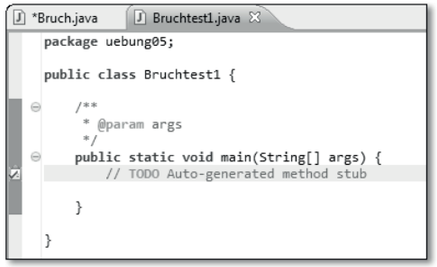

Abbildung 5.6 Von Eclipse erstellter Programmrahmen

Ersetzen Sie den markierten Kommentar, der Sie mit dem Hinweis TODO auffordert, hier die eigenen Ergänzungen vorzunehmen, durch die folgenden Anweisungen:


```
Bruch b = new Bruch();
b.zaehler = 3;
b.nenner = 4;
System.out.print("Bruch b = " + b.zaehler + "/" + b.nenner);
```

Es wird eine Variable vom Typ `Bruch` definiert. Eine solche Variable, die als Typ eine Klasse verwendet, wird *Objekt* genannt. Ihr wird der Wert ¾ zugewiesen, und schließlich wird die Variable zur Kontrolle mit `System.out.print` ausgegeben. Der folgende Abschnitt erläutert, wie sich die Verwendung einer Klasse von der Verwendung eines primitiven Datentyps unterscheidet.

5.2 Objekte

Verwenden Sie primitive Typen, so reicht die Definition einer Variablen von diesem Typ bereits aus, und Sie können diesen Variablen Daten zuweisen. Die Definition einer Klasse darf nicht mit der Definition einer Variablen verwechselt werden. Mit der Definition einer Klasse ist nur festgelegt, wie eine später zu definierende Variable aufgebaut ist. Man kann die Definition einer Klasse als Bauplan auffassen. Eine Variable, die nach diesem Plan anschließend angelegt wird, nennt man *Objekt*, *Instanz* oder *Exemplar* der Klasse.

Merke

Ein Objekt ist ein Exemplar (Instanz), das nach dem Bauplan einer Klassendefinition erstellt wurde. Die Klasse stellt den Bauplan dar (siehe Abbildung 5.7). Das Objekt ist eine Variable, die nach diesem Plan aufgebaut ist.

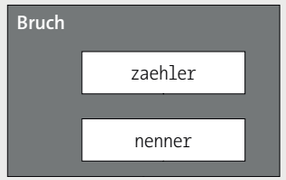

Abbildung 5.7 Bauplan der Klasse »Bruch«

Nach dem Bauplan der Klasse können beliebig viele Objekte (Instanzen) erzeugt werden. Eine Instanz ist mit einer Variablen eines primitiven Typs vergleichbar, weist aber in der Handhabung deutliche Unterschiede auf. Wie in Kapitel 2, »Grundbausteine eines Java-Programms«, beschrieben, wird z. B. mit

```
int zahl1;
```

eine Variable definiert. Damit ist im Arbeitsspeicher unmittelbar auch Speicherplatz verfügbar, auf den über den Bezeichner zugegriffen werden kann.

Die Verwendung einer Klasse stellt sich nicht ganz so einfach dar. Nach der Erstellung eines Bauplans durch die oben beschriebene Definition steht der neue Typ zur Verfügung. Damit kann eine Variable dieses Typs mit der folgenden Anweisung definiert werden:

```
Bruch b;
```

Abbildung 5.8 Definition einer Variablen vom Typ »Bruch«

Wie Abbildung 5.8 zeigt, ist mit dieser Anweisung nur ein Bezeichner definiert, der in der Lage ist, auf eine Instanz der Klasse `Bruch` zu verweisen. Man nennt sie deshalb auch *Referenzvariable*. Im Gegensatz zu den bisher verwendeten Typen ist damit aber im Hauptspeicher noch kein Platz für die einzelnen Attribute `zaehler` und `nenner` reserviert. Auch die Adresse, an der sich die Instanz befindet, steht noch nicht fest. Es existiert im Hauptspeicher noch keine Instanz. Die Variable `b` hat zu diesem Zeitpunkt den vordefinierten Wert `null`. Dieser Wert beschreibt sehr gut, dass die Variable noch keine Instanz referenziert. So kann z. B. in einer `if`-Anweisung

```
if (b != null) ...
```

überprüft werden, ob sich hinter einem Bezeichner tatsächlich schon eine Instanz einer Klasse verbirgt. Nur wenn die Bedingung `(b != null)` den Wert `true` zurückliefert, existiert bereits eine Instanz der Klasse `Bruch`, und nur dann kann auch auf die Attribute dieser Instanz zugegriffen werden.

Eine neue Instanz der Klasse `Bruch` erzeugen Sie mit dem Operator `new` (siehe Abbildung 5.9):

```
new Bruch();
```

Wie Sie aus Abbildung 5.9 entnehmen können, fehlt hier die Verbindung zu einem Bezeichner, über den Sie auf das Objekt zugreifen können. Deshalb werden in der Regel

die beiden Anweisungen `Bruch b;` und `new Bruch();` zu einer Anweisung der Form `Bruch b = new Bruch();` verbunden (siehe Abbildung 5.10).

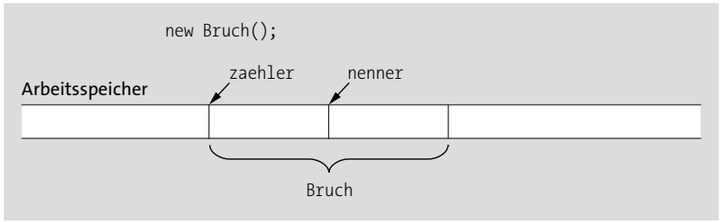

Abbildung 5.9 Erzeugen einer Instanz des Typs »Bruch«

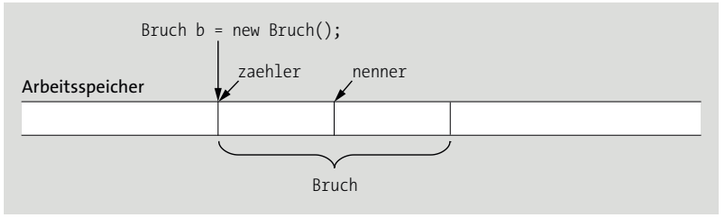

Abbildung 5.10 Erzeugen einer Variablen als Instanz der Klasse »Bruch«

Nun besteht über den Bezeichner `b` eine Verbindung zu dem für das Objekt im Speicher reservierten Speicherbereich, und Sie können auf das Objekt zugreifen.

5.2.1 Zugriff auf die Attribute (Datenelemente)

Die Instanz einer Klasse enthält die in der Klassendefinition festgelegten Attribute. In unserem Beispiel sind dies die Attribute `zaehler` und `nenner`. Diese Datenelemente können einzeln angesprochen werden. Die Syntax für den Elementzugriff lautet:

```
variable.elementname
```

In unserem Beispiel mit der Referenzvariablen `b` können Sie auf den Zähler des Bruchs mit `b.zaehler` und auf den Nenner mit `b.nenner` zugreifen. Somit können Sie mit den Wertzuweisungen

```
b.zaehler = 3;
b.nenner = 4;
```

der Variablen `b` den Wert ¾ zuweisen. Damit stellt sich ein Blick in den Hauptspeicher so dar wie in Abbildung 5.11.

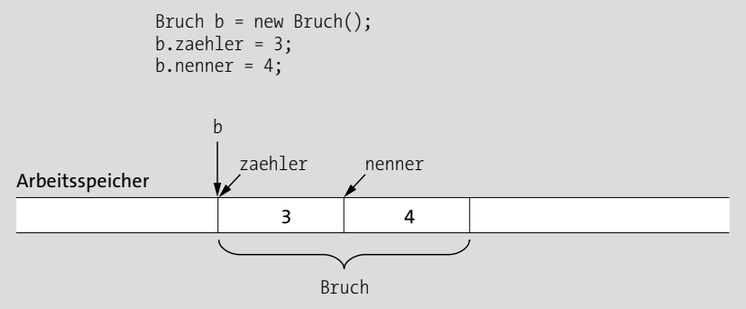

Abbildung 5.11 Variable »b« nach der Wertzuweisung

Merke

Die Attribute eines Objekts werden über den Objektnamen angesprochen. Auf den Objektnamen folgt, durch einen Punkt getrennt, der Name des Attributs. Diese Zugriffsmöglichkeit (von außen) kann und soll sogar vom Programmierer unterbunden werden (siehe Abschnitt 6.2.2, »Datenkapselung«). Sie wird hier nur der Vollständigkeit halber beschrieben.

Mit den Attributen eines Objekts können Sie in gleicher Weise operieren wie mit lokalen Variablen des gleichen Typs. Dementsprechend sind z. B. die folgenden Operationen möglich:

```
b.zaehler++; // Inkrementierung des Zählers
if (b.nenner != 0) //Prüfen, ob der Nenner ungleich null ist
```

An der Schreibweise mit dem Punkt zwischen Objektbezeichner und Datenelementbezeichner können Sie erkennen, dass hier mit dem Attribut eines Objekts und nicht mit einer lokalen Variablen gearbeitet wird.

