

Kurzreferenz

Java

Autor: Michael Puff

Erstellt: 2011-02-04

Homepage: <http://www.michael-puff.de>
E-Mail: mail@michael-puff.de

Inhaltsverzeichnis

1	Einführung	1
1.1	Objektorientierung in Java	1
1.2	Zeiger und Referenzen	1
1.3	Garbage Collection	1
1.4	Ausnahmebehandlung	2
2	Die Sprache	3
2.1	Lexikalische Elemente	3
2.2	Anweisungen	5
2.3	Alles ist Klasse	5
2.4	Modifizierer	6
2.5	Anweisungen und Blöcke	6
2.6	Datentypen	7
3	Operatoren	11
3.1	Arithmetische Operatoren	11
3.2	Bitweise Operatoren	11
3.3	Relationale Operatoren	12
3.4	Boole'sche Operatoren	12
3.5	Zuweisungsoperator	12
4	Kontrollstrukturen	14
4.1	Verzweigungen	14
4.2	Schleifen	15
5	Klassen und Methoden	19
5.1	Klassengrundlagen	19
5.2	Klassenobjekte deklarieren und erzeugen	21
5.3	Konstruktoren	21
5.4	Methoden überladen	22
5.5	Argumentenübergabe	23
5.6	Der Modifizierer <i>static</i>	23
5.7	Zugriffskontrolle	23

6 Vererbung	25
6.1 Zugriff auf die Superklasse	26
6.2 Methoden überschreiben	27
6.3 Abstrakte Klassen	27
7 Ereignisse	29
8 Ausnahmebehandlung mit Exceptions	31
8.1 Ausnahmen behandeln mit <i>try/catch</i>	31
8.2 Abschlussbehandlung mit <i>finally</i>	32
8.3 Nicht behandelte Ausnahmen angeben	33
8.4 Ausnahmen selber auslösen	33
8.5 Definition von eigene Ausnahmen	34
9 MySQL Datenbankbindung	36
9.1 Zugriff über den ODBC-Dienst von Windows	36
9.2 Einbindung des MySQL-Connectors in Java	36
9.3 Basic JDBC Concepts	37
Literaturverzeichnis	40

Tabellenverzeichnis

2.1	Primitive Datentypen	7
2.2	Datentyp Konvertierungsmethoden	10
3.1	Arithmetische Operatoren	11
3.2	Logische bitweise Operatoren	11
3.3	Bitweise Schiebeoperatoren	11
3.4	Relationale Operatoren	12
3.5	Boole'sche Operatoren	12
3.6	Kurzzuweisungsoperatoren	13

1 Einführung

1.1 Objektorientierung in Java

Die Sprache Java ist nicht bis zur letzten Konsequenz objektorientiert, so wie Smalltalk es vorbildlich demonstriert. Primitive Datentypen wie Ganzzahlen oder Fließkommazahlen werden nicht als Objekte verwaltet. Der Design-Grund war vermutlich, dass der Compiler und die Laufzeitumgebung mit der Trennung besser in der Lage waren, die Programme zu optimieren.

1.2 Zeiger und Referenzen

In Java gibt es keine Zeiger (engl. pointer), wie sie aus anderen Programmiersprachen bekannt und gefürchtet sind. Da eine objektorientierte Programmiersprache ohne Verweise aber nicht funktioniert, werden Referenzen eingeführt. Eine Referenz repräsentiert ein Objekt, und eine Variable speichert diese Referenz. Die Referenz hat einen Typ, der sich nicht ändern kann. Ein Auto bleibt ein Auto und kann nicht als Laminiersystem angesprochen werden. Eine Referenz unter Java ist nicht als Zeiger auf Speicherbereiche zu verstehen, obwohl er es im Endeffekt ist.

1.3 Garbage Collection

Java besitzt eine sogenannte *Garbage Collection*. Das bedeutet, dass allozierter Speicher automatisch wieder freigegeben wird, ohne dass sich der Programmierer extra darum kümmern müsste. Erzeugt man zum Beispiel mit dem Schlüsselwort *new* eine Instanz einer Klasse, legt also ein Objekt an, welches Speicher reserviert, muss man diesen Speicher nicht mehr explizit freigeben, in dem man das Objekt zerstört. In anderen Programmiersprachen wie C++ und Delphi, die über keine Garbage Collection verfügen, muss man Speicher, den man selbst reserviert hat, auch zwingen wieder selber freigeben, wenn man ihn nicht mehr benötigt. Tut man dies nicht, ergeben sich

Speicherlöcher (engl. memory leaks). Dies kann unter Umständen dazu führen, dass man im schlimmsten Fall kein Speicher mehr zur Verfügung hat. Nicht so in Java. Das Generieren eines Objekts in einem Block mit anschließender Operation zieht eine Aufräumaktion des *Garbage Collectors* nach sich. Nach Verlassen des Wirkungsbereichs erkennt das System das nicht mehr referenzierte Objekt. Ein weiterer Vorteil des *Garbage Collectors*: Bei der Benutzung von Unterprogrammen werden oft Objekte zurückgegeben, und in herkömmlichen Programmiersprachen beginnt dann wieder die Diskussion, welcher Programmteil das Objekt jetzt löschen muss.

Der *Garbage Collector* ist ein nebenläufiger Thread im Hintergrund, der nicht referenzierte Objekte markiert und von Zeit zu Zeit entfernt. Damit macht der *Garbage-Collector* die Funktionen *free()* aus C oder *delete()* aus C++ überflüssig.

1.4 Ausnahmebehandlung

Java unterstützt ein modernes System, um mit Laufzeitfehlern umzugehen. In der Programmiersprache wurden *Exceptions* eingeführt: Objekte, die zur Laufzeit generiert werden und einen Fehler anzeigen. Diese Problemstellen können durch Programmkonstrukte gekapselt werden. Die Lösung ist in vielen Fällen sauberer als die mit Rückgabewerten und unleserlichen Ausdrücken im Programmfluss. In C++ gibt es ebenso *Exceptions*, die aber nicht so intensiv wie in Java benutzt werden.

Aus Geschwindigkeitsgründen überprüft C(++) die Array-Grenzen (engl. range checking) standardmäßig nicht, was ein Grund für viele Sicherheitsprobleme ist. Ein fehlerhafter Zugriff auf das Element $n + 1$ eines Felds der Größe n kann zweierlei bewirken: ein Zugriffsfehler tritt auf oder, viel schlimmer, andere Daten werden beim Schreibzugriff überschrieben, und der Fehler ist nicht nachvollziehbar.

2 Die Sprache

Jedes Java-Programm besteht aus einer Reihe von Quelldateien mit der Erweiterung `.java`¹. Diese werden vom Compiler in Bytecode übersetzt und in `.class`-Dateien gespeichert, die dann vom Interpreter ausgeführt werden können. Java ist eine vollständig objektorientierte Sprache, und zu jeder Klasse wird eine eigene `.class`-Datei angelegt.

2.1 Lexikalische Elemente

2.1.1 Kommentare

Es gibt in Java drei Arten von Kommentaren:

1. Einzeilige Kommentare beginnen mit `//` und enden am Ende der aktuellen Zeile.
2. Mehrzeilige Kommentare beginnen mit `/*` und enden mit `*/`. Sie können sich über mehrere Zeilen erstrecken.
3. Dokumentationskommentare beginnen mit `/**` und enden mit `*/` und können sich ebenfalls über mehrere Zeilen erstrecken.

