Kapitel 4 Der Umgang mit Zeichenketten

4.1 Strings und deren Anwendung

Ein String ist eine Sammlung von Zeichen, die im Speicher geordnet abgelegt werden. Die Zeichen sind einem Zeichensatz entnommen, der in Java dem Unicode-Standard entspricht. In Java ist eine Symbiose zwischen String als Objekt und String als eingebautem Datentyp vorgenommen worden. Die Sprache ermöglicht zwar die direkte Konstruktion von String-Objekten aus String-Literalen (Zeichenketten in doppelten Anführungszeichen) und die Konkatenation (Aneinanderreihung von Strings mit +) von mehreren Strings, aber alle weiteren Operationen sind als Methoden in den Klassen String , StringBuffer und StringBuilder realisiert. Der Unterschied zwischen StringBuffer und StringBuilder ist lediglich, dass StringBuffer gegen nebenläufige Operationen sicher ist, während StringBuilder das nicht ist.

Die Klasse String repräsentiert Zeichenketten, die sich nicht ändern; sie heißen immutable-Objekte. Mit Objekten vom Typ String lässt sich suchen und vergleichen, aber es können keine Zeichen im String verändert werden. Es gibt einige Methoden, die scheinbar Veränderungen an Strings vornehmen, aber sie erzeugen in Wahrheit neue String-Objekte, die die veränderten Zeichenreihen repräsentieren. So entsteht beim Aneinanderhängen zweier String-Objekte als Ergebnis ein drittes String-Objekt für die zusammengefügte Zeichenreihe. Die Klassen StringBuffer und StringBuilder repräsentieren im Gegensatz dazu dynamische, beliebig änderbare Zeichenreihen.

Die Klassen String und StringBuffer abstrahieren die Funktionsweise und die Speicherung von Zeichenketten. Sie entsprechen der idealen Umsetzung von objektorientierter Programmierung. Die Daten werden gekapselt (die tatsächliche Zeichenkette ist in der Klasse als privates Feld gegen Zugriffe von außen gesichert), und selbst die Länge ist ein Attribut der Klasse.

4.1.1 String-Objekte für konstante Zeichenketten

Damit wir Zeichenketten nutzen können, muss ein Objekt der Klasse String oder StringBuffer erzeugt worden sein. Nutzen wir Literale, so müssen wir String-Objekte nicht von Hand mit new erzeugen, denn für jedes Zeichenketten-Literal im Programm wird (bei Bedarf) automatisch ein entsprechendes String-Objekt erzeugt. Dies geschieht für jede konstante Zeichenkette höchstens einmal, egal wie oft sie im Programmverlauf benutzt wird. (Denken wir hier wieder an das Beispiel println().) Neben den String-Literalen, die uns Strings erzeugen, können wir auch direkt einen Konstruktor der Klasse String – von denen es neun gibt – aufrufen. Als String-Literal wird eine konstante Zeichenkette betrachtet. So ergeben die beiden folgenden Zeilen die Referenz auf ein String-Objekt:

```
String str = "Wer ist Rudolf Wijbrand Kesselaar?";
String str = new String( "Wer ist Rudolf Wijbrand Kesselaar?" );
```

Die zweite Lösung erzeugt unnötigerweise ein zusätzliches String-Objekt, denn das Literal ist ja schon ein String-Objekt.

```
final class java.lang  .String
implements CharSequence, Comparable<String>, Serializable
```

- String()
  Erzeugt ein neues Objekt ohne Zeichen (den leeren String »«).
- String(String string)
  Erzeugt ein neues Objekt mit einer Kopie von string. Es wird selten benötigt, da String-Objekte unveränderbar (immutable) sind.
- String(char value[])
  Erzeugt ein neues Objekt und konvertiert die im char-Feld vorhandenen Zeichen in das String-Objekt.
- String(char value[], int offset, int length)
  Erzeugt wie String(char[]) einen String aus einem Ausschnitt eines Zeichenfelds. Der verwendete Ausschnitt beginnt bei dem Index offset und umfasst length-Zeichen.
- String(byte bytes[])
  Erzeugt ein neues Objekt aus dem Bytefeld. Das Byte-Array enthält keine Unicode-Zeichen, sondern eine Folge von Bytes, die nach der Standardkodierung der jeweiligen Plattform in Zeichen umgewandelt werden.
- String(byte bytes[], int offset, int length)
  Erzeugt wie String(byte[]) einen String aus einem Ausschnitt eines Bytefelds.
- String(byte bytes[], String) throws UnsupportedEncodingException
  Erzeugt einen neuen String von einem Byte-Array mithilfe einer speziellen Zeichenkodierung, die die Umwandlung von Bytes in Unicode-Zeichen festlegt.
- String(byte bytes[], int offset, int length, String)
  throws UnsupportedEncodingException
  Erzeugt einen neuen String mit einem Teil des Byte-Arrays mithilfe einer speziellen Zeichenkodierung.
- String(StringBuffer buffer)
  Erzeugt aus einem veränderlichen StringBuffer-Objekt ein unveränderliches String-Objekt, das dieselbe Zeichenreihe repräsentiert.
- String(StringBuilder builder)
  Erzeugt den String aus einem StringBuilder.
- String(int codePoints[], int offset, int count)
  Neu in Java Java 5 sind die Codepoints, die Zeichen über int kodieren.

Leerer String, Leer-String oder Null-String

Durch

```
String str = "";
```

oder

```
String str = new String();
```

werden String-Objekte erzeugt, die keine Zeichen enthalten. Diesen String nennen wir dann *leeren String*, *Leer-String* oder *Null-String*. Der letzte Begriff ist leider etwas unglücklich und führt oft zur Verwechslung mit Folgendem:

```
String s = null;
System.out.println( s );              // null
```

Hier bekommen wir keinen leeren String und bei der Benutzung im Methodenaufruf, etwa s.length(), eine NullPointerException.

Strings einer gegebenen Länge erzeugen

An einigen Stellen in den Bibliotheken gibt es noch Nachholbedarf – das gilt besonders für die Klasse String. Zum Beispiel gibt es keine Funktion, die eine Zeichenkette einer vorgegebenen Länge aus einem einzelnen Zeichen erzeugt. Selbst in einfachsten Basic-Dialekten gibt es solche Funktionen. In Java müssen wir diese selbst entwickeln.

Zuerst ist zu fragen, ob die Zeichenkette als String oder als StringBuffer/StringBuilder bereitgestellt werden soll? In der Regel wird dieses Objekt ein String sein. Auch ohne Bibliotheksfunktionen lässt sich mit dem Plus-Operator eine Zeichenkette in einer Schleife zusammensetzen. Das ist sicherlich die erste Idee:

```
String s = "";

for ( int i = 0; i < len; ++i )
  s += c;
```

Hier ist len die Länge des Ergebnis-Strings und c das Zeichen.

In einer kritischen Geschwindigkeitsbetrachtung fällt das dauernde Erzeugen von temporären StringBuffer-Objekten auf. Die Lösung ist langsam. Anders können wir dies lösen, indem wir ein char-Feld der passenden Größe erzeugen, dies mit den Zeichen füllen und anschließend einmalig in einen String konvertieren. Anstelle des Zeichenfelds wählen wir besser einen StringBuffer.

4.1.2 String-Länge

String-Objekte verwalten intern die Zeichenreihe, die sie repräsentieren, und bieten eine Vielzahl von Methoden, um die Eigenschaften des Objekts preiszugeben. Eine Methode haben wir schon benutzt: length(). Für String-Objekte ist diese so implementiert, dass die Anzahl der Zeichen im String (die Länge des Strings) zurückgegeben wird.

Beispiel "Hallo".length() hat fünf Zeichen. Leerzeichen und Sonderzeichen werden mitgezählt. Null-Bytes beenden die Zeichenkette nicht.

4.1.3 Gut, dass wir verglichen haben

Um Strings zu vergleichen, existieren eine Menge Möglichkeiten und Optionen. Oft wollen wir einen konstanten String mit einer Benutzereingabe vergleichen. Hier gibt es die aus der Klasse Object geerbte, aber in der Klasse String überschriebene Methode equals(). Die Methode gibt true zurück, falls die Strings Zeichen für Zeichen übereinstimmen. Groß- und Kleinschreibung werden dabei unterschieden. Mit equalsIgnoreCase() werden zwei Zeichenketten verglichen, ohne dass auf die Groß-/Kleinschreibung geachtet wird.

Beispiel equals() liefert für result1 den Wert false und equalsIgnoreCase() für result2 den Wert true.

```
String str = "REISEPASS";
boolean result1 = str.  equals  ( "Reisepass" );              // false
boolean result2 = str.  equalsIgnoreCase  ( "ReISePaSs" );    // true
```

Sortierung mit der Größer/Kleiner-Relation

Wie equals() und equalsIgnoreCase() vergleichen auch die Methoden compareTo(String) und compareToIgnoreCase(String) zwei Strings. equals() aus String überschreibt sie die Methode equals() aus Object, so dass der Parametertyp auch Object ist. Der Argumenttyp beim Aufruf kann natürlich völlig anders sein, doch Gleichheit stellt equals()nur dann fest, wenn das Argument auch vom Typ String ist. (Bei beliebigen Objekten wird nicht automatisch die Methode toString() aufgerufen.) Selbst Vergleiche mit einem inhaltsgleichen StringBuffer-Objekt ergeben immer false – ein StringBuffer ist kein String. Der Rückgabewert von compareTo() ist auch kein boolean, sondern ein int. Das Ergebnis signalisiert, ob das Argument lexikografisch kleiner oder größer als das String-Objekt ist beziehungsweise mit diesem übereinstimmt. Das ist zum Beispiel in einer Sortierfunktion wichtig. Der Sortieralgorithmus muss beim Vergleich zweier Strings wissen, wie sie einzusortieren sind.

Beispiel Ist s der String »Justus«, dann gilt:

```
s.  compareTo  ( "Bob" )      > 0
   //"Justus" ist lexikographisch größer als "Bob"
s.  compareTo  ( "Justus" ) == 0
s.  compareTo  ( "Peter" )   < 0
```

Der von compareTo() vorgenommene Vergleich basiert nur auf der internen numerischen Kodierung der Unicode-Zeichen. Die Vergleichsfunktion berücksichtigt nicht die landestypischen Besonderheiten, etwa die übliche Behandlung der deutschen Umlaute. Dafür müssten wir Collator-Klassen nutzen, die später vorgestellt werden.

compareToIgnoreCase() ist vergleichbar mit equalsIgnoreCase(), bei der die Groß-/Kleinschreibung keine Rolle spielt. Bei Sun wird dies intern mit einem Comparator implementiert, der zwei beliebige Objekte – für Zeichenketten natürlich vom Typ String – in eine Reihenfolge bringt.