5.2.2 Wertzuweisungen bei Objekten

Eine häufige Fehlerquelle beim Umgang mit Objekten besteht darin, dass Wertzuweisungen falsch vorgenommen werden. Objekte sind Referenztypen. Bei solchen Typen hat eine Wertzuweisung andere Folgen als bei den primitiven Typen. Die Zusammenhänge sollen hier an Beispielen deutlich gemacht werden.

Gehen wir zunächst von zwei Variablen des primitiven Typs `int` aus. Die Anweisung

```
int zahl1 = 2387;
```


hat zur Folge, dass im Hauptspeicher eine Variable `zahl1` mit dem Wert 2387 angelegt wird (siehe Abbildung 5.12).

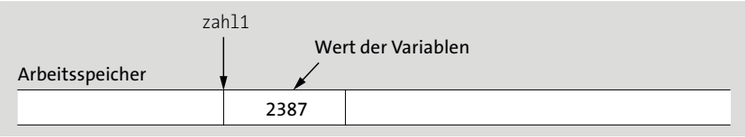

Abbildung 5.12 Anlegen einer »int«-Variablen

Folgt als weitere Anweisung

```
int zahl2 = zahl1;
```

wird eine weitere Variable mit dem gleichen Inhalt angelegt (siehe Abbildung 5.13).

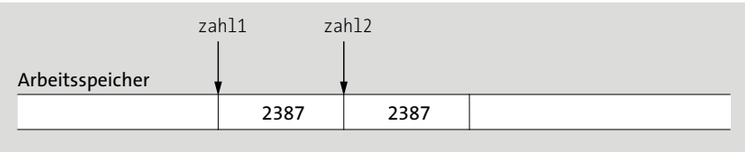

Abbildung 5.13 Erstellen einer Kopie der Integer-Variablen

Es existieren anschließend zwei Variablen mit unterschiedlichen Bezeichnern, die beide den gleichen Wert haben. Ändern Sie mit

```
zahl1 = 46;
```

den Inhalt von `zahl1`, so hat dies keinen Einfluss auf den Inhalt von `zahl2` (siehe Abbildung 5.14).

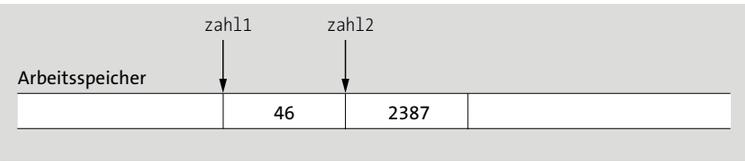

Abbildung 5.14 Wertzuweisung an die erste Variable

Das scheint so weit ganz selbstverständlich und sollte keine Schwierigkeiten bereiten. Wenn Sie nun aber mit Referenzvariablen wie bei den Objekten arbeiten, ergeben sich davon abweichende Verhältnisse.

Angenommen, Sie definieren ähnlich wie oben eine Variable, weisen dieser einen Wert zu und definieren anschließend eine zweite Variable, der Sie den Wert der ersten zuweisen, nur dass Sie jetzt Objekte statt primitiver Typen verwenden. Sie lassen z. B. die folgenden Anweisungen ausführen:

```
Bruch a = new Bruch(); // a als Bruch definiert
a.zaehler = 3; // dem Bruch den Wert ¾ zuweisen
a.nenner = 4;
Bruch b = a; // b als Bruch mit dem Wert von a
```

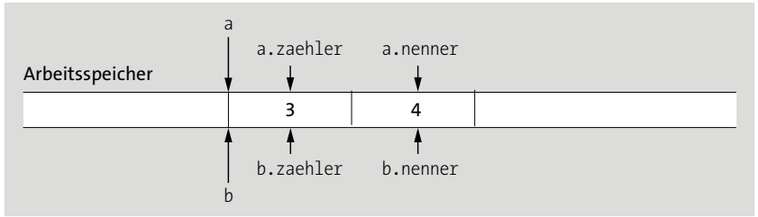

Abbildung 5.15 Wertzuweisung bei Objekten

Wie Sie Abbildung 5.15 entnehmen können, existiert jetzt kein zweites Objekt mit gleichem Zähler und Nenner wie beim ersten, sondern es existiert lediglich ein zweiter Bezeichner, der aber auf das identische Objekt im Hauptspeicher verweist. Dieses Verhalten hat jetzt aber weitreichende Folgen. Zum Beispiel werden durch das Verändern von Zähler und Nenner des zweiten Objekts

```
b.zaehler = 5;
b.nenner = 8;
```

zugleich auch der Zähler und der Nenner des ersten Objekts verändert, denn es handelt sich ja eigentlich immer nur um ein einziges Objekt, auf das Sie mithilfe von zwei unterschiedlichen Bezeichnern zugreifen können. Man nennt dieses Verhalten auch *Aliasing*, denn ein und dasselbe Objekt besitzt dadurch einen Alias-Bezeichner.

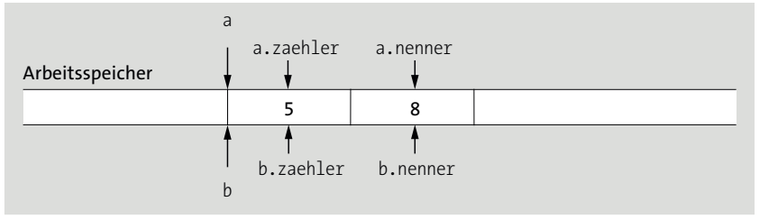

Abbildung 5.16 Auswirkung der Wertzuweisung an ein Objekt

Dieses Aliasing wirkt sich jetzt auch auf Vergleiche von Objekten aus. Vergleicht man z. B. in einer `if`-Anweisung die beiden Brüche `a` und `b` mit

```
if (a == b) ...
```

so liefert der Vergleich nur dann `true` als Ergebnis zurück, wenn es sich tatsächlich um ein und dasselbe Objekt mit zwei unterschiedlichen Bezeichnern handelt (siehe Abbildung 5.16). Im folgenden Beispiel werden zwei unterschiedliche Objekte erzeugt, deren Zähler und Nenner identisch sind. Vergleichen Sie diese beiden Objekte aber, so liefert der Vergleich immer `false` zurück, auch wenn sie vom gespeicherten Wert her eigentlich gleich sind (siehe Abbildung 5.17):

```
Bruch a = new Bruch(); // a als Bruch definieren
a.zaehler = 3; // dem Bruch den Wert ¾ zuweisen
a.nenner = 4;
Bruch b = new Bruch(); // b als zweiten Bruch definieren
b.zaehler = a.zaehler; // b den Wert von a zuweisen
b.nenner = a.nenner;
if (a == b) ... // liefert false zurück
```

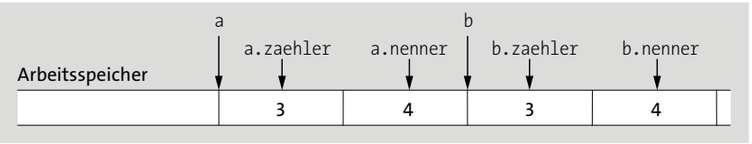

Abbildung 5.17 Echte Kopie eines Objekts

Was sich hier zunächst als etwas problematisch darstellt, ist in der Praxis überhaupt nicht problematisch. Java vermeidet aus Gründen der Performance und des Speicherplatzbedarfs, bei jeder Wertzuweisung eine komplette Kopie eines Objekts anzulegen. In den wenigen Fällen, in denen Sie tatsächlich eine exakte Kopie eines Objekts benötigen, gehen Sie dann in der zuletzt beschriebenen Form vor.

5.2.3 Gültigkeitsbereich und Lebensdauer

Auch was den Gültigkeitsbereich und die Lebensdauer betrifft, gibt es Unterschiede zwischen lokalen Variablen und den Datenelementen eines Objekts. Wie bereits bei der Behandlung der lokalen Variablen bemerkt, gelten diese nur innerhalb des Blocks, in dem sie definiert wurden. Die Attribute eines Objekts haben dagegen unabhängig von der Stelle, an der sie definiert wurden, innerhalb der gesamten Klasse Gültigkeit.

Die Lebensdauer von lokalen Variablen beginnt in dem Augenblick, in dem das Programm die Stelle ihrer Definition erreicht, und sie endet mit dem Verlassen des Blocks, innerhalb dessen die Definition erfolgte.

Beispiel:

```
if (x > 10) {
  Bruch b = new Bruch();
  b.zaehler = 2;
  b.nenner = 3;
}
System.out.println(b.zaehler);
```

Listing 5.4 Zugriffsversuch nach Ablauf der Lebensdauer

Der Bruch `b` wird nur erzeugt, wenn `x` größer als 10 ist. Aber auch in diesem Fall erfolgt die Erzeugung des Bruchs `b` innerhalb des Blocks, der mit der geschweiften Klammer nach der Bedingung (`x > 10`) beginnt. Die Lebensdauer endet somit mit der schließenden Klammer. Danach ist kein Zugriff mehr auf `b` möglich. Die `System.out.println`-Anweisung wird entsprechend eine Fehlermeldung verursachen.

Die Attribute eines Objekts existieren immer so lange, wie das Objekt selbst existiert. Erzeugt wird ein Objekt mit der `new`-Anweisung. Damit ist das auch der Zeitpunkt, zu dem die Attribute des Objekts entstehen. Die Lebensdauer des Objekts endet automatisch, sobald es im Programm keine Referenz mehr auf das Objekt gibt. Wann genau das passiert, kann nicht eindeutig vorhergesagt werden, denn darüber entscheidet das Laufzeitsystem. Man kann aber sagen, dass es spätestens dann passiert, wenn keine Zugriffsmöglichkeit (Referenz) mehr besteht und der verfügbare Speicherplatz zur Neige geht.