2.1.2 Bezeichner

Ein Bezeichner ist eine Sequenz von Zeichen, die dazu dient, die Namen von Variablen, Klassen oder Methoden zu spezifizieren. Ein Bezeichner in Java kann beliebig lang sein, und alle Stellen sind signifikant. Bezeichner müssen mit einem Unicode-Buchstaben beginnen (das sind die Zeichen 'A' bis 'Z', 'a' bis 'z', '_' und '\$') und dürfen dann weitere Buchstaben oder Ziffern enthalten.

¹ Dies gilt nicht für die Microsoft J# Entwicklungsumgebung. Dort haben die Quellcodedateien die Dateinamenerweiterung `.jsl` und es wird eine unter Windows ausführbare Datei (exe) erstellt

Namenskonventionen

In Java gibt es mehrere verbindliche Namenskonventionen.

1. Klassen fangen mit einem Großbuchstaben an.
2. Attribute fangen mit einem Kleinbuchstaben an.
3. Methoden fangen mit einem Kleinbuchstaben an, gefolgt von einer Klammer.
4. Konstruktoren fangen mit einem Großbuchstaben an, gefolgt von einer Klammer.

2.1.3 Literale

Ein Literal ist ein konstanter Ausdruck. Es gibt verschiedene Typen von Literalen:

1. Die Wahrheitswerte `true` und `false`
2. Integrale Literale für Zahlen, etwa `122`
3. Zeichenliterals, etwa `„X2“` oder `„\n“`
4. Gleitkommalliterals: `12.567` oder `9.999E-2`
5. Stringliterals für Zeichenketten wie `„Paolo Pinkas“`

2.1.4 Token

Ein *Token* ist eine lexikalische Einheit, die dem Compiler die Bausteine des Programms liefert. Der Compiler erkennt an der Grammatik einer Sprache, welche Folgen von Zeichen ein Token bilden. Für Bezeichner heißt dies beispielsweise: Nimm die nächsten Zeichen, solange auf einen Buchstaben nur Buchstaben oder Ziffern folgen. Eine Zahl wie `1982` bildet zum Beispiel ein Token durch folgende Regel: Lies so lange Ziffern, bis keine Ziffer mehr folgt.

2.2 Anweisungen

Java zählt zu den *imperativen Programmiersprachen*², in denen der Programmierer die Abarbeitungsschritte seiner Algorithmen durch Anweisungen (engl. statements) vorgibt. Anweisungen können unter anderem sein:

1. Ausdrucksanweisungen etwa für Zuweisungen oder Funktionsaufrufe
2. Fallunterscheidungen (zum Beispiel mit *if*)
3. Schleifen für Wiederholungen (etwa mit *for* oder *do-while*)

2.3 Alles ist Klasse

Programme setzen sich aus Anweisungen zusammen. In Java können jedoch nicht einfach Anweisungen in eine Datei geschrieben und dem Compiler übergeben werden. Java erlaubt eine Programmlogik ausserhalb von Klassen nicht, deshalb muss jedweder Programmcode in einer Klasse „verpackt“ sein:

```
public class Main
{
    public static void main( String[] args )
    {
        // Hier ist der Anfang unserer Programme
        // Jetzt ist hier Platz für unsere eigenen Anweisungen
        // Hier enden unsere Programme
    }
}
```

Aus diesem Grund wird mit dem Schlüsselwort *class* eine Klasse *Main* deklariert, um später eine Funktion mit der Programmlogik anzugeben. Der Klassenname darf grundsätzlich beliebig sein, doch besteht die Einschränkung, dass in einer mit *public* deklarierten Klasse der Klassenname so lauten muss wie der Dateiname. Alle Schlüsselwörter in Java beginnen mit Kleinbuchstaben und Klassennamen üblicherweise mit Großbuchstaben.

In den geschweiften Klammern der Klasse folgen Deklarationen von Methoden, also Funktionen, die die Klasse anbietet. Eine Funktion ist eine Sammlung von Anweisungen unter einem Namen.

²Es gibt auch Programmiersprachen, die zu einer Problembeschreibung selbstständig eine Lösung finden (Prolog). Auch die Datenbanksprache SQL ist keine imperative Programmiersprache, denn wie das Datenbankmanagement-System zu unserer Anfrage die Ergebnisse ermittelt, müssen und können wir weder vorgeben noch sehen.

Die Funktion *main()* ist für die Laufzeitumgebung eine besondere Funktion, denn beim Aufruf des Java-Interpreters mit einem Klassennamen wird diese Funktion als Erstes ausgeführt. Die *main*-Funktion ist der Einsprungspunkt eines Java-Programmes. Sie wird üblicherweise wie folgt deklariert:

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

Die Parameter, mit der ein Programm aufgerufen wird, werden in dem String Array *args* übergeben.

2.4 Modifizierer

Die Deklaration einer Klasse oder Methode kann einen oder mehrere *Modifizierer* (engl. modifier) enthalten, die zum Beispiel die Nutzung einschränken oder parallelen Zugriff synchronisieren.

Der Modifizierer *public* ist ein Sichtbarkeitsmodifizierer. Er bestimmt, ob die Klasse beziehungsweise die Funktion für andere sichtbar ist oder nicht. Der Modifizierer *static* zwingt den Programmierer nicht dazu, vor dem Methodenaufruf ein Objekt der Klasse zu bilden. Anders gesagt: Die Eigenschaft, ob sich eine Methode nur über ein konkretes Objekt aufrufen lässt oder eine Eigenschaft der Klasse ist, so dass für den Aufruf kein Objekt der Klasse nötig wird, bestimmt dieser Modifizierer. Siehe dazu auch Kapitel *Zugriffskontrolle*, Seite 23.

2.5 Anweisungen und Blöcke

Atomare, also unteilbare Anweisungen, heißen auch *elementare Anweisungen*. Zu ihnen zählen zum Beispiel Funktionsaufrufe, Variablendeklarationen oder die leere Anweisung, die nur aus einem Semikolon besteht. Programme bestehen in der Regel aus mehreren Anweisungen, die eine Anweisungssequenz ergeben. Die Laufzeitumgebung von Java führt jede einzelne Anweisung der Sequenz in der angegebenen Reihenfolge hintereinander aus.

Ein *Block* fasst eine Gruppe von Anweisungen zusammen, die hintereinander ausgeführt werden. Anders gesagt: Ein Block ist eine Anweisung, die in

den geschweiften Klammern eine Folge von Anweisungen zu einer neuen Anweisung zusammenfasst. Ein Block kann überall dort verwendet werden, wo auch eine einzelne Anweisung stehen kann. Der neue Block hat jedoch eine Besonderheit für Variablen, da er einen lokalen Bereich für die darin befindlichen Anweisungen inklusive der Variablen bildet.

2.6 Datentypen

Die Datentypen in Java zerfallen in zwei Kategorien: *primitive Typen* und *Referenztypen*. Die einfachen Typen sind die eingebauten Datentypen, die nicht als Objekte verwaltet werden. Referenztypen gibt es in drei Ausführungen: Klassen-Typen, Schnittstellen-Typen (auch Interface-Typen genannt) und Feld-Typen (auch Array-Typen genannt).

2.6.1 Primitive Datentypen

Alle primitiven Datentypen in Java haben eine feste Länge, die von den Designern der Sprache ein- für allemal verbindlich festgelegt wurde. Ein *sizeof*-Operator, wie er in C vorhanden ist, wird in Java daher nicht benötigt und ist auch nicht vorhanden.

Name	Länge in Byte	Wertebereich
boolean	1	true, false
char	2	Alle Unicode-Zeichen
byte	1	-27...27-1
short	2	-215...215-1
int	4	-231...231-1
long	8	-263...263-1
float	4	3.40282347 ³⁸
double	8	1.79769313486231570 ³⁰⁸

Tab. 2.1: Primitive Datentypen

2.6.2 Arrays

Ein Array ist eine Gruppe von Variablen gleichen Typs, die über einen gemeinsamen Namen angesprochen werden. Es können Arrays von allen Ty-

pen angelegt werden. Auf ein bestimmtes Element innerhalb eines Arrays wird über einen Index zugegriffen. Arrays bieten so eine einfache Möglichkeit, gleichartige Informationen zu gliedern.