Endet der String mit ..., beginnt er mit ...

Interessiert uns, ob der String mit einer bestimmten Zeichenfolge beginnt (wir wollen dies »*Präfix*« nennen), so rufen wir die startsWith()-Methode auf. "http://java-tutor.de".startsWith("http") ergibt true. Eine ähnliche Funktion gibt es für *Suffixe*: endsWith(). Sie überprüft, ob ein String mit einer Zeichenfolge am Ende übereinstimmt.

Beispiel endsWith() für Dateinamen

Die Methode ist praktisch für Dateinamenendungen:

```
String filename  = "Echolallie.gif";
boolean issesGif = filename.  endsWith  ( ".gif" );     // true
```

String-Teile vergleichen

Eine Erweiterung der Ganz-oder-gar-nicht-Vergleichsfunktionen bietet die Methode regionMatches(), mit der Teile einer Zeichenkette mit Teilen einer anderen verglichen werden können. Nimmt das erste Argument von regionMatches() den Wahrheitswert true an, dann spielt die Groß-/Kleinschreibung keine Rolle – damit lässt sich dann auch ein startsWith() und endsWith() mit Vergleichen unabhängig von Groß-/Kleinschreibung durchführen. Der Rückgabewert ist wie bei equalsXXX() ein boolean.

Beispiel Der Aufruf von regionMatches() ergibt true.

```
String s = "Deutsche Kinder sind zu dick";
s.  regionMatches  ( 9, "Bewegungsarmut bei Kindern", 19, 6 );
```

Die Methode beginnt den Vergleich am neunten Zeichen, also bei »K« im String s und dem 19. Buchstaben in dem Vergleichsstring, ebenfalls ein »K«. Dabei beginnt die Zählung der Zeichen wieder bei 0. Ab diesen beiden Positionen werden sechs Zeichen verglichen. Im Beispiel ergibt der Vergleich von »Kinder« und »Kinder« dann true.

Beispiel Sollte der Vergleich unabhängig von der Groß-/Kleinschreibung stattfinden, ist das erste Argument der überladenen Funktion true.

```
s.regionMatches( true, 9, "Bewegungsarmut bei Kindern", 19, 6 );
```

4.1.4 String-Teile extrahieren

Die vielleicht wichtigste Funktion der Klasse String ist charAt(int index). Diese Methode liefert das entsprechende Zeichen an einer Stelle, die »Index« genannt wird. Dies bietet eine Möglichkeit, die Zeichen eines Strings (zusammen mit der Methode length()) zu durchlaufen.

Ist der Index kleiner Null oder größer beziehungsweise gleich der Anzahl der Zeichen im String, so löst die Methode eine StringIndexOutOfBoundsException[1] mit der Fehlerstelle aus.

Beispiel Liefere das erste und letzte Zeichen im String s:

```
String s = "Ich bin nicht dick! Ich habe nur weiche Formen.";
char first = s.  charAt( 0 )  ;                          // 'I'
char last  = s.  charAt( s.length() - 1   );             // '.'
```

Wir müssen bedenken, dass die Zählung wieder bei 0 beginnt. Daher müssen wir von der Länge des Strings eine Stelle abziehen. Da der Vergleich auf den korrekten Bereich bei jedem Zugriff auf charAt() stattfindet, ist zu überlegen, ob der String bei mehrmaligem Zugriff nicht stattdessen einmalig in ein eigenes Zeichen-Array kopiert werden sollte.

Zeichenfolgen als Array aus dem String extrahieren

Eine Erweiterung von charAt() ist getChars(), die Zeichen aus einem angegebenen Bereich in ein übergebenes Feld kopiert:

```
String s = "Body-Mass-Index  = " +
  "Körpergewicht (kg) / Körpergröße (m) / Körpergröße (m)";
char chars[] = new char[13];
s.  getChars  ( 18, 18 + 13, chars, 0 );
```

s.getChars() kopiert ab Position 18 aus dem String s 13 Zeichen in die Elemente des Arrays chars. Das erste Zeichen aus dem Ausschnitt steht dann in chars[0]. Die Methode getChars() muss natürlich wieder testen, ob die gegebenen Argumente im grünen Bereich liegen. Das heißt, ob der Startwert nicht < 0 ist und ob der Endwert nicht über die Größe des Strings hinausgeht. Passt das nicht, löst die Methode eine StringIndexOutOfBoundsException aus. Liegt zudem der Startwert hinter dem Endwert, gibt es ebenfalls eine StringIndexOutOfBoundsException, die anzeigt, wie groß die Differenz der Positionen ist. Am besten ist es, die Endposition aus der Startposition zu berechnen, wie im obigen Beispiel. Passen die Werte, kopiert die Implementierung der Methode getChars() mittels System.arraycopy() die Zeichen aus dem internen Array des String-Objekts in das von uns angegebene Ziel.

Möchten wir den kompletten Inhalt eines Strings als ein Array von Zeichen, so können wir die Methode toCharArray() verwenden. Für häufigen Zugriff auf einen String bewirkt dies eine Geschwindigkeitssteigerung. toCharArray() arbeitet intern auch mit getChars(). Als Ziel-Array wird ein neues Array Objekt angelegt, welches wir dann zurückbekommen.

Beispiel Die untersten vier Bits von i in eine hexadezimale Ziffer umwandeln:

```
char c = "0123456789ABCDEF".toCharArray()[i & 15];
```

Für diesen speziellen Fall wäre charAt() zwar schneller gewesen, jedoch demonstriert das Beispiel, dass wir per [] auch direkt auf die Array-Elemente eines Methodenergebnisses zugreifen können. Das ist völlig korrekt, denn toCharArray() liefert ein Array als Ergebnis.

Teile eines Strings als String

Wollen wir bei den Teilstrings keine Zeichenfelder bekommen, sondern bei dem Typ String bleiben, so greifen wir zur Methode substring(), die in zwei Varianten existiert. Sie liefern beide ein neues String-Objekt zurück, das einem Teil des Originals entspricht.

Beispiel substring(int), die das Ende (oder das Endstück) eines Strings ab einer bestimmten Position als neue Zeichenkette liefert.

```
String s1 = "Die erste McDonalds Filiale öffnete 1971 in München";
String s2 = s1.  substring  ( 44 );                  // München
```

Der String s2 ist dann »München«. Der Index von substring() gibt die Startposition an, ab der Zeichen in die neue Teilzeichenkette kopiert werden. substring() liefert den Teil von diesem Zeichen bis zum Ende des ursprünglichen Strings.

Wollten wir die Teilzeichenkette genauer spezifizieren, so nutzen wir die zweite Variante von substring(). Ihre Parameter erwarten den Anfang und das Ende des gewünschten Ausschnitts:

```
String s1 = "fettleibig : adipös";
String s2 = s1.  substring  ( 4, 8 );        // leib
```

Wie man sieht, bezeichnet die Endposition das erste Zeichen des ursprünglichen Strings, das nicht mehr zur Teilzeichenkette dazugehören soll. Bei genauerer Betrachtung ist substring(int) nichts anderes als eine Spezialisierung von substring(int, int), denn die erste Variante mit dem Startindex lässt sich auch schreiben als:

```
s.substring( beginIndex, s.length() );
```

Selbstverständlich kommen nun diverse Indexüberprüfungen hinzu, die wir von match Region() kennen. Eine StringIndexOutOfBoundsException meldet fehlerhafte Positionsangaben. Stimmen diese, konstruiert ein spezieller String-Konstruktor ein neues String-Objekt als Auszug des Originals.

Strings rechtsbündig setzen

Falls wir immer ein fixes Zeichen verwenden und die String-Länge in einem festen Bereich bleibt, so ist eine andere Möglichkeit noch viel eleganter (aber nicht unbedingt schneller). Sie arbeitet mit der substring()-Methode. Wir schneiden aus einem großen String mit festen Zeichen einfach einen String mit der benötigten Länge heraus. Damit lässt sich auch flott eine Zeile formulieren, die einen Text mit so vielen Leerzeichen füllt, dass dieser rechtsbündig ist:

```
text = "                    ".substring( text.length() );
```

Die Anzahl der Zeichen muss natürlich mit der Zeichenkettenlänge harmonisieren.

4.1.5 Suchen und Ersetzen

Um die erste Position eines Zeichens im String zu finden, verwenden wir die indexOf()-Methode. Als Argument lässt sich unter anderem ein Zeichen oder ein String vorgeben, der gesucht wird.

Beispiel Ein Zeichen mit indexOf() suchen

```
String str = "Ernest Gräfenberg";
int index = str.indexOf( 'e' );                    // 3
```

Im Beispiel ist index gleich 3, da an der Position 3 das erste Mal ein »e« vorkommt. Die Zeichen in einem String werden wie Array-Elemente ab 0 durchnummeriert. Falls das gesuchte Zeichen in dem String nicht vorkommt, gibt die Methode indexOf() als Ergebnis –1 zurück.

Beispiel Um das nächste »e« zu finden, können wir die zweite Version von indexOf() verwenden.

```
index = str.  indexOf  ( 'e', index + 1 );          // 11
```

Mit dem Ausdruck index+1 als Argument der Methode wird in unserem Beispiel ab der Stelle 4 weitergesucht. Das Resultat der Methode ist dann 11. Ist der Index kleiner 0, so wird dies ignoriert und automatisch auf 0 gesetzt.

Beispiel Beschreibt das Zeichen c ein Escape-Zeichen, etwa einen Tabulator oder ein Return, dann soll die Bearbeitung weitergeführt werden.

```
if ( "\b\t\n\f\r\"\\".  indexOf  (c) >= 0 )
{
  ...
}
```

Genauso wie am Anfang gesucht werden kann, ist es auch möglich, am Ende zu beginnen.

Beispiel Hierzu dient die Methode lastIndexOf().

```
String str = "Gary Schubach";
int index  = str.  lastIndexOf  ( 'a' );           // 10
```

Hier ist index gleich 10. Genauso wie bei indexOf() existiert eine überladene Version, die rückwärts ab einer bestimmten Stelle nach dem nächsten Vorkommen von »a« sucht. Wir schreiben:

```
index = str.lastIndexOf( 'a', index - 1 );
```

Nun ist der Index 1.[2]

Hinweis Die Parameter der char-orientierten Methoden indexOf() und lastIndexOf() sind alle vom Typ int und nicht, wie erwartet, vom Typ char und int. Das zu suchende Zeichen wird als erstes int-Argument übergeben. Die Umwandlung des char in ein int nimmt der Java-Compiler automatisch vor, so dass dies nicht weiter auffällt. Bedauerlicherweise kann es dadurch aber zu Verwechslungen bei der Reihenfolge der Argumente kommen: Bei s.indexOf(start, c) wird der erste Parameter start als Zeichen interpretiert und das gewünschte Zeichen c als Startposition der Suche.2

Es gibt noch eine weitere Version von indexOf() und lastIndexOf(), die nach einem *Teilstring* (engl. *substring*) sucht. Die Versionen erlauben ebenfalls einen zweiten Parameter, der den Startindex bestimmt.