5.3 Methoden

Eine Erweiterung, zu der es bei den primitiven Datentypen nichts Vergleichbares gibt, sind die Methoden einer Klasse. Methoden können in Klassen neben den Datenelementen definiert werden. Sie beschreiben das Verhalten einer Klasse bzw. die Operationen, die mit den Attributen der jeweiligen Klasse ausgeführt werden können. Im Vergleich dazu beschreiben die Datenelemente einer Klasse den Aufbau und den Zustand einer Instanz.

5.3.1 Aufbau von Methoden

Eine Methode besteht aus dem Kopf, der quasi als Überschrift dient, und dem Rumpf. Im Rumpf wird festgelegt, welche Vorgänge mit dem Aufruf der Methode ablaufen sollen. Im Kopf werden der Rückgabetyp, der Bezeichner der Methode und in runden Klammern die Datentypen und Bezeichner von Übergabeparametern festgelegt. Im Rumpf befinden sich Java-Anweisungen, die wie ein Programm abgearbeitet werden.

Der allgemeine Aufbau einer Methode hat die folgende Form:

```
Rückgabetyp Methodenname(Parametertyp Parameter1, ...)
```

Als erstes Beispiel definieren wir eine Methode, die dafür sorgt, dass ein Objekt der Klasse `Bruch` in einem bestimmten Format ausgegeben wird.

```
void ausgeben() { // Kopf der Methode
    System.out.print(zaehler + "/" + nenner); // Rumpf der Methode
}
```

Das im Kopf verwendete Schlüsselwort `void` ist bereits von der Zeile `public static void main(String[] args)` bekannt. Methoden können Daten als Ergebnis zurückliefern. Von welchem Datentyp das Ergebnis ist, müssen Sie bei der Methodendefinition vor dem Methodennamen angeben. Liefert eine Methode kein Ergebnis zurück, müssen Sie als Pseudodatentyp das Schlüsselwort `void` angeben. Es bedeutet so viel wie: Die Methode liefert nichts zurück.

Der Rumpf der Methode besteht in diesem Beispiel aus einer einzigen Anweisung, kann aber durchaus auch sehr komplex und umfangreich ausfallen. Die geschweiften Klammern, die den Methodenrumpf einleiten und abschließen, müssen Sie immer verwenden, auch wenn wie hier nur eine einzige Anweisung im Methodenrumpf steht. Das ist ein wesentlicher Unterschied zu den Kontrollstrukturen, wo die geschweiften Klammern in einem solchen Fall auch wegfallen können.

5.3.2 Aufruf von Methoden

Abbildung 5.18 zeigt die um die Methode `ausgeben()` erweiterte Version der Klasse `Bruch`. Im Programm *Bruchtest* wird die Ausgabe nicht mehr direkt über den Aufruf von `System.out.print` realisiert, sondern indirekt über den Aufruf der Methode `ausgeben()`. Es handelt sich hierbei um eine Instanzmethode. So wie jede Instanz der Klasse `Bruch` über ihre eigenen Attribute verfügt, so verfügt jede Instanz auch über ihre eigenen Instanzmethoden. Sie sehen, dass der Zugriff auf eine Instanzmethode in gleicher Weise erfolgt wie der Zugriff auf die Attribute eines Objekts. Sie geben hinter dem Objektbezeichner

mit Punkt getrennt den Methodennamen und in Klammern eventuell zu verwendende Parameter an.

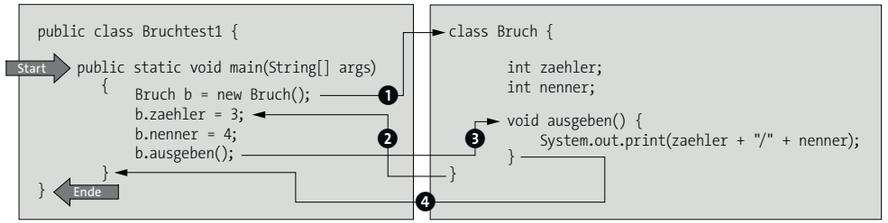

Abbildung 5.18 Methodenaufruf

Die Pfeile links zeigen, dass der Programmablauf immer mit der Methode `main` beginnt und mit dem Erreichen vom Ende des Methodenrumpfes endet (d. h. mit der schließenden geschweiften Klammer).

Bei Erreichen der Anweisung `new Bruch();` wird der erste Zugriff auf die Datei mit der Definition der Klasse `Bruch` erforderlich ❶. Mit dieser Anweisung wird die gesamte Struktur der Klasse `Bruch` im Hauptspeicher angelegt und über den Variablennamen `b` zugreifbar gemacht. Im Hauptspeicher entsteht quasi ein Abbild des Dateiinhalts. Werden mehrere Objekte der gleichen Klasse mit `new` erzeugt, so werden die Methoden im Hauptspeicher allerdings nur ein einziges Mal erzeugt. Alle Objekte verwenden dann die gleiche Implementierung der Methoden. Danach wird die nächste Anweisung im Programm abgearbeitet ❷. In diesem Fall wird dem Zähler unseres Bruchs der Wert 3 und dem Nenner der Wert 4 zugewiesen.

Mit der Anweisung `b.ausgeben();` wird nun auf das Abbild im Hauptspeicher zugegriffen ❸ und die Methode `ausgeben()` des Objekts `Bruch` abgearbeitet. In diesem Fall wird also die Anweisung `System.out.print` ausgeführt. Nachdem das Ende des Methodenrumpfes erreicht wurde, wird mit der nächsten Anweisung im Programm fortgefahren ❹. In unserem Fall ist damit das Ende des Methodenrumpfes von `main` erreicht, und das Programm wird beendet.

Unser Beispiel zeigt nun auch, wie eine Methode benutzt wird. Wie beim Zugriff auf ein Attribut eines Objekts richtet sich auch der Aufruf einer Methode immer an ein bestimmtes Objekt einer Klasse. Ohne diese Zuordnung zu einer bestimmten Instanz kann keine Methode aufgerufen werden. Die Syntax (Schreibweise) entspricht der beim Zugriff auf ein Attribut. In der Form

```
variablenname.methodenname();
```


wird also zunächst der Variablenname und, mit einem Punkt abgetrennt, der Methodenname angegeben. In unserem Beispiel verwenden wir entsprechend `b.ausgeben()`, um den Bruch mit dem Variablennamen `b` auszugeben.

Der Aufruf einer Methode erfolgt immer in drei Schritten:

- Der aufrufende Block wird unterbrochen.
- Der Methodenrumpf wird ausgeführt.
- Der aufrufende Block wird mit der Anweisung nach dem Aufruf fortgesetzt.

Ein Block kann so beliebig viele Aufrufe ein und derselben Methode beinhalten (siehe Abbildung 5.19). Es wird dementsprechend beliebig oft unterbrochen, um immer wieder den gleichen Anweisungsteil der Methode zu durchlaufen (zweiter Aufruf mit ❺ und ❻).

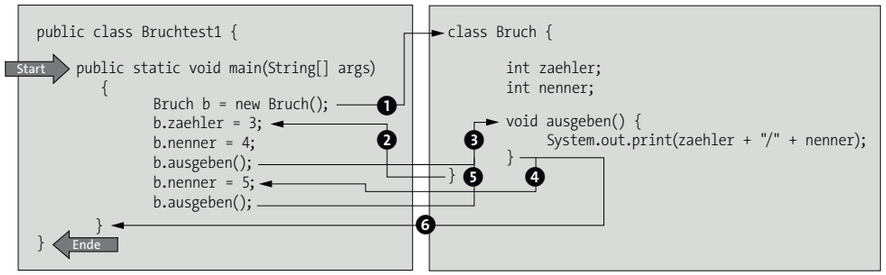

Abbildung 5.19 Mehrfacher Methodenaufruf

Der Methodenrumpf stellt in der gleichen Bedeutung, wie wir ihn bisher kennengelernt haben, einen Block dar. Innerhalb dieses Blocks sind beliebige Anweisungen (damit auch alle Kontrollstrukturen) zulässig. Es können auch lokale Variablen definiert werden, die dann nur innerhalb des Methodenrumpfes gültig sind. Sie werden bei jedem Methodenaufruf neu erzeugt und nach dem Aufruf wieder zerstört. Zur Erinnerung: Die Datenelemente des Objekts werden mit der Ausführung der `new`-Anweisung erstellt. Sie werden erst zerstört, wenn das Objekt insgesamt aus dem Speicher entfernt wird.

Wir erweitern unsere Definition nun um eine Methode, die es dem Bruch ermöglicht, sich zu kürzen. Ein Bruch kann mit dem größten gemeinsamen Teiler (ggT) gekürzt werden. Wir verwenden also unseren Algorithmus zur Berechnung des ggT nach Euklid, den wir in Kapitel 3, »Kontrollstrukturen«, programmiert haben.