Ein Array wird in zwei Schritten angelegt. Als erstes eine Variable des gewünschten Array-Typs angelegt. Anschliessend muss der Array-Variablen mit *new* Speicherplatz zugewiesen werden. Java kennt nur dynamische Arrays.

Eindimensionale Arrays

Die allgemeine Syntax zur Deklaration eines eindimensionalen Arrays lautet:

```
Typ Variablenname[ ];
```

Mit *Typ* wird der Grundtyp des Arrays definiert. Dieser Grundtyp legt den den Datentyp jedes Array-Elementes fest. Wurde die Array-Variable deklariert, muss dem Array mit dem Schlüsselwort *new* Speicherplatz zugewiesen werden:

```
int intArray[];
intArray = new int[32];
```

In diesem Beispiel wird ein Array mit der Bezeichnung *intArray* deklariert und Speicherplatz für 32 Elemente zugewiesen.

Nachdem dem Array Speicherplatz zugewiesen wurde, kann über die Angabe des Index in eckigen Klammern auf ein bestimmtes Element zugegriffen werden. Alle Array-Indizes beginnen mit Null.

Arrays können schon bei der Deklaration initialisiert werden. Eine Array-Initialisierung besteht aus einer Liste durch Komma getrennter Ausdrücke, die in geschweiften Klammern steht. Das Array wird automatisch so angelegt, dass es groß genug zur Aufnahme der bei der Initialisierung angegebenen Anzahl der Elemente ist.

```
int intx[] = {1, 2, 3, 4, 5};
```

2.6.3 Zeichenketten - Strings

Strings sind Objekte, werden aber von Java besonders behandelt, so dass man sie wie primitive Datentypen verwenden kann.

2.6.4 Variablendeklarationen

Mit Variablen lassen sich Daten speichern, die vom Programm gelesen und geschrieben werden können. Um Variablen zu nutzen, müssen sie deklariert werden. Die Schreibweise einer Variablendeklaration ist immer die gleiche: Hinter dem Typnamen folgt der Name der Variablen. Sie ist eine Anweisung und wird daher mit einem Semikolon abgeschlossen.

Eine Variablen-Deklaration kann in jeden Block geschrieben werden.

```
int counter;
double income;
char c;
boolean isDefined
```

Oder mit Initialisierung:

```
int age = 33;
double bodyHeight = 183;
boolean vegetarian = true;
String s = "Dies ist eine Zeichenkette.";
```

2.6.5 Sichtbarkeit und Gültigkeitsbereich

In jedem Block und auch in jeder Klasse können Variablen deklariert werden. Jede Variable hat einen Gültigkeitsbereich (engl. scope). Sie ist nur in dem Block „sichtbar“, in dem sie deklariert wurde. In dem Block ist die Variable lokal.

Innerhalb eines Blocks können Variablennamen nicht genauso gewählt werden wie Namen lokaler Variablen eines äußeren Blocks oder wie die Namen für die Parameter einer Funktion.

2.6.6 Umwandeln von Datentypen

Eingabe in einen String umwandeln

Um eine numerische Eingabe in einen String umzuwandeln stellt die String Klasse die `valueOf` Methode bereit:

```

// Ganzzahl
String s1 = String.valueOf( 10 );           // 10
// Fließkommazahl
String s2 = String.valueOf( Math.PI );     // 3.141592653589793
// Wahrheitswert
String s3 = String.valueOf( 1 < 2 );     // true
// Datum
String s4 = String.valueOf( new Date() );  // Mon Feb 06 14:40:38
CEST 2006

```

Umwandlung einer Zeichenkette in einen elementaren Datentyp

Zum Parsen eines Strings - in zum Beispiel eine Ganzzahl - ist nicht die Klasse String verantwortlich, sondern spezielle Klassen, die für jeden primitiven Datentyp vorhanden sind. Die Klassen deklarieren statische parseXXX()-Methoden, wie Tabelle 2.2 zeigt:

Klasse	Konvertierungsmethode
java.lang.Boolean	parseBoolean(String s)
java.lang.Byte	parseByte(String s)
java.lang.Short	parseShort(String s)
java.lang.Integer	parseInt(String s)
java.lang.Long	parseLong(String s)
java.lang.Double	parseDouble(String s)
java.lang.Float	parseFloat(String s)

Tab. 2.2: Datentyp Konvertierungsmethoden

3 Operatoren

3.1 Arithmetische Operatoren

Operator	Resultat
+	Addition
-	Subtraktion, negatives Vorzeichen
*	Multiplikation
/	Division
%	Restwert
++	Inkrement
--	Dekrement

Tab. 3.1: Arithmetische Operatoren

3.2 Bitweise Operatoren

Operator	Resultat
~	Einerkomplement (bitweise invertierung, NOT)
&	UND
	ODER
^	exklusiv ODER

Tab. 3.2: Logische bitweise Operatoren

Operator	Resultat
>>	Verschiebung nach rechts
>>>	Verschiebung nach rechts, auffüllen mit Nullen
<<	Verschiebung nach links

Tab. 3.3: Bitweise Schiebeoperatoren

3.3 Relationale Operatoren

Operator	Resultat
==	gleich
!=	ungleich
>	größer als
<	kleiner als
>=	größer als oder gleich
<=	kleiner als oder gleich

Tab. 3.4: Relationale Operatoren

3.4 Boole'sche Operatoren

Operator	Resultat
&	logisches UND
	logisches ODER
^	exklusives ODER
	ODER mit Short-Circuit-Evaluation
&&	UND mit Short-Circuit-Evaluation
!	logisches NICHT
==	gleich
!=	ungleich
? :	freifaches if-then-else

Tab. 3.5: Boole'sche Operatoren

Bei der *Short-Circuit-Evaluation* wertet Java den rechten Operator nicht aus, wenn das Ergebnis des Ausdrucks allein über den linken Operator bestimmt werden kann.

3.5 Zuweisungsoperator

Als Zuweisungsoperator dient das einfache Gleichheitszeichen (=):

```
var = Ausdruck;
```

Der Typ *var* muss mit dem Typ *Ausdruck* kompatibel sein. Java erlaubt auch eine Kurzzuweisung:

```
var = var op Ausdruck;
```

Solche eine Anweisung lässt sich noch weiter verkürzen:

```
var op = Ausdruck;
```

Siehe dazu Tabelle 3.6.

Operator	Resultat
+ =	Additionszuweisung
- =	Subtraktionszuweisung
* =	Multiplikationszuweisung
/ =	Divisionszuweisung
% =	Restwertzuweisung
& =	bitweise UND-Zuweisung
=	bitweise ODER-Zuweisung
^ =	bitweise exklusive ODER-Zuweisung
>>=	Zuweisung über Verschiebung nach rechts
>>>=	Zuweisung über Verschiebung nach rechts, mit Nullen auffüllen
<<=	Zuweisung über Verschiebung nach links
& =	UND-Zuweisung
=	ODER-Zuweisung
^ =	XODER-Zuweisung

Tab. 3.6: Kurzzuweisungsoperatoren

4 Kontrollstrukturen

Kontrollstrukturen dienen in einer Programmiersprache dazu, Programmteile unter bestimmten Bedingungen auszuführen. Java bietet zum Ausführen verschiedener Programmteile eine if- und if/else-Anweisung sowie die switch-Anweisung. Neben der Verzweigung dienen Schleifen dazu, Programmteile mehrmals auszuführen.