Beispiel indexOf() mit der Suche nach einem Teilstring:

```
String str = "In Deutschland gibt es immer noch ein Ruhrgebiet, "+
  "obwohl es diese Krankheit schon lange nicht mehr geben soll.";

String s = "es";
int index = str.  indexOf  ( s, str.indexOf(s) + 1 );       // 57
```

Die nächste Suchposition wird ausgehend von der alten Finderposition errechnet. Das Ergebnis ist 57, da dort zum zweiten Mal das Wort »es« auftaucht.

Das Ideom s.indexOf(t) > –1 ist seit Java 5 nicht mehr nötig, um zu testen, ob ein Teilstring t im String s hier, da es seit dem die Methode contains() gibt.

Zeichen ersetzen

Da String-Objekte unveränderlich sind, kann eine Veränderungsfunktion nur einen neuen String mit den Veränderungen zurückgeben. Die replace()-Methode ist ein Beispiel für diese Vorgehensweise.

Beispiel Ändere den in einer Zeichenkette vorkommenden Buchstaben ›o‹ in ›u‹:

```
String s1 = "Honolulu";
String s2 = s1.  replace  ( 'o', 'u' );          // s2 = "Hunululu"
```

Das String-Objekt mit dem Namen s1 wird selbst nicht verändert, nur ein neues String-Objekt mit dem Inhalt »Hunululu« wird erzeugt und von replace() zurückgegeben.

Die replace()-Methode betrachtet dafür intern eine Kopie des Zeichenfelds und geht dann Schritt für Schritt das Feld ab und führt die Ersetzungen aus. Die replace()-Methode ersetzt dabei alle Zeichen. Eine Variante, die nur das erste Zeichen ersetzt, müssen wir uns selbst schreiben.

Suchen und ersetzen

In Java 5 wurde die Methode replace() eingeführt, die in einem String alle auftretenden Strings sucht und durch einen anderen Sting ersetzt.

```
String s = "'Tag, Karl.' 'Wie geht's, Karl?' 'Gut, Karl.' 'Kahl, Karl?'
'Ja, Karl, ganz kahl.'";
System.out.println( s.replace(".", "!") );
```

Hier ersetzen wir alle Punkte durch Ausrufezeichen, so dass das Ergebnis ist:

```
'Tag, Karl!' 'Wie geht's, Karl?' 'Gut, Karl!' 'Kahl, Karl?' 'Ja, Karl,
ganz kahl!'
```

Suchen und ersetzten mit regulären Ausdrücken

Die Funktionen replaceAll() und replaceFirst() suchen in Zeichenketten mit Hilfe von regulären Ausdrücken und nehmen Ersetzungen vor; replaceFirst() ersetzt, wie es der Name schon sagt, nur das erste Auftreten.

Beispiel Im String s soll »Schnecke« durch »Katze« ersetzt werden:

```
String s = "Schnecken erschrecken, wenn Schnecken an Schnecken schlecken, "
+
        "weil zum Schrecken vieler Schnecken, Schnecken nicht schmecken.";
System.out.println( s.replaceAll("Schnecke", "Katze") );
```

Das Ergebnis auf dem Bildschirm ist »Katzen erschrecken, wenn Katzen an Katzen schlecken, weil zum Schrecken vieler Katzen, Katzen nicht schmecken."

Für ganz allgemeine Ersetzungsaufgaben eignen sich replaceAll() und replaceFirst() nicht; hier ist die replace()-Funktion passend. Da der Suchstring immer ein regulärer Ausdruck ist, können Sonderzeichen auftauchen, die eine besondere Rolle spielen. So müssen zum Beispiel Dollar oder Punkt mit \ ausmaskiert werden. Sollen in unserer Karl-Zeichenfolge wieder alle Punkte durch Ausrufezeichen ersetzen werden, so würde s.replaceAll(".", "!") nicht zum Erfolg führen, sondern nur zu einer Zeichenkette

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Der Punkt steht in regulären Ausdrücken für beliebige Zeichen. Erst s.replaceAll("\\.", "!") liefert das gewünschte Ergebnis.

4.1.6 Veränderte Strings liefern

Obwohl String-Objekte selbst unveränderlich sind, bietet die Klasse String Methoden an, die aus einer Zeichenkette Teile herausnehmen oder ihr Teile hinzufügen. Diese Änderungen werden natürlich nicht am String-Objekt vorgenommen, sondern die Methode liefert eine Referenz auf ein neues String-Objekt mit verändertem Inhalt zurück.

Anhängen an Strings

Eine weitere Methode erlaubt das Anhängen von Teilen an einen String. Wir haben dies schon öfters mit dem Plus-Operator realisiert. Die Methode von String dazu heißt concat(String). Wir werden später sehen, dass die StringBuffer-Klasse dies noch weiter treibt und eine Methode append() mit der gleichen Funktionalität anbietet, die Methode aber für unterschiedliche Typen überladen ist. Das steckt auch hinter dem Plus-Operator. Der Compiler wandelt dies automatisch in eine Kette von append()-Aufrufen um.

Beispiel Hänge hinter eine Zeichenkette das aktuelle Tagesdatum:

```
String s1 = "Das aktuelle Datum ist: ";
String s2 = new Date().toString();
String s3 = s1.  concat  ( s2 );
// Das aktuelle Datum ist: Tue Jun 03 14:46:41 CEST 2003
```

Die concat()-Methode arbeitet relativ zügig und effizienter als der Plus-Operator, der einen temporären String-Puffer anlegt. Doch mit dem Plus-Operator ist es hübscher anzusehen. (Aber wie das so ist: Sieht nett aus, aber ...)

Beispiel Ähnlich wie im oberen Beispiel können wir schreiben:

```
String s3 = "Das aktuelle Datum ist: " + new Date().toString();
```

Es geht sogar noch kürzer, denn der Plus-Operator ruft automatisch toString() bei Objekten auf:

```
String s3 = "Das aktuelle Datum ist: " + new Date();
```

concat() legt ein internes Feld an, kopiert die beiden Zeichenreihen per getChars() hinein und liefert mit einem String-Konstruktor die resultierende Zeichenkette.

Groß-/Kleinschreibung

Die Klasse Character definiert einige statische Methoden, um einzelne Zeichen in Groß-/Kleinbuchstaben umzuwandeln. Die Schleife, die das für jedes Zeichen macht, können wir uns sparen, denn dazu gibt es die Methoden toUpperCase() und toLowerCase() in der Klasse String. Interessant ist an beiden Methoden, dass sie einige sprachabhängige Feinheiten beachten. So zum Beispiel, dass es im Deutschen kein großes »ß« gibt, denn »ß« wird zu »SS«. Gammelige Textverarbeitungen bekommen das manchmal nicht auf die Reihe, und im Inhaltsverzeichnis steht dann so etwas wie »SPAß IN DER NAßZELLE«. Aber bei möglichen Missverständnissen müsste »ß« auch zu »SZ« werden, vergleiche »SPASS IN MASZEN« mit »SPASS IN MASSEN« (ein ähnliches Beispiel steht im Duden). Diese Umwandlung ist aber nur von klein nach groß von Bedeutung. Für beide Konvertierungsrichtungen gibt es jedoch im Türkischen Spezialfälle, bei denen die Zuordnung zwischen Groß- und Kleinbuchstaben von der Festlegung in anderen Sprachen abweicht.

Beispiel Konvertierung von groß in klein und umgekehrt:

```
String s1 = "Spaß in der Naßzelle.";
String s2 = s1.  toLowerCase().toUpperCase()  ;      // SPASS IN DER
NASSZELLE.
System.out.println( s2.length() - s1.length() ); // 2
```

Das Beispiel dient zugleich als Warnung, dass sich im Fall von »ß« die Länge der Zeichenkette vergrößert. Das kann zu Problemen führen, wenn vorher Speicherplatz bereitgestellt wurde. Dann könnte die neue Zeichenkette nicht mehr in den Speicherbereich passen. Arbeiten wir nur mit String-Objekten, haben wir dieses Problem glücklicherweise nicht. Aber berechnen wir etwa für einen Texteditor die Darstellungsbreite einer Zeichenkette in Pixel auf diese Weise, dann sind Fehler vorprogrammiert.

Um länderspezifische Besonderheiten zu berücksichtigen, lassen sich die toXXXCase()-Methoden zusätzlich mit einem Locale-Objekt füttern. Wir gehen in einem eigenen Kapitel auf Sprachumgebungen und die Klasse Locale ein. Die parameterlosen Methoden wählen die Sprachumgebung gemäß den Länder-Einstellungen des Betriebssystems:

```
public String toLowerCase() {
  return toLowerCase( Locale.getDefault() );
}
```

Ähnliches steht bei toUpperCase().

Leerzeichen entfernen

In einer Benutzereingabe oder Konfigurationsdatei stehen nicht selten vor oder hinter dem wichtigen Teil eines Texts Leerzeichen. Vor der Bearbeitung sollten sie entfernt werden. Die String-Klasse bietet dazu trim() an. Diese Methode entfernt Leer- und ähnliche Füllzeichen am Anfang und Ende eines Strings. Andere Trendy-Sprachen wie Visual Basic bieten dazu noch trim()-Funktionen an, die nur die Leerzeichen vorher oder nachher verwerfen. Die Java-Bibliothek bietet das leider nicht.[3]

Beispiel Leerzeichen zur Konvertierung einer Zahl abschneiden:

```
String s = " 1234    ".  trim()  ;            // s = "1234"
int i = Integer.parseInt( s );             // i = 1234
```

Die Konvertierungsfunktion selbst schneidet keine Leerzeichen ab und würde einen Parserfehler melden. Die Helden der Java-Bibliothek haben allerdings bei Float.parseFloat() und Double.parseDouble() anders gedacht. Hier wird die Zeichenkette vorher schlank getrimmt. parseInt() unterstützt verschiedene Zahlensysteme, nicht jedoch Gleitkommazahlen. Auch die Verwandtschaft zwischen den Methoden valueOf() und parseXXX() ist in der Klasse Integer gerade andersherum beschrieben als bei Double und Float.