Unser Testprogramm ändern wir so ab, dass der Bruch zunächst den Wert $^3/_{12}$ hat. Zur Kontrolle wird der Bruch ungekürzt ausgegeben, dann wird er gekürzt und schließlich noch einmal in gekürzter Form ausgegeben:

```
public static void main(String[] args) {
  Bruch b = new Bruch();
  b.zaehler = 3;
  b.nenner = 12;
  b.ausgeben();
  System.out.print("\n Und nach dem Kürzen: ");
  b.kuerzen();
  b.ausgeben();
}
```

Listing 5.5 »main«-Methode von »Bruchtest1«

In der Klasse `Bruch` wird die folgende Methode zum Kürzen ergänzt:

```
void kuerzen() {
  int m, n, r;  // lokale Variablen
  m = zaehler;
  n = nenner;
  r = m % n;
  while (r > 0) {
    m = n;
    n = r;
    r = m % n;
  }
  zaehler = zaehler / n;  // in n steht jetzt der ggT
  nenner = nenner / n;
}
```

Listing 5.6 Methode der Klasse »Bruch« zum Kürzen

Das Beispiel zeigt, dass beim Zugriff auf Datenelemente der Klasse (`zaehler` und `nenner`) innerhalb eines Methodenrumpfes desselben Objekts keine ausführliche Schreibweise für den Elementzugriff notwendig ist. Sie schreiben einfach `zaehler`, wenn Sie auf dieses Element zugreifen wollen. Von außerhalb, d. h. von einem Anwendungsprogramm aus, müssen Sie den Objektnamen mit angeben (`b.zaehler`). Sowohl Datenelemente als auch Methoden stehen innerhalb einer Klasse ohne weitere Maßnahmen direkt zur Verfügung. Dadurch ist es auch möglich, dass eine Methode eine andere Methode der Klasse aufruft. Diesen Sachverhalt zeigt die folgende Erweiterung unserer Klasse `Bruch`. Wir ergänzen eine weitere Methode, die einen Bruch gekürzt ausgibt:


```
void gekuerztausgeben() {
  kuerzen();
  ausgeben();
}
```

Die Methode fällt dadurch, dass wir auf die bereits definierten Methoden `kuerzen()` und `ausgeben()` zurückgreifen können, sehr kurz und übersichtlich aus. Im Testprogramm können Sie die beiden Anweisungen

```
b.kuerzen();
b.ausgeben();
```

ersetzen durch die Anweisung:

```
b.gekuerztausgeben();
```

Wie das Beispiel zeigt, können Methoden auch aus anderen Methoden heraus aufgerufen werden. Abbildung 5.20 verdeutlicht die Abläufe beim gegenseitigen Methodenaufruf. Sie entsprechen den Abläufen, wie sie für den Aufruf einer Methode aus einer Anwendung heraus bereits erläutert wurden.

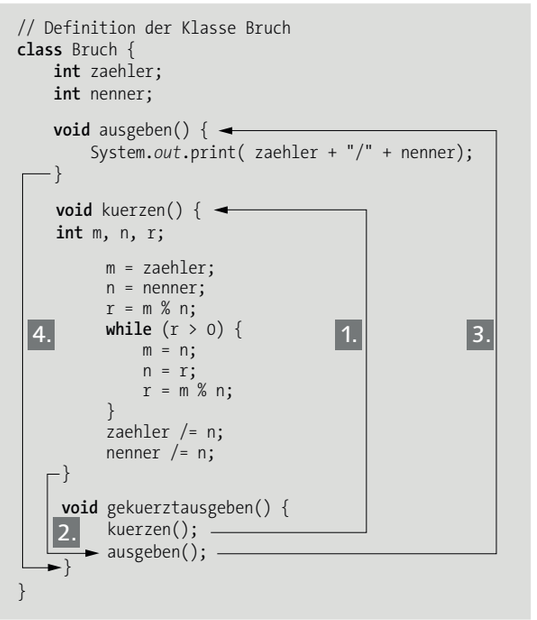

Abbildung 5.20 Methodenaufruf aus einer Methode

5.3.3 Abgrenzung von Bezeichnern

Da innerhalb von Methoden neben den Datenelementen auch lokale Variablen verwendet werden können und beim Zugriff auf Datenelemente kein Objektname vorangestellt ist, kann im Quellcode nicht zwischen einem Zugriff auf eine lokale Variable und einem Zugriff auf ein Datenelement unterschieden werden. Außerdem ist zu beachten, dass Namen von lokalen Variablen und Datenelementen nicht kollidieren. Dadurch ist es durchaus möglich, dass innerhalb einer Methode eine lokale Variable definiert wird, die den gleichen Namen trägt wie ein im Objekt bereits definiertes Datenelement. Zum Beispiel könnte, wie im unten stehenden Beispiel gezeigt, in der Methode `ausgeben()` eine lokale Variable mit dem Namen `zaehler` definiert werden, ohne dass der Compiler eine Fehlermeldung erzeugt.

```
void ausgeben() {
  int zaehler = 0; // namensgleiche lokale Variable
  System.out.print(zaehler + "/" + nenner);
}
```

Die Frage, die sich nun stellt, lautet: Worauf greift die Methode `ausgeben()` zurück?

Wenn Sie das Programm testen, werden Sie feststellen, dass mit `b.ausgeben()` für den `zaehler` immer der Wert 0 ausgegeben wird. Das bedeutet, dass die in der Methode `ausgeben()` definierte lokale Variable `zaehler` das gleichnamige Datenelement des Objekts überdeckt. Dies gilt grundsätzlich bei Namensgleichheit von lokalen Variablen und Datenelementen.

Damit Sie bei einer Überdeckung dennoch an das verdeckte Datenelement einer Klasse bzw. eines Objekts herankommen, existiert in jeder Methode automatisch eine Variable mit dem Namen `this`. Diese Variable wird als *Selbstreferenz* bezeichnet, weil sie immer auf das eigene Objekt verweist. Das eigene Objekt ist dasjenige Objekt, innerhalb dessen die Methode definiert wurde. In unserem Beispiel verweist `this` also auf die Klasse `Bruch` bzw. das Objekt `b`. Wenn wir also in der Methode `ausgeben()` die `System.out.print`-Anweisung folgendermaßen abändern

```
System.out.print(this.zaehler + "/" + nenner);
```

so wird der Wert des Datenelements `zaehler` korrekt mit dem Wert 3 ausgegeben.

Die hier erläuterte Anwendung zum Auflösen von Namenskollisionen ist aber nicht die einzige Anwendung der Selbstreferenz `this`. Auf diese Anwendungen wird später noch eingegangen.

5.4 Werte übergeben

Unsere bisher erstellten Methoden können ihre Arbeit verrichten, ohne dass sie dafür zusätzliche Informationen benötigen, bzw. die Methode kann wie beim Kürzen selbst ermitteln, mit welchem Wert (ggT) der Bruch gekürzt werden kann. In vielen Fällen sollen einer Methode beim Aufruf Informationen übergeben werden. Dadurch wird eine Methode flexibler, weil sie, von unterschiedlichen Werten ausgehend, entsprechend auch unterschiedliche Ergebnisse zurückliefern kann.

Als Beispiel soll die Klasse `Bruch` um eine Methode `erweitern()` ergänzt werden. Beim Erweitern eines Bruchs werden Zähler und Nenner mit dem gleichen Wert multipliziert. Mit welchem Wert multipliziert wird, kann die Methode aber nicht selbst »wissen« oder ermitteln. Den Wert legt der Anwender bzw. der Programmierer bei jedem Aufruf fest. Das bedeutet, dass Sie der Methode beim Aufruf »sagen« müssen, mit welchem Wert erweitert werden soll. Aber wie sag ich's der Methode?

5.4.1 Methoden mit Parameter

Die Übergabeparameter sind die Lösung. Wir haben sie bei unseren bisherigen Beispielen nicht benötigt. Sie wurden aber im Zusammenhang mit der allgemeinen Schreibweise einer Methodendefinition schon erwähnt. Werden keine Übergabeparameter verwendet, so bleibt die Klammer hinter dem Methodennamen leer. Wie bei einer Variablendefinition können in der Klammer Platzhalter zur Übergabe an die Methode eingetragen werden. In unserem Beispiel benötigen Sie einen ganzzahligen Wert, mit dem erweitert werden soll. Angenommen, Sie wollen den Bruch mit dem Wert 4 erweitern, dann müsste die Methode folgendermaßen aufgerufen werden:

```
b.erweitern(4);
```

Damit die Methode diese Information übernehmen kann, muss sie entsprechend einen Behälter vorsehen, in den die Information passt. In unserem Fall benötigt die Methode einen Behälter für die ganze Zahl, die in der Klammer übergeben wird. Entsprechend sieht die Methodendefinition folgendermaßen aus:

```
void erweitern(int a) {
  zaehler *= a;
  nenner *= a;
}
```

Listing 5.7 Methodendefinition mit Übergabeparameter

Zur Erinnerung:

`zaehler *= a;` ist gleichbedeutend mit `zaehler = zaehler * a;`.

Abbildung 5.21 zeigt, wie beim Aufruf der Methode `erweitern` der in Klammern beim Aufruf angegebene Zahlenwert in die `int`-Variable kopiert wird. Die in der Methodendefinition in Klammern stehenden Variablen werden als *Parameter* bezeichnet. Die beim Aufruf in Klammern stehenden Werte nennt man *Argumente*. Daher rührt übrigens auch der Name `args` in der Methode `main`. Es handelt sich hierbei um den Namen der Parameter, die als Argumente dem Hauptprogramm von der Kommandozeile übergeben werden.