4.1 Verzweigungen

4.1.1 if

Die if-Anweisung besteht aus dem Schlüsselwort if, dem zwingend ein Ausdruck mit dem Typ boolean in Klammern folgt. Es folgt eine Anweisung, die oft eine Blockanweisung ist.

```
if (age == 14)
{
    // weitere Anweisungen
}
```

Die weitere Abarbeitung der Anweisungen hängt vom Ausdruck im if ab. Ist das Ergebnis des Ausdrucks wahr (*true*), wird die Anweisung ausgeführt; ist das Ergebnis des Ausdrucks falsch (*false*), so wird mit der ersten Anweisung nach der if-Anweisung fortgefahren. Hinter dem if und der Bedingung erwartet der Compiler eine Anweisung. Sind mehrere Anweisungen in Abhängigkeit von der Bedingung auszuführen, ist ein Block zu setzen; andernfalls ordnet der Compiler nur die nächstfolgende Anweisung der Fallunterscheidung zu.

Das optionale Schlüsselwort else veranlasst die Ausführung der alternativen Anweisung, wenn der Test falsch ist:

```
if ( x < y )
    System.out.println( "x ist echt kleiner als y." );
else
    System.out.println( "x ist größer oder gleich y." );
```

4.1.2 switch

Ein Ersatz für mehrere *if*-Anweisungen ist die *switch*-Anweisung:

```
switch ( op )
{
    case '+':
        System.out.println( x + y );
        break;
    case '-':
        System.out.println( x - y );
        break;
    case '*':
        System.out.println( x * y );
        break;
    case '/':
        System.out.println( x / y );
        break;
    default
        System.out.println("Unbekannter Operator");
}
```

Jeder Anweisungsblock eines *case*-Zweiges muss mit *break* abgeschlossen werden, wenn das Programm nicht auch den folgenden *case*-Zweig abarbeiten soll. Wird die *break* Anweisung absichtlich weggelassen, spricht man von einem „Fall-Through“. Dieser sollte im Quellcode auch gesondert kommentiert werden, damit man sieht, dass dies Absicht ist und kein *break* vergessen wurde vom Programmierer.

Der optionale *default*-Zweig wird immer dann ausgeführt, wenn keiner der vorangegangenen Zweige mit seiner Bedingung zu trifft.

Die *switch*-Anweisung hat allerdings zwei Nachteile: Der Ausdruck ist auf einen Ganzzahldatentyp beschränkt. Datentypen wie *long* oder Fließkommadatentypen sind nicht möglich. Auch kann man keine Objekte für den Ausdruck einsetzen und somit auch keine Strings. Desweiteren ist es nicht möglich Bereiche abzudecken. Will man Bereiche abdecken, muss man auf die *if*-Anweisung zurückgreifen.

4.2 Schleifen

Schleifen dienen dazu, bestimmte Anweisungen immer wieder abzuarbeiten. Zu einer Schleife gehören die Schleifenbedingung (Schleifenkopf) und der

Rumpf. Die Schleifenbedingung, ein Boolescher Ausdruck, entscheidet darüber, unter welcher Bedingung die Wiederholung ausgeführt wird. In Abhängigkeit von der Schleifenbedingung kann der Rumpf mehrmals ausgeführt werden. Dazu wird bei jedem Schleifendurchgang die Schleifenbedingung geprüft. Das Ergebnis entscheidet, ob der Rumpf ein weiteres Mal durchlaufen (*true*) oder die Schleife beendet wird (*false*). Java bietet vier Typen von Schleifen:

- while-Schleife
- do-while-Schleife
- Einfache for-Schleife
- Erweiterte for-Schleife

4.2.1 while-Schleife

Die while-Schleife ist eine abweisende Schleife¹, die vor jedem Schleifeneintritt die Schleifenbedingung prüft. Ist die Bedingung wahr, führt sie den Rumpf aus, andernfalls beendet sie die Schleife. Wie bei *if* muss auch bei den Schleifen der Typ der Bedingungen *boolean* sein.

```
int cnt = 12;
while ( cnt > 0 )
{
    System.out.println( cnt );
    cnt--;
}
```

Vor jedem Schleifendurchgang wird der Ausdruck neu ausgewertet, und ist das Ergebnis *true*, so wird der Rumpf ausgeführt. Die Schleife ist beendet, wenn das Ergebnis *false* ist. Ist die Bedingung schon vor dem ersten Eintritt in den Rumpf nicht wahr, so wird der Rumpf erst gar nicht durchlaufen. Der Typ der Bedingung muss *boolean* sein.

4.2.2 do-while-Schleife

Dieser Schleifentyp ist eine annehmende Schleife², da do-while die Schleifenbedingung erst nach jedem Schleifendurchgang prüft. Bevor es zum ersten Test kommt, ist der Rumpf also schon einmal durchlaufen worden:

¹Kopf gesteuert

²Fuß gesteuert

```
int pos = 1;
do
{
    System.out.println( pos );
    pos++;
} while ( pos <= 10 );
```

Es ist wichtig, auf das Semikolon hinter der while-Anweisung zu achten. Liefert die Bedingung ein *true*, so wird der Rumpf erneut ausgeführt. Andernfalls wird die Schleife beendet, und das Programm wird mit der nächsten Anweisung nach der Schleife fortgesetzt.

4.2.3 for-Schleife

Die for-Schleife ist eine spezielle Variante einer while-Schleife und wird typischerweise zum Zählen benutzt. Genauso wie while-Schleifen sind for-Schleifen abweisend, der Rumpf wird also erst dann ausgeführt, wenn die Bedingung wahr ist.

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

Der Schleifenkopf einer *for*-Schleife besteht aus drei Teilen:

1. *Initialisierung der Zählvariablen / Laufvariable*: Der erste Teil der *for*-Schleife ist ein Ausdruck wie $i = 1$, der vor der Durchführung der Schleife genau einmal ausgeführt wird. Der erste Teil kann lokale Variablen deklarieren und initialisieren. Diese Zählvariable ist dann außerhalb des Blocks nicht mehr gültig.
2. *Schleifenbedingung*: Der mittlere Teil, wie $i \leq 10$, wird vor dem Durchlaufen des Schleifenrumpfs - also vor jedem Schleifeneintritt - getestet. Ergibt der Ausdruck *false*, wird die Schleife nicht durchlaufen und beendet. Das Ergebnis muss, wie bei einer while-Schleife, vom Typ *boolean* sein. Ist kein Test angegeben, so ist das Ergebnis automatisch *true*.
3. *Schleifen-Inkrement*: Der letzte Teil, wie $i++$, wird immer am Ende jedes Schleifendurchlaufs, aber noch vor dem nächsten Schleifeneintritt ausgeführt. Das Ergebnis wird nicht weiter verwendet. Ergibt die Bedingung des Tests *true*, dann befindet sich beim nächsten Betreten des Rumpfs der veränderte Wert im Rumpf.

for-Schleifen sollten immer dann benutzt werden, wenn eine Variable um eine konstante Größe erhöht wird. Tritt in der Schleife keine Schleifenvariable auf, die inkrementiert oder dekrementiert wird, sollte eine while-Schleife genutzt werden. Eine do-while-Schleife sollte dann ihren Einsatz finden, wenn die Abbruchbedingung erst am Ende eines Schleifendurchlaufs ausgewertet werden kann. Auch sollte die for-Schleife dort eingesetzt werden, wo sich alle drei Ausdrücke im Schleifenkopf auf dieselbe Variable beziehen. Vermieden werden sollten unzusammenhängende Ausdrücke im Schleifenkopf. Der Zugriff auf die Schleifenvariable im Rumpf ist eine schlechte Idee, wenn sie auch gleichzeitig im Kopf modifiziert wird - das ist schwer zu durchschauen.