4.1.7 Unterschiedliche Typen in Zeichenketten konvertieren

Bevor ein Datentyp auf dem Bildschirm ausgegeben werden kann, zum Drucker geschickt oder in einer ASCII-Datei gespeichert wird, muss er in einen String konvertiert werden. Wenn wir etwa die Zahl 7 ohne Umwandlung ausgeben würden, hätten wir keine 7 auf dem Bildschirm, sondern einen Pieps.

Die String-Repräsentation eines primitiven Werts oder eines Objekts kennt die überladene Methode valueOf(). Sie konvertiert einen Datentyp in einen String. Alle valueOf()-Methoden sind statisch.

Beispiel Konvertierungen einiger Datentypen in Strings:

```
String s1 = String.  valueOf  ( 10 );              // "10"
String s2 = String.  valueOf  ( Math.PI );         // "3.141592653589793"
String s3 = String.  valueOf  ( 1 < 2 );           // "true"
String s4 = String.  valueOf  ( new Date() );
// "Tue Jun 03 14:40:38 CEST 2003"
```

Sehen wir uns einige Implementierungen an:

```
public static String valueOf( boolean b ) {
  return b ? "true" : "false";
}
```

Die Methode gibt einfach das passende Literal zurück, unabhängig von der Landessprache. Hier ist die Funktionalität direkt ausprogrammiert. Die Klasse Boolean, die Wrapper-Klasse für den primitiven Datentyp boolean, ist nicht im Spiel, obwohl es ordentlicher wäre. Genauso bei einem Zeichenfeld:

```
public static String valueOf( char data[] ) {
  return new String(data);
}
```

Wir können auch selbst den Konstruktor nutzen, der aus dem Zeichenfeld ein String-Objekt konstruiert.

valueOf() wälzt die Arbeit bei allen anderen primitiven Datentypen an die zuständige Wrapper-Klasse ab, so auch bei einer Ganzzahl:

```
public static String valueOf( int i ) {
  return Integer.toString( i, 10 );
}
```

Dies ist wichtig einzusehen, denn es ist nicht Aufgabe der Klasse String, sich darum zu kümmern, wie eine Zahl in eine Zeichenfolge umgewandelt wird.

Es bleibt abschließend die Frage nach valueOf(Object). Hier kann weder eine Wrapper-Klasse helfen noch kann String selbst etwas machen. Hier ist jedes Objekt gefragt:

```
public static String valueOf( Object obj ) {
  return (obj == null) ? "null" : obj.toString();
}
```

Und ein Objekt kann helfen, da jedes eine toString()-Methode besitzt. Dynamisch gebunden (dazu später mehr) landet der Aufruf in der jeweiligen Klasse des Objekts, sofern diese die Methode toString() überschreibt.

---

[1] Mit 31 Zeichen gehört dieser Methodenname schon zu den längsten. Übertroffen wird aber noch um 5 Zeichen von TransformerFactoryConfigurationError.

[2] Diese Ungenauigkeit ist unter der Bug-ID 4075078 bei Sun gemeldet und immer noch in Bearbeitung. Eine rückwärts kompatible Lösung gibt es aber nicht.

[3] Wieder ein Hinweis, dass Visual Basic einfach die bessere Sprache ist ..

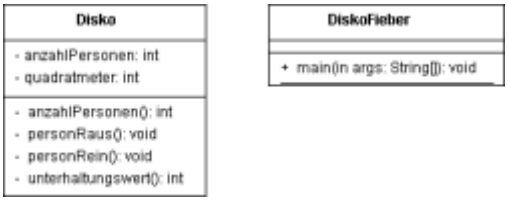
Kapitel 6 Eigene Klassen schreiben

6.1 Eigene Klassen definieren

Die Deklaration einer Klasse wird durch das Schlüsselwort class eingeleitet. Im Rumpf der Klasse lassen sich Variablen deklarieren und Methoden definieren. Zusätzlich sind erlaubt – und zu einem späteren Zeitpunkt erklärt – Konstruktoren, Klassen- sowie Exemplarinitialisierer und innere Klassen beziehungsweise innere Schnittstellen.

Wir wollen das am Beispiel der Klasse Disko darstellen. Diese einfache Klasse definiert Attribute wie die Anzahl von Personen (int anzahlPersonen), die sich in der Disko aufhalten, und die Größe der Disko in Quadratmetern (int quadratmeter). Des Weiteren berechnen wir

den Spaßfaktor einer Disko aus der Anzahl der Personen und der Quadratmeter durch eine obskure willkürliche Formel.



Hier klicken, um das Bild zu Vergrößern

Abbildung 6.1 UML-Diagramme für eine Disko

Zu unserer Disko-Klasse können wir ein konkretes Java-Programm angeben.

Listing 6.1 v1/Disko.java

```
package v1;

public class Disko
{
  int anzahlPersonen;     // Anzahl Personen in der Disko

  int quadratmeter;       // Größe der Disko

  /**
   * Person kommt in die Disko.
   */
  void personRein()
  {
    anzahlPersonen++;
  }

  /**
   * Person verlässt die Disko.
   */
  void personRaus()
  {
    if ( anzahlPersonen > 0 )
    anzahlPersonen--;
  }

  /**
   * Liefert Anzahl Personen in der Disko.
   *
   * @return Anzahl Personen.
   */
  int anzahlPersonen()
  {
    return anzahlPersonen;
  }

  /**
   * Liefert den Unterhaltungswert der Disko.
   *
   * @return  Unterhaltungswert.
   */
  int unterhaltungswert()
  {
```

```
    return (int) (anzahlPersonen * Math.sqrt( quadratmeter ));
  }
}
```

Die angegebene Klasse enthält die Methode zum Erhöhen und Verringern der Personenanzahl in der Disko und zum Berechnen des Spaßfaktors. Die beiden Attribute sind öffentlich und können von außen von jedem genutzt werden.

Um schnell von einer Methode (oder Variablen) zur anderen zu navigieren, lässt sich mit **(Ctrl)+(O)** ein Outline anzeigen (das ist dieselbe Ansicht wie im View Outline). Im Unterschied zur View lässt sich in diesem kleinen gelben Fenster mit den Cursor-Tasten navigieren und ein Return befördert uns zu der angewählten Funktion oder zur Variablen. Wird in der Ansicht erneut **(Ctrl)+(O)** gedrückt, finden sich dort auch die in den Oberklassen definierten Eigenschaften. Sie sind grau, und zusätzlich finden sich hinter den Eigenschaften die Klassennamen.

Hier klicken, um das Bild zu Vergrößern

Eine andere Klasse DiskoFieber soll in der main()-Funktion ein Disko-Objekt erzeugen und die Methoden zum Testen aufrufen.

Listing 6.2 v1/DiskoFieber.java

```
package v1;

public class DiskoFieber
{
  public static void main( String args[] )
  {
    Disko abzappler = new Disko();
    abzappler.quadratmeter = 5;

    System.out.println( "Fun: " + abzappler.unterhaltungswert() );

    abzappler.personRein();
    abzappler.personRein();

    System.out.println( abzappler.anzahlPersonen() );     // 2

    System.out.println( "Fun: " + abzappler.unterhaltungswert() );
  }
}
```

Um dem neu angelegten Disko-Objekt einen Besenkammer-Charakter zu geben, setzen wir die Anzahl Quadratmeter auf 5. Die Nachricht (auch Botschaft) unterhaltungswert() wird an das gewünschte Exemplar der Klasse Disko geschickt. In der Konsolenausgabe erfahren wir

dann, dass der Unterhaltungswert eine Disko mit keiner Person 0 ist. Lassen wir zwei Personen rein, so ändert sich der Unterhaltungswert.

6.1.1 Methodenaufrufe und Nebeneffekte

Alle Variablen und Methoden einer Klasse sind in der Klasse selbst sichtbar. Das heißt, innerhalb einer Klasse werden die Objektvariablen und Funktionen mit ihrem Namen verwendet. Somit greift die Funktion unterhaltungswert() direkt auf die nötigen Attribute zu. Dies wird oft für Nebeneffekte (Seiteneffekte) genutzt. Eine Methode wie personRein() ändert ausdrücklich eine Objektvariable und verändert so den Zustand des Objekts. personRaus() liest nur den Zustand aus und gibt in nach außen frei.

6.1.2 Argumentübergabe mit Referenzen

In Java werden alle Datentypen als Wert übergeben (engl. *copy by value*). Das heißt, die formalen Parameter sind lokale Variablen des Unterprogramms, die mit den aktuellen Parameterwerten – den Argumenten – initialisiert werden. Objekte werden bei der Parameterübergabe nicht kopiert, sondern es wird ihre Referenz übergeben. Die aufgerufene Methode kann dann das Objekt verändern. Dies muss in der Dokumentation der Methode angegeben werden.

Listing 6.3 DiskoFueller.java

```
class PersonenDisko
{
  int anzahlPersonen;
}

class PinkFloyd
{
  static void fuellSie( PersonenDisko d )
  {
    d.anzahlPersonen = 1000;
  }
}

public class DiskoFueller
{
  public static void main( String args[] )
  {
    PersonenDisko partykracher = new PersonenDisko();

    partykracher.anzahlPersonen = 2;
    System.out.println( partykracher.anzahlPersonen );  // 2

    PinkFloyd.fuellSie( partykracher );
    System.out.println( partykracher.anzahlPersonen);  // 10000
  }
}
```

Das Beispiel zeigt eine einfache Disko (PersonenDisko), die durch prominente Gäste gefüllt wird. Die Objektreferenz, die an fuellSie() übergeben wird, lässt eine Attributänderung im Disko-Objekt zu. Referenziert die Variable partykracher ein Disko-Objekt, findet die Änderung in der Methode fuellSie() statt, da die Methode das Objekt über eine Kopie der Objektreferenz in der Variablen d anspricht. In Java wird, anders als zum Beispiel in C++, bei

der Parameterübergabe niemals eine Kopie des übergebenen Objekts angelegt, nur die Objektreferenz wird kopiert und per Wert übergeben.

Wir wollen an dieser Stelle noch einmal den Unterschied zu primitiven Typen hervorheben. Wird ein primitiver Typ an eine Funktion übergeben, so gibt es nur Veränderungen in dieser Methode am aktuellen Parameter, der ja als lokale Variable behandelt werden kann. Eine Veränderung dieser lokalen Variablen tritt somit nicht nach außen und bleibt lokal.