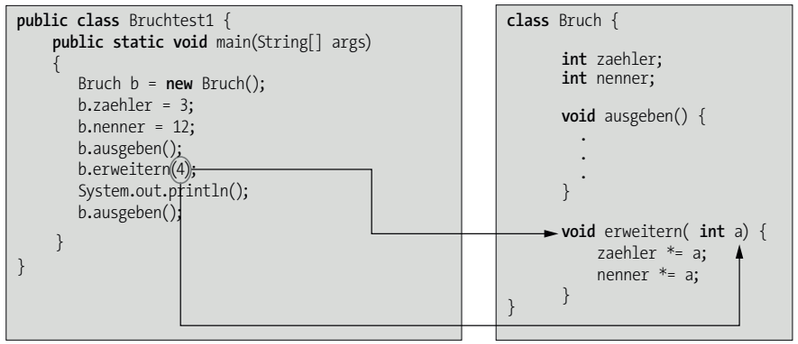

Abbildung 5.21 Parameterübergabe beim Methodenaufruf

Der bei der Methodendefinition gewählte Name (hier `a`) spielt für den Aufruf keine Rolle, denn der Bezeichner wird nur innerhalb der Methode verwendet.

Innerhalb der Klammern einer Methodendefinition können Sie auch mehrere Parameter angeben. Sie werden dann als Liste mit durch Kommata getrennten Definitionen angegeben:

```
void methodenname(typ1 name1, typ2 name2, ...)
```

Beim Aufruf werden dann auch die Argumente als durch Kommata getrennte Liste angegeben. Dabei müssen Sie beachten, dass die Reihenfolge der Argumente bestimmt, welcher Wert in welche Variable kopiert wird. Es wird der erste Wert in die erste Variable kopiert usw. Entsprechend muss die Anzahl der Argumente identisch mit der Anzahl der Parameter in der Definition sein. Es muss sich bei den Argumenten nicht um konstante Werte handeln. Es können dort beliebige Ausdrücke stehen, die bei der Auswertung zu einem Ergebnis führen, das zum Parametertyp passt. In unserem Beispiel könnte z. B. auch ein zu berechnender Ausdruck stehen:


```
b.erweitern(6 - 2);
```

Damit würde das Ergebnis der Berechnung (also der Wert 4) an die Methode übergeben.

In Abschnitt 5.3.2 wurden die drei Schritte angegeben, in denen ein Methodenaufruf abgewickelt wird. Diese drei Schritte gelten für Methodenaufrufe ohne Parameter. Werden Parameter benutzt, dann werden auch mehr Schritte notwendig:

- Die Werte aller Argumente werden berechnet.
- Die Parameter werden angelegt.
- Die Argumentwerte werden an die Parameter übergeben.
- Der aufrufende Block wird unterbrochen
- Der Methodenrumpf wird abgearbeitet.
- Die Parameter werden wieder zerstört.
- Der aufrufende Block wird fortgesetzt.

5.4.2 Referenztypen als Parameter

Es können auch Referenztypen (d. h. also auch Objekte) als Parameter übergeben werden. Als Beispiel soll hier eine Methode erstellt werden, die einen Bruch mit einem als Parameter übergebenen Bruch multipliziert. Das Ergebnis bildet die neuen Werte für Zähler und Nenner des Objekts, dessen Multiplikationsmethode aufgerufen wurde. Die Methode mit dem Namen `multipliziere` kann folgendermaßen codiert werden:

```
void multipliziere(Bruch m) {
  zaehler *= m.zaehler;
  nenner *= m.nenner;
}
```

Listing 5.8 Methode mit einem Referenztyp als Parameter

Sie können bereits an der Codierung der Methode erkennen, dass hier kein neues Objekt erstellt wird, denn es wird nirgendwo der Operator `new` eingesetzt. Somit stellt `m` nur einen Alias für das beim Aufruf verwendete Argument dar. Der folgende Programmcode nutzt die Multiplikationsmethode (siehe Abbildung 5.22).

Das beim Aufruf der Methode verwendete Argument `b` wurde mit `new` erzeugt. Beim Aufruf der Methode wird nur eine neue Referenz (ein neuer Verweis) auf das Argument mit dem Namen `m` erzeugt. Das bedeutet, dass beim Zugriff auf `m` eigentlich immer auf das »Original« `b` zugegriffen wird.

```
public class Bruchtest {
    public static void main(String[] args) {
        Bruch a = new Bruch();
        a.zaehler = 3;
        a.nenner = 4;
        Bruch b = new Bruch();
        b.zaehler = 3;
        b.nenner = 12;
        a.ausgeben();
        a.multipliziere(b);
        System.out.println();
        a.ausgeben();
    }
}
```

```
class Bruch {

    int zaehler;
    int nenner;

    void ausgeben() {
      .
      .
      .
    }

    void multipliziere(Bruch m) {

        zaehler *= m.zaehler;
        nenner *= m.nenner;
    }
}
```

Abbildung 5.22 Referenztyp als Parameter

Sie sehen, dass bei der Verwendung von Referenzvariablen als Parametern keine Kopie des Objekts, sondern eine Kopie der Referenz erstellt wird (siehe Abbildung 5.23).

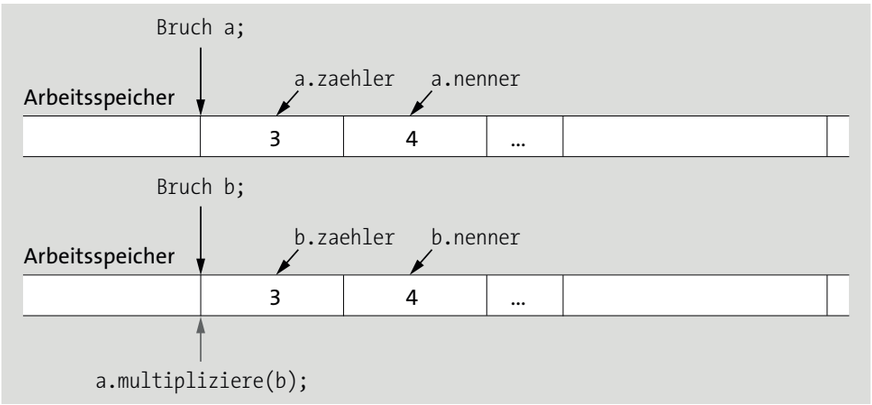

Abbildung 5.23 Zugriff auf einen Referenztyp

Der Aufruf `a.multipliziere(b);` bewirkt, dass die Methode mit `m` als Referenz auf `b` abgearbeitet wird. Problematisch kann dieses Verhalten sein, wenn Sie in der Methode schreibend auf die Parameter zugreifen. Ändern Sie z. B. mit der Anweisung `m.zaehler = 34;` den Wert des Zählers von `m`, dann haben Sie damit eigentlich den Wert von `b.zaehler` geändert. Dies müssen Sie bei Schreibzugriffen auf Parameter beachten, die als Referenz übergeben werden. Primitive Datentypen werden nicht als Referenz übergeben, sondern als echte Kopie. Damit wirken sich dort Schreibzugriffe nicht auf die Aufrufargumente aus.

5.4.3 Überladen von Methoden

Innerhalb einer Klasse können mehrere Methoden mit gleichem Namen existieren. Das kann aber nur funktionieren, wenn es ein eindeutiges Unterscheidungsmerkmal gibt. Dieses Unterscheidungsmerkmal ist die Parameterliste. Wenn Sie eine Methode mit einem Namen erstellen, der bereits für eine andere Methode verwendet wurde, dann bezeichnet man das als *Überladen* einer Methode.

Dieses Überladen bietet sich immer an, wenn Sie mehrere Methoden für eine ähnliche Funktion benötigen. Ein Beispiel für solche Methoden sind Methoden, die ein Objekt auf einen definierten Anfangszustand, vergleichbar mit einer Initialisierung, setzen. So können wir folgende beiden Methoden mit dem Namen `setze` definieren:

```
void setze(int z) {
  zaehler = z;
  nenner = 1;
}
void setze(int z, int n) {
  zaehler = z;
  nenner = n;
}
```

Listing 5.9 Beispiel für das Überladen einer Methode

Die erste Methode erwartet nur einen ganzzahligen Wert als Argument und übernimmt diesen Wert als Zähler. Den Nenner setzt die Methode immer auf den Wert 1. Damit entspricht der Wert, der übergeben wird, dem Gesamtwert des Bruchs.

Die zweite Methode erwartet zwei ganzzahlige Parameter, von denen der erste als Wert für den Zähler und der zweite als Wert für den Nenner übernommen wird.

Bei der Auswahl einer überladenen Methode ist es für den Compiler nicht immer eindeutig, welche Methode zu wählen ist. Zum Beispiel kann es sein, dass durch die implizite Typumwandlung mehrere Methoden geeignet wären. Eine Methode, die einen `double`-Wert erwartet, kann auch mit einem `int` als Argument aufgerufen werden. Die folgende Methode kann nur mit einem `Integer`-Wert als Argument aufgerufen werden:

```
void erweitern(int a) {
    ...
}
```

Der Aufruf `a.erweitern(5);` wäre gültig. Dagegen wäre der Aufruf `a.erweitern(5.0);` ungültig.