4.2.4 *break* und *continue*

Mit *break* kann man einen Block und somit auch eine Schleife vorzeitig verlassen. Das Programm wird dann mit dem, der Schleife folgenden Code, fortgesetzt.

Mit *continue* kann man von einer beliebigen Stelle im Schleifenrumpf zum Schleifenkopf, und somit zur Abfrage der Bedingung, springen.

5 Klassen und Methoden

5.1 Klassengrundlagen

Eine Klasse ist eine *Vorlage* für ein Objekt, welches seinerseits eine *Instanz* einer Klasse bildet. Bei der Definition einer Klasse werden deren genaue Form und ihre Eigenschaften festgelegt. Dazu enthält eine Klasse Daten und Methoden, um diese Daten zu manipulieren. Es gibt auch Klassen, die nur Daten enthalten und als Datencontainer fungieren und Klassen, die nur Methoden enthalten. Meist enthalten Klassen allerdings beides, Daten und Methoden.

Eine Klasse wird mit dem Schlüsselwort *class* deklariert. das Format einer Klassendefinition sieht dann wie folgt aus:

```
class Klassenname
{
    Typ Instanzvariable1;
    Typ Instanzvariable2;
    // ...
    Typ InstanzvariableN;

    Typ Methodenname1(Parameterliste)
    {
        // Implementation der Methode
    }

    Typ Methodenname2(Paramterliste)
    {
        // Implementation der Methode
    }

    Typ MethodennameN(Parameterliste)
    {
        // Implementation der Methode
    }
}
```

Die innerhalb einer Klasse definierten Variablen werden *Instanzvariablen*, weil sie Variablen der Instanz sind genannt, oder auch *Felder* (Delphi) oder

Attribute. Der *Typ* gibt den Datentyp einer Instanzvariable bzw. den Rückgabetyt einer Methode an. Gibt eine Methode keinen Wert zurück muss als Rückgabetyt *void* (leer) angegeben werden. In der Regel greift man über Methoden auf die Instanzvariablen zu. Diese Methoden werden als *Getter* und *Setter* bezeichnet. Dies hat den Vorteil, dass man die Daten noch validieren kann bevor sie genutzt werden. *Parameter*¹ sind Variablen, die beim Aufruf der Methode als *Argumente* übergeben werden und so innerhalb der Methode zur Verfügung stehen. Werden keine Parameter übergeben, bleibt die Klammer leer. Beispiel:

```
class Beispiel
{
    int a, b;

    int getA()
    {
        return a;
    }

    void setA(int A)
    {
        a = A;
    }

    int getB()
    {
        return b;
    }

    void setB(int B)
    {
        b = B;
    }

    int sum()
    {
        return a + b;
    }
}
```

¹Argument ist ein auch häufig verwendeter Begriff für Parameter.

5.2 Klassenobjekte deklarieren und erzeugen

Um eine Klasse nutzen zu können, muss Instanz dieser Klasse erzeugt und in einer Variablen gespeichert werden über die man dann auf die Daten und Methoden der Klasse zugreifen kann. Eine neue Instanz einer Klasse wird mit dem Schlüsselwort *new* angelegt. *new* weist einem Objekt dynamisch Speicherplatz zu und gibt einen Verweis auf das Objekt zurück. Bei diesem Verweis handelt es sich mehr oder weniger um die Speicheradresse, an der das erzeugte Objekt liegt. Beispiel:

```
// Deklaration der Variablen für den Objektverweis
Beispiel bsp;
// erzeugen des Objektes
bsp = new Beispiel();

// oder zusammengefasst:
Beispiel bsp = new Beispiel();
```

Nachdem ein Klassenobjekt erzeugt wurde, kann mit dem Punktoperator (.) auf die Methoden und Instanzvariablen zugegriffen werden. Beispiel:

```
bsp.sum(40, 2);
```

5.3 Konstruktoren

Konstruktoren sind besondere Methoden. Sie werden direkt bei der Erzeugung aufgerufen und können dazu genutzt werden Instanzvariablen zu initialisieren oder andere Aufgaben auszuführen, um das Objekt vollständig zu initialisieren und dessen internen Status zu setzen. Konstruktoren haben grundsätzlich den gleichen Namen wie die Klasse. Konstruktoren können auch, wie andere Methoden, eine Parameterliste haben. Überlicherweise werden die Parameter des Konstruktors dazu genutzt Instanzvariablen zu initialisieren. Allerdings haben sie im Gegensatz zu „normalen“ Methoden keinen Rückgabtyp auch nicht *void*. Das liegt daran, dass der implizite Rückgabtyp eines Klassenkonstruktors der Klassentyp selber ist. Beispiel:

```
// Klassen
class Beispiel
{
    int a, b;

    // Konstruktor mit Parameterliste
```

```

    Beispiel(int x, int y)
    {
        a = x;
        b = y;
    }

    int sum()
    {
        return a + b;
    }
}

// Aufruf
Beispiel bsp = new Beispiel(40, 2);
System.out.println(bsp.sum());
// Ausgabe
42

```

Wird kein Konstruktor definiert, erzeugt Java automatisch einen Standardkonstruktor, der automatisch alle Instanzvariablen mit *Null* initialisiert.

Da Java eine Garbage Collection (siehe Seite 1) besitzt, gibt es keine Destruktoren wie in anderen Programmiersprachen. Da aber in manchen Fällen sichergestellt werden muss, dass noch bestimmter Code ausgeführt wird (freigeben von Ressourcen), bevor ein Objekt zerstört wird, gibt es die Methode *finalize()*. Diese Methode wird nur von dem Garbage Collector aufgerufen und nicht unbedingt, wenn das Objekt seine Gültigkeit verliert. Das bedeutet, dass unbekannt ist, wann die Methode *finalize()* ausgeführt wird.

5.4 Methoden überladen

Von *überladenen* Methoden spricht man dann, wenn in einer Klasse mehrere Methoden mit identischer Bezeichnung aber unterschiedlichen Parameterlisten definiert wurden. Wird eine überladene Methode aufgerufen, dann verwendet Java den Typ und die Anzahl der Argumente als Entscheidungshilfe dafür, welche der überladenen Methoden aufgerufen werden soll.

5.5 Argumentenübergabe

Argumente können an eine Methode auf zwei Weisen übergeben werden: einmal als *Aufruf nach Wert*² oder *Aufruf über einen Verweis*³. Bei Aufruf nach Wert werden die Parameter kopiert. So dass, wenn sie innerhalb der Methode verändert werden, dies keine Auswirkung auf die ursprünglich übergebenen Argumente hat. Erfolgt der Aufruf über einen Verweis, so wird die Adresse eines Speicherbereiches übergeben, der das Objekt enthält. Das heißt, wird der Wert des Verweises geändert, ändert er sich auch in der aufrufenden Methode, da ja nur ein Verweis auf einen Speicherbereich übergeben wurde und nicht dessen Inhalt.

Wird einer Methode ein einfacher Typ übergeben, ruft Java die Methode mit Aufruf nach Wert auf. Wird einer Methode ein Objekt übergeben, erfolgt der Aufruf mit Aufruf über eine Referenz, da Variablen, die ein Objekt enthalten, ja nur Verweise auf eine Speicheradresse sind.

5.6 Der Modifizierer *static*

Methoden oder Instanzvariablen, die mit dem Modifizierer *static* versehen sind, können direkt angesprochen werden, ohne dass ein Instanz der Klasse existiert. Die gebräuchlichste Verwendung ist die deklaration der *main()*-Methode (Siehe 5).