6.1.3 Die this-Referenz

In jedem Konstruktor und jeder Objektmethode einer Klasse existiert eine Referenz mit dem Namen this, die auf das aktuelle Exemplar der Klasse zeigt. Mit dieser this-Referenz lassen sich elegante Lösungen realisieren, wie folgende Beispiele zeigen:

- Die this-Referenz löst das Problem, wenn lokale Variablen Objektvariablen verdecken.
- Wenn Methoden this-Referenzen liefern, hat das gute Gründe, denn Methoden können einfach hintereinander gesetzt werden. Es gibt viele Beispiele für diese Arbeitsweise in den Java-Bibliotheken, etwa bei der Klasse StringBuffer mit der Methode append().

Beispiel Eine Klasse definiert eine Methode personRein(), die den internen Wert einer privaten Variablen anzahlPersonen hochzählt.

Listing 6.4 DiskoZaehler.java

```
public class DiskoZaehler
{
  private int anzahlPersonen;

  public int getAnzahlPersonen()
  {
    return anzahlPersonen;
  }

    public DiskoZaehler personRein()
    {
      anzahlPersonen++;

      return this;
  }

  public static void main( String args[] )
  {
    DiskoZaehler mausefalle = new DiskoZaehler();
    mausefalle.personRein().personRein().personRein();


    System.out.println( mausefalle.getAnzahlPersonen() );
                                                          // 3

    System.out.println( new DiskoZaehler().personRein().getAnzahlPersonen()
);
                                                          // 1
  }
```

```
}
```

Aus diesem Beispiel mit der main()-Methode können wir erkennen, dass new DiskoZaehler() eine Referenz liefert, die wir sofort für den Methodenaufruf nutzen. Da personRein() wiederum eine Objektreferenz vom Typ DiskoZaehler liefert, ist getAnzahlPersonen() möglich. Die Verschachtelung von personRein().personRein() bewirkt, dass immer das interne Attribut erhöht wird und der nächste Methodenaufruf in der Kette eine Referenz auf dasselbe Objekt, aber mit verändertem internem Zustand (= Zählerstand), über this bekommt.

6.1.4 Überdeckte Objektvariablen nutzen

Hat eine lokale Variable den gleichen Namen wie eine Objektvariable, so verdeckt sie diese. Das heißt aber nicht, dass auf die äußere Variable nicht mehr zugegriffen werden kann. Mit der this-Referenz kann auf das aktuelle Objekt zugegriffen werden und entsprechend mit dem Punkt-Operator auf einzelne Variablen des Objekts. Häufiger Einsatzort sind Funktions- oder Konstruktorparameter, die genauso genannt werden wie die Exemplarvariablen, um damit eine starke Zugehörigkeit auszudrücken.

Listing 6.5 DiskoThis.java

```
class DiskoThis
{
  int quadratmeter;

  void setQuadratmeter( int quadratmeter )
  {
    quadratmeter = 12;
                              // Zuweisung an lokale Variable quadratmeter
      this.quadratmeter    = 12;
                              // Zuweisung an Objektvariable
      this.quadratmeter    = quadratmeter;
                              // Initialisierung der Objektvariablen
  }
}
```

Der Methode setQudratmeter() wird ein Wert übergeben, der anschließend die Objektvariablen initialisieren soll. Genau in dem Moment, wo eine lokale Variable deklariert wird und sie eine Objekt- oder Klassenvariable überlagert, wird beim Zugriff auf die lokale Variable verwiesen. Das zeigt die erste Zeile: quadratmeter = 12 überschreibt den aktuellen Parameterwert der Funktion, der damit verloren ist. Erst mit this.quadratmeter greifen wir auf die Objektvariable direkt zu.

Hier klicken, um das Bild zu Vergrößern

Eclipse erkennt unsinnige Konstruktionen wie anzahlPersonen = anzahlPersonen.

Ein this-Problem

Nutzen wir Konstruktionen wie this.anzahlPersonen = anzahlPersonen, so kann das zu einem schwer zu findenden Fehler führen. Das nachfolgende Beispiel zeigt das Problem unter der Annahme, es gebe eine Objektvariable anzahlPersonen:

```
void setAnzahlPersonen( int anzahlPerson )
{
  this.anzahlPersonen = anzahlPersonen;
}
```

Die Methode kompiliert, doch sie enthält einen logischen Fehler. Erkannt? Die Parameter-Variable heißt anzahlPerson, müsste aber eigentlich anzahlPersonen heißen. Der Fehler fällt so erst nicht auf, da die Objektvariable anzahlPersonen einfach mit sich selbst überschrieben wird – glücklicherweise merkt das Eclipse. Doch wie perfekt programmiert man, wenn man 10.000 Mal programmiert hat?

Kapitel 7 Exceptions

»Wir sind in Sicherheit! Er kann uns nicht erreichen!«
»Sicher?«
»Ganz sicher! Bären haben Angst vor Treibsand!«
Hägar, Dik Browne

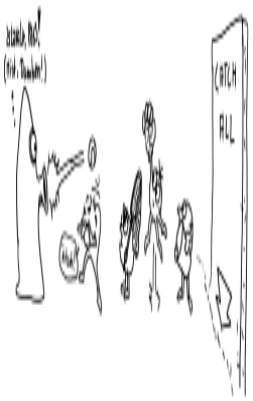
Dass Fehler beim Programmieren auftauchen, ist unvermeidlich. Schwierig sind nur die unkalkulierbaren Situationen, und daher ist der Umgang mit Fehlern ganz besonders heikel. Java bietet die elegante Methode der Exceptions, um mit Fehlern flexibel umzugehen.

7.1 Problembereiche einzäunen

Werden in C Routinen aufgerufen, dann haben diese keine andere Möglichkeit, als über den Rückgabewert einen Fehlschlag anzuzeigen. Der Fehlercode ist häufig -1, aber auch NULL oder 0. Allerdings kann die Null auch Korrektheit anzeigen. Irgendwie ist das willkürlich. Die Abfrage dieser Werte ist unschön und wird von uns gerne unterlassen, zumal wir oft davon ausgehen, dass ein Fehler in dieser Situation gar nicht auftreten kann – diese Annahme kann eine Dummheit sein. Zudem wird der Programmfluss durch Abfragen der Funktionsergebnisse unangenehm unterbrochen, zumal der Rückgabewert, wenn er nicht gerade einen Fehler anzeigt, weiterverwendet wird. Der Rückgabewert ist also im weitesten Sinne überladen, da er zwei Zustände anzeigt. Häufig entstehen mit den Fehlerabfragen kaskadierte if-Abfragen, die den Quellcode schwer lesbar machen.

7.1.1 Exceptions in Java mit try und catch

Bei der Verwendung von Exceptions wird der Programmfluss nicht durch Abfrage des Rückgabestatus unterbrochen, sondern ein besonders ausgezeichnetes Programmstück wird bezüglich auftretender Fehler überwacht und gegebenenfalls spezieller Code zur Behandlung solcher Fehler aufgerufen. Der überwachte Programmbereich (Block) wird durch das Schlüsselwort try eingeleitet und durch catch beendet. Hinter dem catch folgt der Programmblock, der beim Auftreten eines Fehlers ausgeführt wird, um den Fehler abzufangen oder zu behandeln. Daher auch der Ausdruck catch.



Hier klicken, um das Bild zu Vergrößern

7.1.2 Eine Datei auslesen mit RandomAccessFile

Wir wollen eine Datei mithilfe der Klasse RandomAccessFile zeilenweise auslesen. Die Verbindung zwischen der Datei und dem zugehörigen Objekt gelingt mit dem Konstruktor, dem wir einen Dateinamen mitgeben.

Eine nicht behandelte Ausnahme wird von Eclipse als Fehler angezeigt.

Hier klicken, um das Bild zu Vergrößern

Aus der API-Dokumentation geht hervor, dass der Konstruktor von RandomAccessFile eine FileNotFound Exception-Ausnahme auslösen kann. Mithilfe der Funktion readLine() lesen wir so lange Zeilen ein, bis die Datei ausgeschöpft ist. Die Methode readLine() kann eine IOException auslösen. Wir müssen diese behandeln und setzen daher die Problemzonen in einen try- und catch-Block**.**

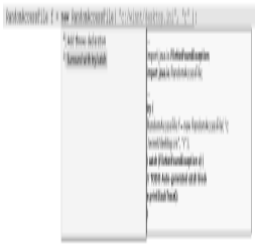
Listing 7.1 ReadFileWithRAF.java

```
import java.io.*;

public class ReadFileWithRAF
{
  public static void main( String args[] )
  {
    try
    {
      RandomAccessFile f;
      f = new RandomAccessFile( "c:/windows/desktop.ini", "r" );

      for ( String line; (line=f.readLine()) != null; )
        System.out.println( line );
    }
    catch ( FileNotFoundException e )  // Datei gibt's nich'
    {
      System.err.println( "Datei gibt's nicht!" );
    }
```

```
    catch ( IOException e )            // Schreib- Leseprobleme
    {
      System.err.println( "Schreib- Leseprobleme!" );
    }
  }
}
```

Tritt beim Laden einer Datei ein Fehler auf, wird dieser im try-Block abgefangen und im catch-Teil bearbeitet. Einem try-Block können mehrere catch-Klauseln zugeordnet sein, um verschiedene Fehlertypen aufzufangen.

Ein try/catch-Block kann Eclipse auch selbst anlegen. Dazu wird wieder **(Strg)**+**(1)** bemüht und aus Auswahl steht einmal die Möglichkeit den Fehler an den Aufrufer weiterzuleiten – das schauen wir uns etwas später an – und ein try/catch-Block anzulegen.



Hier klicken, um das Bild zu Vergrößern

Leere catch-Blöcke

Java schreibt vor, dass Ausnahmen in einem catch behandelt (oder nach oben geleitet) werden, aber nicht, was in catch-Blöcken zu geschehen hat. Ein leerer catch-Block ist in der Regel wenig sinnvoll, denn dann werden die Fehler klammheimlich unterdrückt. (Das wäre genauso wie ignorierte Statusrückgabewerte von C-Funktionen.) Das Mindeste ist eine minimale Fehlerbehandlung wie System.err.println(e) oder das informativere e.printStackTrace() für eine Exception e. Optimal ist das Loggen von diesen Fehlern.