Wurde die Parameterliste folgendermaßen definiert

```
void erweitern(double a) {
    ...
}
```

dann sind die beiden Aufrufe `a.erweitern(5);` und `a.erweitern(5.0);` gültig.

Es ist nun aber auch möglich, dass beide Methoden existieren:

```
void erweitern(int a) {
    ...
}

void erweitern(double a) {
    ...
}
```

Da sie sich in der Parameterliste unterscheiden, stellt das kein Problem dar. Es stellt sich aber die Frage, welche der beiden Methoden tatsächlich vom Compiler ausgewählt wird, wenn der für beide Methoden passende Aufruf `a.erweitern(5);` verwendet wird.

Lösung: Der Compiler geht bei der *Overload-Resolution* nach folgenden Regeln vor: Zuerst werden alle passenden Methoden gesammelt; auch die Methoden, bei denen eine implizite Typumwandlung erforderlich ist, werden dabei mit einbezogen. Bleibt nur eine Methode übrig, wird diese ausgewählt. Passt überhaupt keine Methode, dann ist der Aufruf fehlerhaft und wird nicht übersetzt. Passen mehrere Methoden, wird diejenige ausgewählt, die am besten passt. Passen mehrere Methoden gleich gut, dann ist der Aufruf nicht eindeutig und wird nicht übersetzt. In unserem Beispiel passt die Methode genauer, bei der keine Typumwandlung erforderlich ist.

5.5 Ergebnisse

Mit der Übergabe von Argumenten an die Parameter einer Methode teilen wir der Methode mit, mit welchen Werten sie arbeiten soll. Es fließen Informationen von der aufrufenden Anweisung an die Methode. Häufig soll auch eine Information von der Methode zurück an die aufrufende Anweisung möglich sein. Zum Beispiel berechnet die Methode aus den Parametern einen Ergebniswert, der dann an die aufrufende Methode zurückgeliefert werden soll.

5.5.1 Methoden mit Ergebnisrückgabe

In unserer Klasse `Bruch` können wir z. B. eine Methode erstellen, die den Wert des Bruchs als Dezimalzahl zurückliefern soll. Folgende Schritte sind dazu erforderlich:

- Vor dem Methodennamen wird anstelle von `void` der Typ des Ergebnisses angegeben.
- Im Rumpf der Methode steht eine `return`-Anweisung, die einen Ausdruck enthält, der dem Typ des Ergebnisses entspricht.

Allgemein sieht der Aufbau einer Methode mit Ergebnisrückgabe folgendermaßen aus:

```
Datentyp methodenname(...) {
  return ausdruck;
}
```

Listing 5.10 Allgemeiner Aufbau einer Methode mit Rückgabewert

Der Ausdruck hinter `return` gibt den Wert an, der von der Methode zurückgegeben wird. Die oben als Beispiel genannte Methode kann dann folgendermaßen aussehen:

```
double dezimalwert() {
  return (double) zaehler/nenner;
}
```

Listing 5.11 Methode zur Rückgabe des Wertes als Dezimalzahl

Der Name der Methode kann nun überall dort verwendet werden, wo ein `double`-Wert stehen kann. Das heißt: In allen Ausdrücken und Anweisungen, die einen `double`-Wert verarbeiten können, kann die Methode `dezimalwert` verwendet werden. Als Beispiel wird hier die Ausgabe mit `System.out.print` gezeigt. Mit dieser Anweisung kann ein `double`-Wert auf der Konsole ausgegeben werden.

```
System.out.print(a.dezimalwert());
```

Die `print`-Anweisung ruft die Methode `dezimalwert` auf, die keine Parameter benötigt. Diese liefert als Ergebnis einen `double`-Wert zurück, der dann von der `print`-Anweisung ausgegeben wird. Im folgenden Beispiel wird der Methodenaufruf in einer `if`-Anweisung verwendet:

```
if (a.dezimalwert() < 3.5) {
    ...
}
```

Listing 5.12 Verwendung eines Methodenaufrufs in einer »if«-Anweisung

In einer Methode können auch mehrere `return`-Anweisungen stehen. Die nach der Programmlogik zuerst erreichte `return`-Anweisung entscheidet über den tatsächlich zurückgelieferten Wert, denn mit dem Erreichen der ersten `return`-Anweisung kehrt der Programmablauf zum Aufruf der Methode zurück.

Die folgende Methode `signum` liefert den Wert 1, wenn der Bruch einen Wert größer als 0 hat. Hat der Bruch einen Wert kleiner als 0, dann liefert sie den Wert –1 zurück, und wenn der Bruch den Wert 0 hat, liefert auch die Methode den Wert 0 zurück.

```
int signum() {
  if (this.dezimalwert() == 0) {
    return 0;
  }
  if (this.dezimalwert() > 0) {
    return 1;
  }
  return -1;
}
```

Listing 5.13 Methode »signum« der Klasse »Bruch«

Die dritte `return`-Anweisung wird nur erreicht, wenn keine der beiden `if`-Bedingungen erfüllt ist. Dies bestätigt die oben gemachte Aussage, dass die Methode nur bis zum Erreichen der ersten `return`-Anweisung abgearbeitet wird.

Als Programmierer müssen Sie sicherstellen, dass in jedem Fall eine `return`-Anweisung erreicht wird. Falls Sie in der obigen `signum`-Methode die letzte `return`-Anweisung auskommentieren, meldet Eclipse einen Fehler und weist darauf hin, dass diese Methode einen `int`-Wert zurückliefern muss. Dies kann sie aber nur durch Erreichen einer entsprechenden `return`-Anweisung.

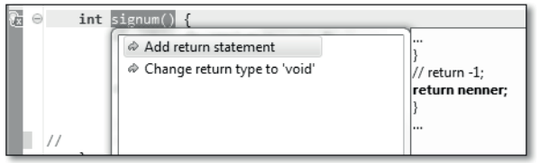

Abbildung 5.24 Hinweis auf fehlende »return«-Anweisung

Abbildung 5.24 zeigt den Hinweis von Eclipse. Ein Klick auf das Fehlersymbol am linken Rand des Editorfensters bringt die als *Quick-Fix* bezeichneten Vorschläge von Eclipse zum Vorschein. Im linken Bereich werden in diesem Fall zwei Vorschläge gemacht. Sie können eine `return`-Anweisung hinzufügen, oder Sie können als Rückgabewert `void` angeben und so auf jegliche Rückgabe verzichten. Natürlich macht hier nur der erste Vorschlag richtig Sinn.

Je nachdem, welchen Vorschlag Sie markieren, wird im rechten Bereich angezeigt, welche Änderungen im Quellcode erforderlich sind. Ein Doppelklick auf einen der beiden Vorschläge bringt Eclipse dazu, den Eintrag im Quellcode vorzunehmen. Auch wenn die Vorschläge selten zu 100 % passen, so stellt Eclipse mit Quick-Fix eine insgesamt sehr komfortable Möglichkeit zur Fehlerkorrektur zur Verfügung. Es reichen als eigene Änderungen meist geringfügige Anpassungen.

5.5.2 Methoden ohne Ergebnisrückgabe

Soll eine Methode kein Ergebnis zurückliefern, wird als Ergebnistyp der Ausdruck `void` angegeben. Es handelt sich dabei um einen *Pseudo-Typ*, der so viel bedeutet wie »nichts«. Wir haben solche Methoden bereits mehrmals verwendet. Sie können aus einer solchen Methode an jeder Stelle mit einer `return`-Anweisung ohne Ergebnisausdruck, also mit folgender Anweisung zur aufrufenden Anweisung zurückkehren:

```
return;
```

Am Ende einer Methode ohne Ergebnisrückgabe kehrt der Programmablauf automatisch zur aufrufenden Anweisung zurück. Sie müssen dort keine `return`-Anweisung einfügen. Entsprechend kann bei einer solchen Methode die `return`-Anweisung komplett fehlen. Sie wird am Ende der Methode quasi implizit ergänzt.

Beim Überladen von Methoden müssen Sie beachten, dass überladene Methoden sich nicht ausschließlich durch den Ergebnistyp unterscheiden dürfen. Sie müssen sich also zusätzlich noch in der Parameterliste unterscheiden. Da der Ergebnistyp beim Aufruf nicht angegeben wird, könnte der Compiler nicht entscheiden, welche Methode verwendet werden soll.

5.6 Konstruktoren als spezielle Methoden

Objekte werden mit dem `new`-Operator erzeugt. Nach dem Erzeugen eines Objekts sollte es sich grundsätzlich in einem definierten Anfangszustand befinden. Der Zustand eines Objekts unserer Klasse `Bruch` wird durch die Werte der Attribute `zaehler` und `nenner` beschrieben. Wenn wir davon ausgehen, dass nach dem Erzeugen eines Objekts der Klasse `Bruch` beide Attribute den Wert 0 haben, dann befindet sich unser Bruch in einem Zustand, der in der Mathematik als ungültig angesehen wird. Um dies zu vermeiden,

können Sie nach der Erzeugung eines Objekts grundsätzlich zuerst den Wert mit der Methode `setze` auf einen definierten Wert festlegen:

```
Bruch b = new Bruch();
b.setze(0);
```

Die Verwendung eines Konstruktors vereinfacht diese Vorgehensweise dadurch, dass er die Aufgabe, das Objekt in einen definierten Anfangszustand zu versetzen, mit der Erzeugung des Objekts zu einer einzigen Anweisung zusammenfasst. Ein Konstruktor ist zunächst nichts anderes als eine Methode. Es gibt aber einige Besonderheiten, die einen Konstruktor von einer gewöhnlichen Methode unterscheiden:

- Der Name eines Konstruktors entspricht immer exakt dem Namen der Klasse.
- Die Definition eines Konstruktors beginnt immer mit dem Namen, ohne vorangestelltes `void` oder eine andere Typangabe.
- Ein Konstruktor wird automatisch mit dem `new`-Operator aufgerufen.