Um Klassenmethoden oder -instanzen zu nutzen erfolgt der Aufruf über den Klassennamen und dem Namen der entsprechenden Methode:

```
Klassenname.methode();
```

5.7 Zugriffskontrolle

Man kann mit Sichtbarkeitsmodifizierern den Zugriff auf Klassen, Methoden und Instanzvariablen kontrollieren. So kann man zum Beispiel eine Methoden nach aussen hin „unsichtbar“ machen. Sie ist somit nur innerhalb der eigenen Klasse sichtbar und man kann sie nicht von ausserhalb der Klasse aufrufen. Java kennt drei Zugriffsangaben:

²engl.: call by value

³engl.: call by reference

1. *private*: Eine *private*-Methode, -Variable oder -Klasse ist ausserhalb der, in der sie definiert ist, nicht sichtbar.
2. *protected*: Methoden oder Variablen vom Typ *protected* sind in der aktuellen Klasse und in abgeleiteten Klassen sichtbar. Darüber hinaus sind sie für Methoden anderer Klassen innerhalb desselben Pakets sichtbar (das ist ein wichtiger Unterschied beispielsweise zu C++). Sie sind jedoch nicht für Aufrufer der Klasse sichtbar, die in anderen Paketen definiert wurden.
3. *public*: Eine mit *public* deklarierte Methode, Variable oder Klasse ist öffentlich und nach aussen hin sichtbar. Sie kann also von einer anderen Klasse aus aufgerufen werden.

6 Vererbung

In Java kann eine neue Klasse die Eigenschaften einer anderen Klasse erben. Man spricht dann von einer abgeleiteten Klasse oder auch Unterklasse. Die Klasse von der geerbt wird, wird in Java als Superklasse bezeichnet. Eine abgeleitete Klasse erbt alle für die Superklasse definierten Instanzvariablen und Methoden. Java kennt keine Mehrfachvererbung, d. h. eine Klasse kann immer nur von einer Klasse abgeleitet werden.

Eine Klasse wird vererbt, indem die Definition der einen Klasse über das Schlüsselwort *extends* in die andere Klasse eingebunden wird:

```
class Unterklasse extends Superklasse
{
}
```

Der folgende Code zeigt ein Beispiel für Vererbung. Gegeben seien die Klassen *SuperKlasse* und *UnterKlasse*:

```
public class SuperKlasse
{
    public int add(int a, int b)
    {
        return a + b;
    }
}

public class UnterKlasse extends SuperKlasse
{
    public int sub(int a, int b)
    {
        return a - b;
    }
}
```

Der folgende Code zeigt nun, wie in der Unterklasse auf eine Methode der Superklasse zugegriffen werden kann:

```
SuperKlasse sk = new SuperKlasse();
UnterKlasse uk = new UnterKlasse();
int c = 0;
```

```

System.out.println("Addition Superklasse:");
c = sk.add(40, 2);
System.out.println(c);

System.out.println("Subtraktion Unterklasse:");
c = uk.sub(42, 2);
System.out.println(c);

System.out.println("geerbte Addition Unterklasse:");
c = uk.add(40, 2);
System.out.println(c);

```

Dies wird schon bei der Programmierung in der Codevervollständigung deutlich. In der Methodenauswahl der Unterklasse wird einem auch die Methode *add* der Superklasse angeboten (siehe Grafik 6.1).

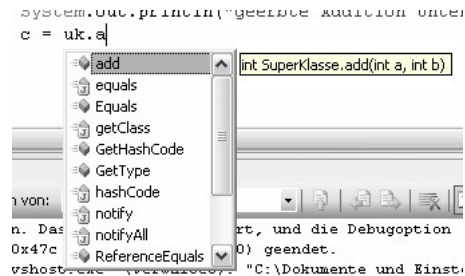


Abb. 6.1: Codevervollständigung Methodenauswahl der Unterklasse

6.1 Zugriff auf die Superklasse

Will man innerhalb der Unterklasse auf eine Instanzvariable oder Methode der Superklasse zugreifen, so geht das mittels des Schlüsselwortes *super*:

```

public class SuperKlasse
{
    String text = "Attribut der Superklasse.";
    public int add(int a, int b)
    {
        return a + b;
    }
}

```

```

public class UnterKlasse extends SuperKlasse
{
    public int sub(int a, int b)
    {
        return a - b;
    }

    public void showTextSuperKlasse()
    {
        System.out.println(super.text);
    }
}

```

Den Konstruktor der Superklasse ruft man auf, indem man das Schlüsselwort *super* mit den Parametern des Konstruktors der Superklasse aufruft:

```

super(Parameter1, Parameter2);

```

Hat die abgeleitete Klasse einen eigenen Konstruktor, so muss als aller erste Anweisung in diesem Konstruktor, der Konstruktor der Superklasse mit *super* aufgerufen werden.

6.2 Methoden überschreiben

Methoden lassen sich in einer Unterklasse überschreiben. Man kann in einer Unterklasse eine Methode mit der gleichen Bezeichnung implementieren, wie eine Methode, die schon in der Superklasse existiert. Dabei wird die Methode der Superklasse verborgen.

6.3 Abstrakte Klassen

Als abstrakte Klasse bezeichnet man eine Klasse, die zwar Methodendeklarationen enthält, diese aber nicht selber implementiert, also leer sind. Abstrakte Methoden werden mit dem Schlüsselwort *abstract* deklariert:

```

abstract Typ name(Parameterliste);

```

Es folgt kein Methodenrumpf, da eine abstrakte Methode von der abgeleiteten Klasse implementiert werden muss. Enthält eine Klasse mindestens eine abstrakte Methode, so muss die ganze Klasse als abstrakt deklariert werden:

```
abstract class Klassenname
{
}

```

6.3.1 Warum abstrakte Klassen?

Abstrakte Klassen stellen einen Prototypen oder eine Basisklasse dar, worauf die abgeleiteten Klassen dann aufbauen. Durch abstrakte Methoden wird den abgeleiteten Klassen mitgeteilt, welche Methoden sie noch selber implementieren muss. Abstrakte Methoden sind dann sinnvoll, wenn abgeleitete Klassen identische Eigenschaften haben, die sich aber unterschiedlich verhalten. So haben geometrische Figuren die gemeinsame Eigenschaft *Fläche*. Die Berechnung der Fläche hängt jedoch von der geometrischen Figur ab. In der Basisklasse wird die Methode zur Berechnung der Fläche als abstrakt deklariert, da sie nicht sinnvoll implementiert werden kann, da sich die Fläche der unterschiedlichen geometrischen Körper auch unterschiedlich errechnet.

Abstrakte Klassen kann man nicht instanzieren in Java, da sie Methoden besitzen, die nicht implementiert sind und somit ohne Funktion. Solch eine Klasse mit leeren Methoden zu instanzieren, wäre nicht sinnvoll. Ist eine Methode der Klasse abstrakt gekennzeichnet, so muss die ganze Klasse als abstrakt gekennzeichnet werden. Im Gegensatz zu Interfaces kann eine abstrakte Klasse aber auch schon implementierte Methoden enthalten.