7.1.3 Ablauf einer Ausnahmesituation

Ein Ausnahme-Objekt wird vom Laufzeitsystem erzeugt, wenn ein Fehler über eine Exception angezeigt werden soll. Dann wird die Abarbeitung der Programmzeilen sofort unterbrochen, und das Laufzeitsystem steuert selbstständig die erste catch-Klausel an (oder springt weiter zum Aufrufer, wie wir später sehen werden). Wenn die erste catch-Anweisung nicht zur Art des aufgetretenen Fehlers passt, werden der Reihe nach alle übrigen catch-Klauseln untersucht und die erste übereinstimmende Klausel wird angesprungen (oder ausgewählt). Erst wird etwas versucht, und wenn im Fehlerfall ein Exception-Objekt in den Programmtext geworfen wird, kann es an einer Stelle aufgefangen werden. Da immer die erste passende catch-Klausel ausgewählt wird, darf im Beispiel die letzte catch-Klausel keinesfalls zuerst stehen, da diese auf jeden Fehler passt. Alle anderen Anweisungen in den catch-Blöcken würden dann nicht ausgeführt. Mittlerweile erkennt der Compiler dieses Problem und gibt einen Fehler aus.

7.1.4 Wiederholung kritischer Bereiche

Es gibt in Java bei Ausnahmen bisher keine von der Sprache unterstützte Möglichkeit, an den Punkt zurückzukehren, der den Fehler ausgelöst hat. Das ist aber oft gewünscht, etwa in dem Fall, wenn eine fehlerhafte Eingabe zu wiederholen ist.

Wir werden mit JOptionPane.showInputDialog() nach einem String fragen und versuchen, diesen in eine Zahl zu konvertieren. Dabei kann natürlich etwas schief gehen. Wenn ein Benutzer eine Zeichenkette eingibt, die keine Zahl repräsentiert, dann wird eine NumberFormatException ausgelöst. Wir wollen in diesem Fall die Eingabe wiederholen.

Listing 7.2 ContinueInput.java

```
import javax.swing.*;

public class ContinueInput
{
  public static void main( String args[] )
  {
    int number = 0;

    while ( true )
    {
      try
      {
        String s = JOptionPane.showInputDialog(
                     "Bitte Zahl eingeben" );
        number = Integer.parseInt( s );

          break;
      }
      catch ( NumberFormatException e )
      {
        System.out.println( "Das war keine Zahl!" );
      }
    }

    System.out.println( "Danke für die Zahl " + number );
    System.exit( 0 );           // Beendet die Anwendung
  }
}
```

Die gewählte Lösung ist einfach. Wir programmieren den gesamten Teil in einer Endlosschleife. Geht die problematische Stelle ohne Fehler durch, so beenden wir die Schleife mit break. Kommt es zu einer Ausnahme, dann wird break nicht ausgeführt, und nach der Exception gelangen wir wieder in die Endlosschleife.

7.1.5 throws im Methodenkopf angeben

Neben dem Einzäunen von problematischen Blöcken durch einen try- und catch-Block gibt es noch eine andere 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.

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 auswerfen 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();
}
```

Dadurch »bubbelt« der Fehler entlang der Kette von Methodenaufrufen nach oben und kann irgendwann von einem Block abgefangen werden, der sich darum kümmert.

Wenn main() alles weiterleitet

Ist die Fehlerbehandlung in einem Hauptprogramm ganz egal, so können wir alle Fehler auch an die Laufzeitumgebung weiterleiten, die dann das Programm im Fehlerfall abbricht.

Listing 7.3 MirIstAllesEgal.java

```
import java.io.*;

class MirIstAllesEgal
{
  public static void main( String args[] )   throws Exception
  {
    RandomAccessFile f = new RandomAccessFile( "Datei.txt", "r" );
    System.out.println( f.readLine() );
  }
}
```

Das funktioniert, da alle Fehler von der Klasse Exception[1] –Exception-Oberklasse Throwable abgeleitet.

abgeleitet sind. Wir werden das in den folgenden Kapiteln weiterverfolgen. Wird der Fehler nirgendwo sonst aufgefangen, dann wird eine Laufzeitfehlermeldung ausgegeben, denn das Exception-Objekt ist beim Interpreter, also bei der virtuellen Maschine, auf der äußersten Aufrufebene gelandet. Natürlich ist das kein guter Stil – obwohl es aus Gründen kürzerer Programme auch in diesem Buch so gemacht wird. Denn Fehler sollten in jedem Fall behandelt werden.

7.1.6 Abschließende Arbeiten mit finally

Nach einem catch-Block kann optional noch ein finally-Block folgen. Der Teil im finally wird immer ausgeführt, auch wenn in try und catch ein return, break oder continue steht. Das heißt, der Block wird auf jeden Fall ausgeführt. Eine typische Anwendung ist die Freigabe von Ressourcen oder das Schließen von Dateien.

Ein try ohne catch

Es kommt zu einer merkwürdigen Konstellation, wenn mit throws eine Exception nach oben geleitet wird. Dann ist ein catch für diese Fehlerart nicht notwendig. Dennoch lässt sich dann ein Block mit einer Ereignisbehandlung umrahmen, um ein finally auszuführen:

```
void read() throws MyException
{
  try
  {
    // hier etwas Arbeiten, was eine MyException auslösen könnte

    return;
  }
    finally
  {
    System.out.prinln( "Ja, das kommt danach" );
  }
}
```

Ein return im finally lässt Ausnahmen verschwinden

Ein Phänomen in der Ausnahmebehandlung von Java ist eine return-Anweisung innerhalb eines finally-Blocks. Wird dort ein return eingesetzt, wird eine ausgelöste Ausnahme nicht zum Aufrufer weitergeleitet.

Beispiel Die Methode buh() löst eine ArithmeticException aus, eine spezielle Art von RuntimeException.

Der Aufrufer von buh() ist die main()-Funktion. Es ist zu erwarten, dass main() abbricht, denn die Exception wird doch nicht abgefangen. Dem ist aber bei einem return im finally nicht so. Erst wenn wir diese Zeile entfernen, wird die erwartete Ausnahme die Laufzeitumgebung beenden.

Listing 7.4 NoExceptionBecauseOfFinallyReturn.java

```
public class NoExceptionBecauseOfFinallyReturn
{
  static void buh()
  {
    try
    {
      throw new ArithmeticException( "Keine Lust zu rechnen" );
    }
```

```
    finally
    {
      // Das return bewirkt normalen Rücksprung aus buh() ohne Exception
        return;          // Spannende Zeile.
    }
  }

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

Hinweis Haarspalterisch genau lässt sich auch ein Beispiel finden, in dem finally nicht ausgeführt wird.

```
try
{
  System.exit( 1 );
}
finally
{
  System.out.println( "Das wirst du nicht erleben!" );
}
```

7.1.7 Nicht erreichbare catch-Klauseln

Eine catch-Klausel heißt *erreichbar*, wenn es in dem try- und catch-Block eine Anweisung gibt, die die Fehlerart, die in der catch-Klausel aufgefangen wird, tatsächlich auslösen kann. Zusätzlich darf vor dieser catch-Klausel natürlich kein anderes catch stehen, das diesen Fehlerfall mit abfängt. Wenn wir zum Beispiel catch(Exception e) als erstes Auffangbecken bereitstellen, dann werden natürlich alle Ausnahmen dort behandelt. Die Konsequenz daraus ist, catch-Klauseln immer von den speziellen zu den allgemeinen Fehlerarten zu sortieren.

Wenn wir ein Objekt RandomAccessFile aufbauen und anschließend readLine() verwenden, so muss eine FileNotFoundException vom Konstruktor und eine IOException von readLine() abgefangen werden. Da eine FileNotFoundException eine Spezialisierung, also eine Unterklasse von IOException ist, würde eine catch(IOException e) schon reichen, denn dies fängt alles ab, auch die FileNotFoundException. Steht im Quellcode folglich der catch für die FileNotFoundException dahinter, wird der Teil nie ausgeführt werden können und der Compiler merkt das zu Recht an.

Leere Blöcke

Da der leere Block keine Exception auslösen kann, ist der catch-Block des nächstens Programms nicht erreichbar und daher ist das Programm falsch.

Listing 7.5 NoException.java

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

```
    try {
    }
    catch ( Exception e )
    {
     System.out.println( "Hab' dich" );
    }
  }
}
```

Ein Compiler sollte diese Situation erkennen, obwohl leider einige Compiler mit dieser Situation noch Schwierigkeiten haben. (Ein leerer try-Block ist natürlich auch ein recht seltener Spezialfall, sonst sind ziemlich viele Arten von Exceptions auch in scheinbar harmlosem Code denkbar: ArrayIndexOutOfBoundsException oder andere RuntimeExceptions.)

Übertriebene throws

Ein anderes Problem sind übertriebene throws-Klauseln. Es ist nicht falsch, wenn eine Methode zu viele oder zu allgemeine Fehlerarten in ihrer throws-Klausel angibt. Beim Aufruf solcher Methoden in try-Blöcken sind catch-Klauseln für die zu viel deklarierten Exceptions formal korrekt, können aber natürlich nicht wirklich erreicht werden.

---

[1] Genauer gesagt sind alle Ausnahmen in Java von der

Kapitel 8 Die Funktionsbibliothek

8.1 Die Java-Klassenphilosophie

Eine Programmiersprache besteht nicht nur aus einer Grammatik, sondern, wie im Fall von Java, aus einer Programmierbibliothek. Eine plattformunabhängige Sprache – so wie sich viele C oder C++ vorstellen – ist nicht wirklich plattformunabhängig, wenn auf jedem Rechner andere Funktionen und Programmiermodelle eingesetzt werden. Genau dies ist der Schwachpunkt von C(++). Die Algorithmen, die kaum abhängig vom Betriebssystem sind, lassen sich überall gleich anwenden, doch spätestens bei grafischen Oberflächen ist Schluss. Dieses Problem tritt in Java seltener auf, da sich die Entwickler viel Mühe gegeben haben, alle wichtigen Funktionen in wohlgeformte Pakete zu setzen. Diese decken insbesondere die zentralen Bereiche Datenstrukturen, Ein- und Ausgabe, Grafik- und Netzwerkprogrammierung ab.

8.1.1 Übersicht über die Pakete der Standardbibliothek

Es folgt eine Übersicht über die Pakete, die Java in der Version 1.4 definiert. Sie beschreiben zusammen mehr als 2.700 Klassen und Schnittstellen. Das ergibt zusammen etwa 4.800 Attribute, mehr als 3.700 statische Variablen, mehr als 21.300 Objektmethoden und fast 2.500 statische Methoden. Die Exemplare der Klassen werden mit mehr als 3.500 Konstruktoren erzeugt.

Uns wird auffallen, dass es bei der Benennung der Pakete eine Methodik gibt. Die SPI-Pakete (SPI für *Service Provider Implementation*) implementieren die abstrakten Klassen und Schnittstellen aus dem Oberpaket.