Die letzte Aussage wirft die Frage auf, welcher Konstruktor in unseren Beispielprogrammen aufgerufen wurde, denn bisher haben wir noch keinen Konstruktor definiert.

Wurde vom Programmierer kein Konstruktor definiert, so erzeugt der Compiler beim Übersetzen der Klasse einen *Default-Konstruktor*, der keine weiteren Anweisungen beinhaltet. Somit besitzt jede Klasse einen Konstruktor: entweder einen vom Programmierer definierten oder den Standardkonstruktor (Default-Konstruktor) mit leerer Parameterliste und leerem Rumpf. Da der Konstruktor den gleichen Namen wie die Klasse besitzt, heißt dieser z. B. für unsere Klasse `Bruch`

```
Bruch()
```

und genau diesen Konstruktor rufen wir mit der Zeile

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

auf. Der Standardkonstruktor existiert also für jede Klasse, ohne dass wir ihn definieren müssen. Anstelle des Default-Konstruktors können wir als Programmierer einen selbst definierten Konstruktor erstellen, der das Objekt in einen definierten Anfangszustand versetzt. Ein Konstruktor wird genauso definiert wie eine andere Methode. Es müssen lediglich die oben genannten Besonderheiten beachtet werden. Entsprechend muss der Name des Konstruktors dem Objektnamen entsprechen, und es wird kein Datentyp bzw. kein `void` vorangestellt. Um den Bruch auf den Wert 0 vorzubesetzen, können wir entsprechend folgenden Konstruktor definieren:


```
Bruch() {
  zaehler = 0;
  nenner = 1;
}
```

Listing 5.14 Selbst definierter Konstruktor der Klasse »Bruch«

Der Nenner könnte ebenso gut auf einen anderen Wert ungleich 0 gesetzt werden. Entscheidend ist, dass der `zaehler` auf 0 und der `nenner` auf einen Wert ungleich 0 gesetzt wird. Damit hat der Bruch unmittelbar nach der Erzeugung mit

```
Bruch a = new Bruch();
```

den rechnerischen Wert 0. Wurde der Default-Konstruktor einmal überschrieben, wird grundsätzlich der neue Konstruktor verwendet. Auch der neue Konstruktor heißt Standard- oder Default-Konstruktor, solange er eine leere Parameterliste verwendet.

5.6.1 Konstruktoren mit Parametern

Es können beliebig weitere Konstruktoren mit Parametern definiert werden, so wie wir es von den Methoden her kennen. Beim Erzeugen eines Objekts wird dann immer der von der Parameterliste her passendste Konstruktor verwendet. Konstruktoren mit Parameter heißen *Custom-Konstruktoren* (spezielle Konstruktoren).

Der folgende Konstruktor ist z. B. geeignet, um einen Bruch direkt beim Erzeugen auf einen Wert ungleich 0 zu setzen:

```
Bruch (int z, int n) {
    zaehler = z;
    nenner = n;
}
```

Mit der Anweisung `Bruch a = new Bruch(2, 3);` wird der Wert des Bruchs bei der Erzeugung direkt auf den Wert $^2/_3$ gesetzt. Hier wird nun endgültig deutlich, dass hinter dem `new`-Operator ein Methodenaufruf steht.

Im Zusammenhang mit Konstruktoren ist Ihnen vielleicht aufgefallen, dass die Attribute eines Objekts nicht initialisiert werden. Vor dem Überschreiben des Default-Konstruktors unseres Bruchs trat kein Fehler auf. Würde man eine lokale Variable verwenden, ohne dass ihr explizit ein Wert zugewiesen wird, würde das Programm nicht übersetzt werden und der Fehler »The local variable may not have been initialized« würde angezeigt werden. Werden hingegen Attribute (Datenelemente) eines

Objekts nicht initialisiert, werden Default-Werte verwendet. Welche das sind, hängt vom jeweiligen Datentyp ab (siehe Tabelle 5.1).

Datentyp	Default-Wert
int	0
double	0.0
boolean	false
char	\u0000
Referenztypen	null

Tabelle 5.1 Initialisierung von Attributen

Wurden die Attribute einer Klasse mit der Definition bereits initialisiert, so nehmen sie diese Werte zeitlich bereits vor der Ausführung des Konstruktors an. Das bedeutet, dass ein anschließender Konstruktoraufruf diese Werte wieder überschreibt.

5.6.2 Verketten von Konstruktoren

Die meisten Konstruktoren müssen mehr Aufgaben als die Initialisierung von Attributen erledigen. Diese Aufgabe könnten Sie, wie oben gezeigt, auch ohne Konstruktoren durch explizite Initialisierungen realisieren. Oft finden in Konstruktoren bereits Überprüfungen statt, die in aufwendigen Kontrollstrukturen vorgenommen werden. Damit Sie diese Abläufe nicht in jedem einzelnen Konstruktor codieren müssen, können Sie Konstruktoren verketten. Bei der Verkettung erstellen Sie einen Konstruktor, der alle allgemeingültigen Abläufe beinhaltet, und rufen dann in weiteren Konstruktoren zuerst diesen Konstruktor auf, bevor Sie die zusätzlich zu erledigenden Abläufe hinzufügen. Für den Aufruf eines Konstruktors innerhalb eines anderen Konstruktors brauchen Sie das Schlüsselwort `this`. `This()` dient dazu, einen Konstruktor der Superklasse aufzurufen. Der folgende Quellcodeausschnitt zeigt diese Vorgehensweise am Beispiel unserer Klasse `Bruch`:

```
// Definition der Klasse Bruch mit verketteten Konstruktoren
class Bruch {
  int zaehler;
  int nenner;
  Bruch (int z, int n) {
    int hz, hn, r;
    if (n == 0) {
```



```
      System.out.print("Fehler! Der Nenner darf nicht 0 sein!");
    } else {
        hz = z;
        hn = n;
        r = hz % hn;
        while (r > 0) {
          hz = hn;
          hn = r;
          r = hz % hn;
        } // in hn steht jetzt der ggT
        zaehler = z/hn;
        nenner = n/hn;
    }
  }
  Bruch() {
    this(0, 1);
  }

  Bruch(int n) {
    this(n, 1);
  }
}
```

Listing 5.15 Definition der Klasse »Bruch« mit verketteten Konstruktoren

Es wird zuerst ein Konstruktor definiert, der zwei Parameter für Zähler und Nenner erwartet. Die beiden Parameter werden zur Initialisierung des Bruchs verwendet. Der Konstruktor übernimmt hier zusätzliche Aufgaben. Zuerst wird geprüft, ob der Nenner 0 ist. Da dies zu einem ungültigen Bruch führt, wird eine Fehlermeldung ausgegeben. Ist der Nenner nicht 0, so ist der Bruch gültig. Es könnte aber sein, dass der Bruch noch gekürzt werden kann. Deshalb wird zuerst der ggT berechnet und damit der gekürzte Wert für Zähler und Nenner bestimmt. Initialisiert werden Zähler und Nenner dann mit den gekürzten Werten.

Es folgt die Definition eines Konstruktors, der keinen Parameter erwartet. Er soll den Bruch mit dem Wert Null (Zähler = 0 und Nenner = 1) initialisieren. Diese Aufgabe kann aber der erste Konstruktor übernehmen. Er wird mit der Anweisung `this(0, 1);` aufgerufen.

Ebenso wird mit dem zweiten Konstruktor verfahren. Dieser erwartet einen Parameter `n` und soll den Bruch mit dem ganzzahligen Wert des Parameters, also `n/1`, initialisieren. Auch diese Aufgabe wird einfach an den ersten Konstruktor mit `this(n, 1);` übertragen.

Ein verketteter Konstruktoraufruf mit `this` muss immer als erste Anweisung im Konstruktorrumpf stehen. Anschließend können andere Anweisungen folgen, die nur für diesen Konstruktor gelten.

5.7 Übungsaufgaben

Aufgabe 1

Erstellen Sie im Projekt *JavaUebung05* im Package *uebung05* eine Klasse mit dem Namen `Kreis`. Die Klasse soll nur über ein Datenelement (Attribut) mit dem Namen `radius` verfügen, in dem der Radius als Kommazahl festgehalten wird. Erstellen Sie einen Konstruktor mit leerer Parameterliste, der den Radius mit dem Wert 0 initialisiert, und einen Konstruktor, dem als Parameter eine Kommazahl zur Initialisierung des Radius übergeben wird. Die Klasse soll über folgende Methoden verfügen:

```
double getRadius();
setRadius(double r);
double getUmfang();
double getFlaeche();
```

Erstellen Sie dazu ein Testprogramm mit dem Namen *Kreistest*, das mit einem `JOptionPane.showInputDialog` den Radius eines Kreises einliest und anschließend durch Aufruf der drei Methoden den Radius, den Umfang und die Fläche des Kreises in der Konsole (mit `System.out.println`) ausgibt.