7 Ereignisse

Um Ereignisse zu erstellen sind drei Schritte notwendig. Zu erst muss ein Interface deklariert werden, welches die auszulösenden Ereignisse zur Verfügung stellt:

```

/*
 * Interface des Ereignisses
 */
interface KundeListener
{
    public void onDelKontoSuccess(Kunde source, String
        kontoBezeichner);
    public void onDelKontoError(Kunde source);
    public void onAddKonto(Kunde source);
}

```

Als nächstes muss die Klasse, welche die Ereignisse auslösen, soll Methoden implementieren, um die Ereignismethoden zu registrieren:

```

/*
 * Methode um Ereignismethoden zu registrieren
 */
public void addKundeListener(KundeListener listener)
{
    _listeners.add(listener);
}

/*
 * Methode um Ereignismethode zu entfernen
 */
public void removeKundeListener(KundeListener listener)
{
    _listeners.add(listener);
}

```

Dann wird noch eine Methode benötigt, die alle Ereignismethoden benachrichtigt, wenn ein Ereignis ausgelöst wird:

```

private void fireDelKontoSuccessEvent(String kontoBezeichner)
{
    ListIterator iterListeners = _listeners.listIterator();
    while (iterListeners.hasNext())

```

```

    {
        KundeListener kl = (KundeListener)iterListeners.next();
        ;
        kl.onDelKontoSuccess(this, kontoBezeichner);
    }
}

```

Hat man diese drei Schritte implementiert kann man das Ereignis auslösen:

```
fireDelKontoSuccessEvent (beschreibung);
```

Will man nun auf das Ereignis reagieren in einer übergeordneten Klasse, so muss mit dem Schlüsselwort *implements* angegeben werden, dass die übergeordnete Klasse das Interface der Ereignisse implementiert:

```

public class Program implements KundeListener
\begin{lstlisting}
Dann müssen natürlich noch alle Ereignismethoden implementiert werden:
\begin{lstlisting}
/*
 * EventListener Implementationen
 */
public void onDelKontoSuccess(Kunde source, String kontoBezeichnung)
{
    System.out.println("Konto ' " + kontoBezeichnung + "' geloescht.
        ");
}

```

Wird das Ereignis nun ausgelöst, wird der Code der EventListener Methode ausgeführt. In diesem Fall wird in der Konsole ein Text ausgegeben.

8 Ausnahmebehandlung mit Exceptions

Fehler beim Programmieren sind unvermeidlich. Schwierigkeiten bereiten nur die unkalkulierbaren Situationen - hier ist der Umgang mit Fehlern ganz besonders heikel. Java bietet die elegante Methode der Exceptions, um mit Fehlern flexibel zu behandeln. Bei der Verwendung von Exceptions wird der Programmfluss nicht durch Abfrage des Rückgabestatus unterbrochen. Ein besonders ausgezeichnetes Programmstück überwacht mögliche Fehler und ruft gegebenenfalls speziellen Programmcode zur Behandlung auf. Den überwachten Programmbereich (Block) leitet das Schlüsselwort *try* ein und *catch* beendet es. Hinter *catch* folgt der Programmblock, der beim Auftreten eines Fehlers ausgeführt wird, um den Fehler abzufangen oder zu behandeln. Weitere Schlüsselwörter für die Ausnahmebehandlung sind *throw*, *throws* und *finally*.

8.1 Ausnahmen behandeln mit *try/catch*

Programmanweisungen, die auf Ausnahmen hin überwacht werden sollen, befinden sich innerhalb eines *try*-Blocks. Kommt es innerhalb dieses *try*-Blocks zu einer Ausnahme, wird eine Exceptions ausgelöst und die weitere Ausführung des folgenden Codes wird abgebrochen und direkt in den *catch*-Block gesprungen. Für jede Ausnahme, die abgefangen werden soll, muss eine *catch*-Anweisung vorhanden sein:

```
import java.io.*;

public class ReadFileWithRAF
{
    public static void main( String[] args )
    {
        try
        {
            RandomAccessFile f;
            f = new RandomAccessFile( "EastOfJava.txt", "r" );

            for ( String line; (line=f.readLine()) != null; )
                System.out.println( line );
        }
    }
}
```

```

catch ( FileNotFoundException e ) // Datei gibt es nicht
{
    System.err.println( "Datei gibt es nicht!" );
}
catch ( IOException e ) // Schreib-/Lese probleme
{
    System.err.println( "Schreib-/Lese probleme!" );
}
}
}

```

8.2 Abschlussbehandlung mit *finally*

Nach einem (oder mehreren) *catch* kann optional ein *finally*-Block folgen. Die Laufzeitumgebung führt die Anweisungen im *finally*-Block immer aus, egal, ob ein Fehler auftrat, oder die Anweisungen im *try-catch*-Block optimal durchliefen. Das heißt, der Block wird auf jeden Fall ausgeführt, auch wenn im *try-catch*-Block ein *return*, *break* oder *continue* steht oder eine Anweisung eine neue Ausnahme auslöst. Der Programmcode im *finally*-Block bekommt auch gar nicht mit, ob vorher eine Ausnahme auftrat oder alles glatt lief.

```

import java.io.*;

public class ReadFileWithRAF
{
    public static void main( String[] args )
    {
        try
        {
            RandomAccessFile f;
            f = new RandomAccessFile( "EastOfJava.txt", "r" );

            for ( String line; (line=f.readLine()) != null; )
                System.out.println( line );
        }
        catch ( FileNotFoundException e ) // Datei gibt es nicht
        {
            System.err.println( "Datei gibt es nicht!" );
        }
        catch ( IOException e ) // Schreib-/Lese probleme
        {
            System.err.println( "Schreib-/Lese probleme!" );
        }
        finally
        {
            // Datei auf alle Fälle wieder schliessen.
        }
    }
}

```

```
f.close();
    }
}
```

8.3 Nicht behandelte Ausnahmen angeben

Neben der rahmenbasierten Ausnahmebehandlung - dem Einzäunen von problematischen Blöcken durch einen *try*- und *catch*-Block - gibt es eine weitere Möglichkeit, auf Exceptions zu reagieren: Im Kopf der betreffenden Methode wird eine *throws*-Klausel eingeführt. Dadurch zeigt die Methode an, dass sie eine bestimmte Exception nicht selbst behandelt, sondern diese unter Umständen an die aufrufende Methode weitergibt. Nun kann von einer Funktion eine Exception ausgelöst werden. Die Funktion wird abgebrochen und gibt ihrerseits eine Exception zurück.

Dazu ein Beispiel: Eine Methode soll eine Datei öffnen und die erste Zeile auslesen. Der Dateiname wird als Argument der Methode übergeben. Da das Öffnen der Datei sowie das Lesen einer Zeile eine Ausnahme auslösen kann, müssen wir diese Ausnahme behandeln. Wir fangen sie jedoch nicht in einem eigenen *try*- und *catch*-Block auf, sondern leiten sie an den Aufrufer weiter. Das bedeutet, dass er sich um den Fehler kümmern muss.

```
String readFirstLineFromFile( String filename )
    throws FileNotFoundException, IOException
{
    RandomAccessFile f = new RandomAccessFile( filename, "r" );
    return f.readLine();
}
```

8.4 Ausnahmen selber auslösen

Mit Hilfe des Schlüsselwortes *throw* kann man auch selber eine Exception auslösen. dazu gibt man wieder im Funktionskopf die Ausnahme an, die man auslösen will und löst sie dann durch *throw* aus:

```
private static Konto CreateKonto(String bezeichnung, int saldo) throws
    RuntimeException
{
    Konto _konto = new Konto();
```

```

    if (_konto != null)
    {
        _konto.setBezeichnung(bezeichnung);
        _konto.setSaldo(saldo);
        return _konto;
    }
    else
    {
        throw new RuntimeException();
    }
}