Tabelle 8.1 Java 2-Plattform-Pakete zum SDK 1.4

java.applet	Stellt Klassen für Java-Applets bereit, damit diese auf Web-Seiten ihr Leben führen können.
java.awt	Das Paket AWT (*Abstract Windowing Toolkit*) bietet Klassen zur Grafikausgabe und Nutzung von grafischen Bedienoberflächen.
java.awt.color	Unterstützung von Farbräumen und Farbmodellen
java.awt.datatransfer	Informationsaustausch zwischen (Java-)Programmen über die Zwischenablage des Betriebssystems
java.awt.dnd	Drag&Drop, um unter grafischen Oberflächen Informationen zu übertragen oder zu manipulieren
java.awt.event	Schnittstellen für die verschiedenen Ereignisse unter grafischen Oberflächen werden definiert.
java.awt.font	Klassen, damit Zeichensätze genutzt und modifiziert werden können
java.awt.geom	Paket für die Java 2D API, um ähnlich wie im Grafikmodell von Postscript beziehungsweise PDF affine Transformationen auf beliebige 2D-Objekte anwenden zu können
java.awt.im	Klassen für alternative Eingabegeräte
java.awt.im.spi	Schnittstellen für Eingabemethoden
java.awt.image	Erstellen und Manipulieren von Rastergrafiken
java.awt.image.renderable	Klassen und Schnittstellen zum allgemeinen Erstellen von Grafiken
java.awt.print	Bietet Zugriff auf Drucker und kann Druckaufträge erzeugen.
java.beans	Mit JavaBeans definiert Sun wiederverwendbare Komponenten auf der Client-Seite, die beim Programmieren visuell konfiguriert werden können.
java.beans.beancontext	Mehrere zusammenarbeitende Beans sind in einem Kontext miteinander verbunden. Mit diesem Paket lässt sich dieser nutzen.
java.io	Definiert Möglichkeiten zur Ein- und Ausgabe. Dateien werden als Objekte repräsentiert. Datenströme erlauben sequenziellen Zugriff auf die Dateiinhalte.
java.lang	Ein Paket, welches automatisch geladen wird und unverzichtbare Klassen wie String-, Thread- oder Wrapper-Klassen enthält.
java.lang.annotation	Unterstützung für die in Java 5 hinzugekommenen Annotationen
java.lang.instrument	Klassen können instrumentalisiert werden

java.lang.management	Überwachung der virtuellen Maschine
java.lang.ref	Behandelt Referenzen
java.lang.reflect	Mit Reflection ist es möglich, dass Klassen und Objekte über sich erzählen.
java.math	Beliebig lange Ganzzahlen oder Fließkommazahlen
java.net	Kommunikation über Netzwerke. Klassen zum Aufbau von Client- und Serversystemen, die sich über TCP beziehungsweise IP mit dem Internet verbinden lassen.
java.nio	Neue IO-Implementierung (daher NIO[1]) für performante Ein- und Ausgabe
java.nio.channels	Datenkanäle für nicht blockierende Ein- und Ausgabeoperationen
java.nio.channels.spi	Anbieter für die Kanäle
java.nio.charset	Definierte Zeichensätze, Decoder und Encoder für die Übersetzung zwischen Bytes und Unicode-Zeichen
java.nio.charset.spi	Anbieter für die Zeichensätze aus java.nio.charset
java.rmi	Aufruf von Methoden auf entfernten Rechnern
java.rmi.activation	Unterstützung für die RMI-Aktivierung, wenn Objekte auf ihren Aufruf warten
java.rmi.dgc	Der verteilte Gargabe-Collector DGC (Distributed Garbage-Collection)
java.rmi.registry	Zugriff auf den Namensdienst unter RMI, die Registry
java.rmi.server	Definition der Server-Seite von RMI
java.security	Klassen und Schnittstellen für Sicherheit
java.security.acl	Unwichtig, da sie durch Klassen in java.security ersetzt wurden
java.security.cert	Analysieren und Verwalten von Zertifikaten, Pfaden und Rückruf (Verfall) von Zertifikaten
java.security.interfaces	Schnittstellen zu RSA-Verschlüsselung (Rivest, Shamir Adleman AsymmetricCipher Algorithm) und DSA (Digital Signature Algorithm) Schlüsseln
java.security.spec	Definition der Schlüssel und Algorithmen für Verschlüsselung
java.sql	Zugriff auf Datenbanken über SQL
java.text	Unterstützung für internationalisierte Programme. Behandlung von Text, Formatierung von Datumswerten und Zahlen
java.util	Datenstrukturen, Raum und Zeit sowie Teile der Internationalisierung, Zufallszahlen
java.util.concurrent	Hilfsklassen für nebenläufiges Programmieren, etwa ThreadPools
java.util.concurrent.atomic	Atomare Operationen auf Variablen
java.util.concurrent.locks	Lock-Objekte
java.util.jar	Möglichkeiten, um auf das eigene Archiv-Format JAR (Java

	Archive) zuzugreifen
java.util.logging	Protokollieren von Daten und Programmabläufen
java.util.prefs	Benutzer- und Systemeigenschaften werden über Konfigurationsdateien verwaltet.
java.util.regex	Unterstützung von regulären Ausdrücken
java.util.zip	Zugriff auf komprimierte Daten mit GZIP und Archive (ZIP)
javax.accessibility	Schnittstellen zwischen Eingabegeräten und Benutzerkomponenten
javax.crypto	Klassen und Schnittstellen für kryptografische Operationen
javax.crypto.interfaces	Schnittstellen für Diffie-Hellman-Schlüssel
javax.crypto.spec	Klassen und Schnittstellen für Schlüssel und Parameter der Verschlüsselungsfunktionen
javax.imageio	Schnittstellen zum Lesen und Schreiben von Bilddateien in verschiedenen Formaten
javax.imageio.event	Ereignisse, die während des Ladens und Speicherns bei Grafiken auftauchen
javax.imageio.metadata	Unterstützung für beschreibende Metadaten in Bilddateien
javax.imageio.plugins.bmp	Klassen, die das Lesen und Schreiben von BMP-Bilddateien unterstützen
javax.imageio.plugins.jpeg	Klassen, die das Lesen und Schreiben von JPEG-Bilddateien unterstützen
javax.imageio.spi	Einlesen und Schreiben von Bildern in Java gemäß der in javax.imageio definierten Schnittstellen
javax.imageio.stream	Unterstützt das Einlesen und Schreiben von Bildern durch die Behandlung der unteren Ebenen
javax.management	Management API (JMX) mit einigen Unterpaketen
javax.naming	Zugriff auf Namensdienste
javax.naming.directory	Zugriff auf Verzeichnisdienste, erweitert das javax.naming-Paket
javax.naming.event	Ereignisse, wenn sich etwas beim Verzeichnisdienst ändert
javax.naming.ldap	Unterstützung von LDAPv3-Operationen
javax.naming.spi	Definition für Anbieter von Namensdiensten, damit diese JNDI nutzen können
javax.net	Klassen mit einer Socket-Fabrik
javax.net.ssl	SSL-Verschlüsselung
javax.print	Java Print Service API
javax.print.attribute	Attribute (wie Anzahl der Seiten, Ausrichtung) beim Java Print Service
javax.print.attribute.standard	Standard für einige Drucker-Attribute
javax.print.event	Ereignisse beim Drucken

javax.rmi	Nutzen von RMI über das CORBA-Protokoll RMI-IIOP
javax.rmi.CORBA	Unterstützt Portabilität von RMI-IIOP.
javax.security.auth	Framework für Authentifizierung und Autorisierung
javax.security.auth.callback	Informationen wie Benutzernamen oder Passwort vom Server beziehen
javax.security.auth.kerberos	Unterstützung von Kerberos zur Authentifizierung in Netzwerken
javax.security.auth.login	Framework für die Authentifizierungsdienste
javax.security.auth.spi	Schnittstelle für Authentifizierungsmodule
javax.security.auth.x500	Für X.509 Zertifikate, X.500 Principal und X500PrivateCredential
javax.security.cert	Public-Key-Zertifikate
javax.security.sasl	Unterstützung für SASL (Simple Authentication and Security Layer)
javax.sound.midi	Ein- und Ausgabe, Synthetisierung von MIDI-Daten
javax.sound.midi.spi	Schnittstellen für Anbieter von neuen MIDI-Diensten
javax.sound.sampled	Schnittstellen zur Ausgabe und Verarbeitung von Audio-Daten
javax.sound.sampled.spi	Schnittstellen für Anbieter von Audio-Konvertern, Lese- und Schreibroutinen
javax.sql	Datenquellen auf Serverseite
javax.sql.rowset	Implementierung von RowSet. Mit Unterpaketen
javax.swing	Definiert die einfachen Swing-Komponenten.
javax.swing.border	Grafische Rahmen für die Swing-Komponenten
javax.swing.colorchooser	Anzeige vom JColorChooser, einer Komponente für die Farbauswahl
javax.swing.event	Ereignisse der Swing-Komponenten
javax.swing.filechooser	Dateiauswahldialog unter Swing: JFileChooser
javax.swing.plaf	Unterstützt auswechselbares Äußeres bei Swing durch abstrakte Klassen.
javax.swing.plaf.basic	Besonders einfach gehaltenes Erscheinungsbild für Swing-Komponenten
javax.swing.plaf.metal	Plattformunabhängiges Standarderscheinungsbild von Swing-Komponenten
javax.swing.plaf.multi	Benutzerschnittstellen, die mehrere Erscheinungsbilder kombinieren
javax.swing.plaf.synth	Swing-Look-and-Feel aus XML-Dateien
javax.swing.table	Rund um die grafische Tabellenkomponente javax.swing.JTable
javax.swing.text	Unterstützung für Textkomponenten
javax.swing.text.html	HTMLEditorKit zur Anzeige und Verwaltung eines HTML-Texteditors

javax.swing.text.html.parser	Einlesen, visualisieren und strukturieren von HTML-Dateien
javax.swing.text.rtf	Editorkomponente für Texte im Rich-Textformat (RTF)
javax.swing.tree	Zubehör für die grafische Baumansicht javax.swing.JTree
javax.swing.undo	Undo- oder Redo-Operationen, etwa für einen Texteditor
javax.transaction	Ausnahmen bei Transaktionen
javax.transaction.xa	Beziehung zwischen Transactions-Manager und Resource-Manager für Java Transaction API (JTA), besonders für verteilte Transaktionen (Distributed Transaction Processing: The XA Specification)
javax.xml	Konstanten aus der XML-Spezifikation
javax.xml.datatype	Schema-Datentypen Dauer und Gregorianischer Kalender
javax.xml.namespace	QName für den Namensraum
javax.xml.parsers	Einlesen von XML-Dokumenten
javax.xml.transform	Allgemeine Schnittstellen zur Transformation von XML-Dokumenten
javax.xml.transform.dom	Implementiert Transformationen auf der Basis von XML-Parsern nach dem DOM-Standard.
javax.xml.transform.sax	Implementiert Transformationen auf der Basis von XML-Parsern nach dem SAX2-Standard.
javax.xml.transform.stream	Transformationen auf der Basis von linearisierten XML-Dokumenten.
javax.xml.validation	Validation nach einem Schema
javax.xml.xpath	XPath API
org.ietf.jgss	Framework für Sicherheitsdienste wie Authentifizierung, Integrität, Vertraulichkeit
org.w3c.dom	Klassen für die Baumstruktur eines XML-Dokuments nach DOM-Standard. Mit Unterpaketen bootstrap und ls.
org.xml.sax	Ereignisse, die beim Einlesen eines XML-Dokuments nach dem SAX-Standard auftreten.
org.xml.sax.ext	Zusätzliche Behandlungsroutinen für SAX2-Ereignisse
org.xml.sax.helpers	Adapterklassen und Standardimplementierungen

Daneben definieren die Pakete org.omg eine Reihe von CORBA-Diensten, die für unsere Betrachtung jedoch zu speziell sind.