Hinweis

Als Hilfestellung können Sie auf die Programme zur Kreisberechnung aus Kapitel 1, »Einführung«, und Kapitel 2, »Grundbausteine eines Java-Programms«, zurückgreifen.

Aufgabe 2

Erstellen Sie im Projekt *JavaUebung05* im Package *uebung05* eine Klasse mit dem Namen `Rechteck`. Die Klasse soll über die Attribute `laenge` und `breite` als `double`-Werte verfügen. Erstellen Sie einen Konstruktor mit leerer Parameterliste, der die beiden Kantenlängen jeweils mit dem Wert 0 initialisiert. Ein weiterer Konstruktor mit zwei `double`-Parametern soll die beiden Kantenlängen mit den übergebenen Werten initialisieren. Die Klasse soll zusätzlich über die folgenden Methoden verfügen:


```
setLaenge(double l);
setBreite(double b);
setSeiten(double l, double b);
double getLaenge();
double getBreite();
double getLangeSeite();
double getKurzeSeite();
double getDiagonale();
double getFlaeche();
double getUmfang();
```

Erstellen Sie ein Programm mit dem Namen *Rechtecktest*, das ein Objekt der Klasse `Rechteck` verwendet. Länge und Breite des Rechtecks sollen mit `JOptionPane.showInputDialog` eingegeben werden, und anschließend sollen die lange und die kurze Seite, die Diagonale, die Fläche und der Umfang in der Konsole ausgegeben werden.

Aufgabe 3

Erstellen Sie in der Klasse `Rechteck` die Methode `laengeAusgeben()`, wie unten vorgegeben. In der Methode wird eine lokale Variable mit dem gleichen Namen erstellt, wie er schon für das Attribut der Länge verwendet wurde, und ihr wird der Wert 5,4 zugewiesen.

```
void laengeAusgeben() {
    double laenge = 5.4;
    System.out.println("Länge: " + laenge);
}
```

Listing 5.16 Methode zum Ausgeben der Länge

Frage: Wird die Variable als Fehler markiert, weil der Name schon für das Attribut verwendet wurde?

Rufen Sie die Methode `laengeAusgeben()` als letzte Anweisung im Programm *Rechtecktest* auf.

Frage: Welcher Wert wird ausgegeben? Ist es der Wert des Attributs, den Sie beim Programmstart eingeben, oder ist es immer der Wert der lokalen Variablen `laenge` (5,4)?

Aufgabe 4

Erweitern Sie die Klasse `Rechteck` um folgende Methoden:

```
void laengeVergroessern(double l)
void breiteVergroessern(double b)
void laengeVerkleinern(double l)
void breiteVerkleinern(double b)
```

Die beiden Methoden vergrößern bzw. verkleinern die Länge bzw. die Breite des Rechtecks um den als Argument übergebenen Wert.

Testen Sie die Methoden im Programm *Rechtecktest*, indem Sie die eingegebenen Werte vor der Ausgabe vergrößern bzw. verkleinern.

Aufgabe 5

Erweitern Sie die Klasse `Kreis` um die folgenden Methoden:

```
void setUmfang(double u)
void setFlaeche(double f)
```

Die Methoden berechnen den Radius für einen Kreis mit dem übergebenen Umfang bzw. der übergebenen Fläche und setzen das Attribut `radius` auf den berechneten Wert.

Aufgabe 6

Erstellen Sie im Projekt *JavaUebung05* im Package *uebung05* ein Programm mit dem Namen *Kreistabelle*. Die Anwendung soll die Klasse `Kreis` verwenden und nach Eingabe (`JOptionPane.showInputDialog`) eines Startwertes für den Radius und einer Radiuserhöhung eine 30-zeilige Tabelle mit Radius, Umfang und Fläche nach folgendem Muster ausgeben:

Radius	Umfang	Fläche
5.0	31.41592653589793	78.53981633974483
10.0	62.83185307179586	314.1592653589793
15.0	94.24777960769379	706.8583470577034
20.0	125.66370614359172	1256.6370614359173
25.0	157.07963267948966	1963.4954084936207
30.0	188.49555921538757	2827.4333882308138
35.0	219.9114857512855	3848.4510006474966

Tabelle 5.2 Ausgabe des Programms Kreistabelle

Radius	Umfang	Fläche
40.0	251.32741228718345	5026.548245743669
45.0	282.7433388230814	6361.725123519332
50.0	314.1592653589793	7853.981633974483
55.0	345.57519189487726	9503.317777109125
60.0	376.99111843077515	11309.733552923255
65.0	408.4070449666731	13273.228961416875
...		

Tabelle 5.2 Ausgabe des Programms Kreistabelle (Forts.)

Hinweis

Verwenden Sie als Trennzeichen zwischen den einzelnen Ausgabewerten einer Zeile mehrere Tabulatorzeichen.

Aufgabe 7

Erstellen Sie im Projekt *JavaUebung05* eine Klasse `FlaechengleicherKreis` als Anwendungsprogramm, das ein Objekt der Klasse `Rechteck` und ein Objekt der Klasse `Kreis` verwendet.

Zuerst sollen die Länge und die Breite eines Rechtecks eingelesen werden (mit `JOptionPane.showInputDialog`). Anschließend ist der Radius des Kreises so zu bestimmen, dass er den gleichen Flächeninhalt wie das Rechteck hat.

Zur Kontrolle sollen die Länge, Breite und Fläche des Rechtecks und der Radius und die Fläche des Kreises untereinander in der Konsole ausgegeben werden. Die Ausgabe des Programms soll folgendermaßen aussehen:

```
Rechtecklänge: 10.0
Rechteckbreite: 20.0
Rechteckfläche: 200.0

Kreisradius: 7.978845608028654
Kreisfläche: 200.0
```

5.8 Ausblick

Sie kennen jetzt den für die moderne Programmierung so eminent wichtigen Begriff der Objektorientierung. Sie können neue Klassen mit Attributen und Methoden definieren und nach diesem Bauplan Objekte für Ihre Programme erzeugen. Sie können damit die zur Verfügung stehenden Datentypen gewissermaßen um eigene Typen erweitern, die zudem wesentlich leistungsfähiger sind und besser an Ihre Bedürfnisse angepasst werden können. Sie können damit die Vorteile der Objektorientierung nutzen.

Dadurch, dass Methoden zum Bestandteil der Klassen bzw. Objekte geworden sind, ergibt sich eine zwangsläufig sinnvolle Zuordnung. Die Methoden befinden sich immer dort, wo sie auch hingehören. Gerade in größeren Projekten ist es damit wesentlich einfacher, den Überblick zu behalten. Jedes Programm dient letztendlich dazu, Abläufe und Gegenstände der Realität abzubilden. Objekte erhöhen die Nähe zur Realität, denn auch in der Realität haben wir es mit Objekten zu tun, die sich durch Eigenschaften (Attribute) und Fähigkeiten (Methoden) auszeichnen. Was liegt also näher, als diese Sichtweise auch in die Programmierung zu übernehmen? Nicht zuletzt vereinfacht die Nutzung der Objektorientierung die Wiederverwendbarkeit einmal erstellten Programmcodes.

Sie haben bereits bei der Verwendung der Klasse `JOptionPane` feststellen können, dass es mit wenig Programmcode möglich ist, sehr leistungsfähige Objekte in eigenen Programmen zu verwenden. Sie müssen nichts über den sicher sehr komplexen Programmcode wissen, mit dem die Komponenten programmiert wurden. Aber Sie müssen diese Objekte einbinden und erzeugen können, indem Sie deren Konstruktoren aufrufen, und Sie müssen sich über die verfügbaren Attribute und Methoden informieren, damit Sie diese für Ihre eigenen Zwecke einsetzen können. Spätestens bei der Erstellung von grafischen Oberflächen werden Sie davon reichlich Gebrauch machen.

Sie haben damit einen ganz wichtigen Schritt auf dem Weg, den Sie eingeschlagen haben, hinter sich und sind damit gut vorbereitet, um die weiteren Kapitel erfolgreich zu meistern und noch weitere Vorteile der Objektorientierung zu nutzen.

Auch das folgende Kapitel wird sich um die Objektorientierung drehen. Sie werden erfahren, wie Sie auf bestehende Klassen zurückgreifen und daraus neue Klassen ableiten können. Sie können somit sehr effektiv auf bereits erstellte Funktionalitäten zurückgreifen, um diese zu modifizieren und um neue Fähigkeiten zu erweitern.

Auf einen Blick

Inhalt

Index

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

Hans-Peter Habelitz

Programmieren lernen mit Java

537 Seiten, broschiert, mit DVD, 3. Auflage 2015
19,90 Euro, ISBN 978-3-8362-3517-4

www.rheinwerk-verlag.de/3776

Hans-Peter Habelitz unterrichtet Informatik an einer Berufsschule. Er hat schon vielen Einsteigern das Programmieren beigebracht. Sein Wissen über wirksamen Unterricht hat er über 10 Jahre lang als Dozent für Fachdidaktik der Informatik an angehende Lehrer weitergegeben.

Teilen Sie Ihre Leseerfahrung mit uns!