```

8.5 Definition von eigene Ausnahmen

Der eingebaut Mechanismus zur Behandlung von Ausnahmen kann zwar mit den gängigsten Fehlern umgehen, doch kann es nötig werden eigene Ausnahmetypen zu definieren. dazu wird eine eigene Klasse deklariert, die von der Klasse *Exception* abgeleitet wird. Die abgeleitete Klasse muss eigentlich nichts weiter implementieren. Aber es können natürlich auch selbst definierte Eigenschaften und Methoden für den zu behandelnden Ausnahmetyp hinzugefügt werden.

```

public class BankException extends Exception
{
    public String Message;
    public BankException(String msg)
    {
        Message = msg;
    }
}

```

Diese kann man dann genauso auslösen wie schon von java implementierte Exceptions:

```

private static Konto CreateKonto(String bezeichnung, int saldo) throws
BankException
{
    Konto _konto = new Konto();
    if (_konto != null)
    {
        _konto.setBezeichnung(bezeichnung);
        _konto.setSaldo(saldo);
        return _konto;
    }
    else

```

```
        {  
            throw new BankException("Konto ist null");  
        }  
    }  
  
    // ...;  
    // ...;  
  
    Kunde kundel = new Kunde();  
    try  
    {  
        kundel = CreateCustomer("Mueller", "Emil");  
        kundel.addKonto(CreateKonto("Privatkonto", 500));  
        kundel.addKonto(CreateKonto("Firmenkonto", 2500));  
        kunden.add(kundel);  
    }  
    catch (BankException e)  
    {  
        System.out.println(e.Message);  
    }  
}
```

9 MySQL Datenbankankbindung

Der Zugriff auf eine MySQL-Datenbank ist mit Java entweder über den ODBC-Dienst¹ von Windows möglich oder mit einem separaten Treiber direkt.

9.1 Zugriff über den ODBC-Dienst von Windows

Die Verbindung zum Datenbanksystem kann über den ODBC Dienst auf Rechnern mit Microsoft Betriebssystem hergestellt werden. Hierzu wird eine Datenbank im ODBC-Dienst angemeldet und dann auf diese Datenquelle von Java aus zugegriffen.

```
import java.sql.*;

String dbTreiber = "sun.jdbc.odbc.JdbcOdbcDriver";

String dbName = "kalender";

String dbUrl = "jdbc:odbc:" + dbName;
```

Auf dem Rechner ist dann direkt ein Datenbanktreiber für das jeweilige Datenbanksystem zu installieren.

9.2 Einbindung des MySQL-Connectors in Java

Für den Zugriff von Java auf ein MySQL-Datenbanksystem wird ein Treiber bzw. Connector benötigt. Dieser Connector kann von der MySQL-Homepage bezogen werden.

¹Open Database Connectivity (ODBC, dt. etwa: „Offene Datenbank-Verbindungsfähigkeit“) ist eine standardisierte Datenbankschnittstelle unter Windows, die SQL als Datenbanksprache verwendet. ODBC bietet eine Programmierschnittstelle (API), die es einem Programmierer erlaubt, seine Anwendung relativ unabhängig vom verwendeten Datenbankmanagementsystem (DBMS) zu entwickeln, wenn dafür ein ODBC-Treiber existiert. Der Datenzugriff erfolgt nie unmittelbar auf eine Tabelle oder eine Datenbank, sondern immer über die entsprechende (ODBC-)Komponente. Mit ODBC kann auf jede lokale oder ferne Datenquelle zugegriffen werden.

Wenn der ODBC-Dienst nicht benutzt wird, stehen Klassen für den direkten Zugriff von Java auf ein MySQL-Datenbanksystem zur Verfügung. Diese Klassen sind in der Regel kompiliert in einem JAR-Archiv zusammengefasst.

Damit dieses Archiv von einer Java-Anwendung benutzt werden kann, muss das Archiv zu dem Klassenpfad (Umgebungsvariable CLASSPATH) hinzugefügt werden.

```
SET CLASSPATH=%CLASSPATH%;mysql-connector-java-3.0.17-ga-bin.jar
```

Hinzufügen des MySQL-Connector-Archivs zu einem Projekt in einem Softwareentwicklungswerkzeug am Beispiel von Eclipse:

In Eclipse wird im Kontextmenü zu dem aktuellen Projekt die Option Properties (Eigenschaften) gewählt. Unter Java Build Path (Java Build Pfad) lassen sich unter dem Reiter Libraries externe Jar-Archive unter dem Punkt Add external Jars hinzufügen. Hier wird das Archiv mysql-connector-java-3.0.17-ga-bin.jar hinzugefügt.

9.3 Basic JDBC Concepts

Wenn die JDBC² ausserhalb von Application Servern genutzt wird, stellt die *Drivermanager* Klasse die Verbindung zur Datenbank her. Dem *DriverManager* muss mitgeteilt mit welchem JDBC Treiber er sich verbinden soll. Dies geschieht am einfachsten mit der Methode *Class.forName()*. Für den MySQL Connector lautet der Name der Klasse *com.mysql.jdbc.Driver*.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

// Notice, do not import com.mysql.jdbc.*
// or you will have problems!

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

²Java Database Connectivity (JDBC) ist eine Datenbankschnittstelle der Java-Plattform, die eine einheitliche Schnittstelle zu Datenbanken verschiedener Hersteller bietet und speziell auf relationale Datenbanken ausgerichtet ist. Zu den Aufgaben von JDBC gehört es, Datenbankverbindungen aufzubauen und zu verwalten, SQL-Anfragen an die Datenbank weiterzuleiten und die Ergebnisse in eine für Java nutzbare Form umzuwandeln und dem Programm zur Verfügung zu stellen.

```

        // The newInstance() call is a work around for some
        // broken Java implementations

        Class.forName("com.mysql.jdbc.Driver").newInstance();
    } catch (Exception ex) {
        // handle the error
    }
}
}

```

Nach dem der Treiber mit dem *Drivermanager* registriert wurde, kann man eine Verbindung mit der Methode *DriverManager.getConnection()* zur gewünschten Datenbank herstellen:

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

Connection conn = null;
...
try {
    conn =
        DriverManager.getConnection("jdbc:mysql://localhost/test?" +
                                   "user=monty&password=greatsqldb");

    // Do something with the Connection

    ...
} catch (SQLException ex) {
    // handle any errors
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
}

```

Besteht eine Verbindung, kann man sie benutzen um *Statement* und *PreparedStatement* Objekte für Queries zu erzeugen, aber auch um Metadaten der Datenbank abzufragen. Um ein *Statement* Objekt zu erzeugen, ruft man die Methode *createStatement* der *Connection* Klasse auf. Nachdem man ein *Statement*-Objekt erzeugt hat, kann man eine *Select*-Abfrage ausführen, indem man die Methode *executeQuery(String)* mit der entsprechenden Abfrage aufruft. Um Daten in der Datenbank zu ändern wird die Methode *updateQuery(String)* verwendet.

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

```

```

import java.sql.ResultSet;

// assume that conn is an already created JDBC connection (see
// previous examples)

Statement stmt = null;
ResultSet rs = null;

try {
    stmt = conn.createStatement();
    rs = stmt.executeQuery("SELECT foo FROM bar");

    // or alternatively, if you don't know ahead of time that
    // the query will be a SELECT...

    if (stmt.execute("SELECT foo FROM bar")) {
        rs = stmt.getResultSet();
    }

    // Now do something with the ResultSet ....
}
catch (SQLException ex){
    // handle any errors
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
}
finally {
    // it is a good idea to release
    // resources in a finally{} block
    // in reverse-order of their creation
    // if they are no-longer needed

    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException sqlEx) { } // ignore

        rs = null;
    }

    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException sqlEx) { } // ignore

        stmt = null;
    }
}

```

Literaturverzeichnis

- [1] Herbert Schildt, Joe O'Neil: *Java 5 Ge-packt*. mitp, 2. Auflage, 2005, 3-8266-1555-7