Standard Extension API

Einige der Java-Pakete beginnen mit javax. Dies sind Erweiterungspakete (eXtentions), die die Kern-Klassen ergänzen. Im Laufe der Zeit sind jedoch viele der früher zusätzlich einzubindenden Pakete in die Standard-Distribution gewandert, so dass heute ein recht großer Anteil mit javax beginnt, aber keine Erweiterungen mehr darstellen, die zusätzlich installiert werden müssen. Sun wollte die Pakete nicht umbenennen, um so eine Migration nicht zu

erschweren. Fällt heute im Quellcode ein Paketname mit javax auf, so ist es daher nicht mehr so einfach zu entscheiden, ob eine externe Quelle mit eingebunden werden muss, beziehungsweise ab welcher Java-Version das Paket Teil der Distribution ist.

Echte externe Pakete von Sun sind unter anderem:

- Enterprise/Server API mit den Java Enterprise Bean, Servlets und JavaServer Faces
- JDO zum Ablegen von Objekten in Datenbanken
- JavaSpaces für gemeinsamen Speicher unterschiedlicher Laufzeitumgebungen
- JXTA erlaubt das Aufbauen von P2P-Netzwerken
- 3D-API
- Java Telephony API

---

[1] NIO steht ja für New IO. Dumm, wenn es auch wieder überholt ist. Erwartet uns dann NNIO?

Kapitel 9 Threads und nebenläufige Programmierung

9.1 Prozesse und Threads

Moderne Betriebssysteme geben dem Benutzer die Illusion, dass verschiedene Programme gleichzeitig ausgeführt werden – die Betriebssysteme nennen sich multitaskingfähig. Was wir dann wahrnehmen, ist eine Quasiparallelität, die im Deutschen auch »Nebenläufigkeit«[1] genannt wird. Diese Nebenläufigkeit der Programme wird durch das Betriebssystem gewährleistet, welches auf Einprozessormaschinen die Prozesse alle paar Millisekunden umschaltet. Daher ist das Programm nicht wirklich parallel, sondern das Betriebssystem gaukelt uns dies durch verzahnte Bearbeitung der Prozesse vor. Ob wir aber nun einen oder beliebig viele kleine Männchen im Rechner arbeiten haben, soll uns egal sein.

Der Teil des Betriebssystems, der die Prozessumschaltung übernimmt, heißt *Scheduler*. Die dem Betriebssystem bekannten, aktiven Programme bestehen aus Prozessen. Ein Prozess setzt sich aus dem Programmcode und den Daten zusammen und besitzt einen eigenen Adressraum. Die virtuelle Speicherverwaltung des Betriebssystems trennt die Adressräume der einzelnen Prozesse. Dadurch ist es unmöglich, dass ein Prozess den Speicherraum eines anderen Prozesses korrumpiert; er sieht den anderen Speicherbereich nicht. Amok laufende Programme sind zwar möglich, werden jedoch vom Betriebssystem gestoppt. Zu jedem Prozess gehören des Weiteren Ressourcen wie geöffnete Dateien oder belegte Schnittstellen.

Es ist wünschenswert, Parallelität innerhalb eines einzigen Programms zur Verfügung zu haben und nicht nur innerhalb des Betriebssystems, welches ebenfalls quasiparallel ausführt.

Und da ist Java eine Sprache, die nebenläufige Programmierung direkt unterstützt. Das Konzept beruht auf so genannten *Threads* (zu Deutsch »Faden« oder »Ausführungsstrang«). Dies sind parallel ablaufende Aktivitäten, die sehr schnell in der Umschaltung sind. Innerhalb eines Prozesses kann es mehrere Threads geben, die alle zusammen in demselben Adressraum ablaufen. Die einzelnen Threads eines Prozesses können daher untereinander auf ihre öffentlichen Daten zugreifen.

Mehrere Threads können wie Prozesse verzahnt ausgeführt werden, so dass sich für den Benutzer der Eindruck von Gleichzeitigkeit ergibt. Trifft diese Aussage zu, dann nennen wir die Umgebung, in der das Programm abläuft, auch *multithreaded*. Threads können vom Scheduler sehr viel schneller umgeschaltet werden als Prozesse, so dass wenig Laufzeitverlust entsteht. Unterstützt das Betriebssystem des Rechners, auf dem die JVM läuft, Threads direkt, so nutzt die Laufzeitumgebung diese Fähigkeit in der Regel. In diesem Fall haben wir es mit *nativen Threads* zu tun. Falls das Betriebssystem jedoch keine Threads unterstützt, wird die Parallelität von der virtuellen Maschine simuliert. Der Java-Interpreter regelt dann den Ablauf, die Synchronisation und die verzahnte Ausführung. Die Sprachdefinition lässt die Art Implementierung – also nativ oder nicht – von Threads bewusst frei. Es ist aber wahrscheinlich, dass threadunterstützende Betriebssysteme die Thread-Verwaltung auch nativ umsetzen. Das ist jedoch nicht immer ideal: Native Threads haben einen höheren konstanten Speicher-Overhead.

Es stellt sich die Frage, warum denn nicht alle Laufzeitumgebungen die Threads auf das Betriebssystem abbilden. Dann wäre die JVM entlastet und die Verteilung auch auf Mehrprozessorsystemen geregelt. In diesem Fall brächte das aber den Nachteil mit sich, dass das Betriebssystem in den Threads auch Bibliotheksaufrufe ausführen kann, zum Beispiel, um das Eingabe- und Ausgabesystem zu verwenden oder um grafische Ausgaben zu machen. Damit das aber ohne Probleme funktioniert, müssen diese Bibliotheken threadsicher sein. Da hatten die Unix-Versionen jedoch diverse Probleme, insbesondere die grafische Standardbibliothek *X11* und *Motif* waren lange nicht threadsicher. Um schwer wiegenden Problemen mit grafischen Oberflächen aus dem Weg zu gehen, haben die Entwickler daher auf eine native Multithreaded-Umgebung zunächst verzichtet.

Mittlerweile unterstützen viele Betriebssysteme POSIX-Threads, und auch in C(++) wird paralleles Programmieren durch passende Bibliotheken populär. Doch besonders die Integration in die Sprache macht das Entwerfen nebenläufiger Anwendungen in Java einfacher. Durch die verzahnte Ausführung kommt es allerdings zu Problemen, die Datenbankfreunde von Transaktionen kennen. Es besteht die Gefahr konkurrierender Zugriffe auf gemeinsam genutzte Ressourcen. Um dies zu vermeiden, kann der Programmierer durch synchronisierte Programmblöcke gegenseitigen Ausschluss sicherstellen. Damit steigt aber auch die Gefahr für *Verklemmungen* (engl. *deadlocks*)

9.1.1 Wie parallele Programme die Geschwindigkeit steigern können

Auf den ersten Blick ist es nicht ersichtlich, warum auf einem Einprozessorsystem die nebenläufige Abarbeitung eines Programms geschwindigkeitssteigernd sein kann. Betrachten wir daher ein Programm, welches eine Folge von Anweisungen ausführt. Die Programmsequenz dient zum Visualisieren eines Datenbank-Reports. Zunächst wird ein Fenster zur Fortschrittsanzeige dargestellt. Anschließend werden die Daten analysiert, und der Fortschrittsbalken wird kontinuierlich aktualisiert. Schließlich werden die Ergebnisse in eine Datei geschrieben. Die Schritte sind:

1. Baue Fenster auf.

2. Öffne Datenbank vom Netz-Server und lese Datensätze.

3. Analysiere Daten und visualisiere den Fortschritt.

4. Öffne Datei und schreibe erstellten Report.

Was auf den ersten Blick wie ein typisches sequenzielles Programm aussieht, kann durch geschickte Parallelisierung beschleunigt werden.

Zum Verständnis ziehen wir noch einmal den Vergleich zu Prozessen. Nehmen wir an, auf einer Einprozessormaschine sind fünf Benutzer angemeldet, die im Editor Quelltext tippen und hin und wieder den Java-Compiler bemühen. Die Benutzer würden vermutlich die Belastung des Systems durch die anderen nicht mitbekommen, denn Editoroperationen lasten den Prozessor nicht aus. Wenn Dateien kompiliert und somit vom Hintergrundspeicher in den Hauptspeicher transferiert werden, ist der Prozessor schon besser ausgelastet, doch dies geschieht nicht regelmäßig. Im Idealfall übersetzen alle Benutzer nur dann, wenn die anderen gerade nicht übersetzen – im schlechtesten Fall möchten natürlich alle Benutzer gleichzeitig übersetzen.

Übertragen wir die Verteilung auf unser Problem, nämlich wie der Datenbank-Report schneller zusammengestellt werden kann. Beginnen wir mit der Überlegung, welche Operationen parallel ausgeführt werden können.

- Das Öffnen von Fenster, Ausgabedatei und Datenbank kann parallel geschehen.
- Das Lesen von neuen Datensätzen und das Analysieren von alten Daten kann gleichzeitig ablaufen.
- Alte analysierte Werte können während der neuen Analyse in die Datei geschrieben werden.

Wenn die Operationen wirklich parallel ausgeführt werden, kann bei Mehrprozessorsystemen ein enormer Leistungszuwachs verzeichnet werden. Doch interessanterweise ergibt sich dieser auch bei nur einem Prozessor, was in den Aufgaben begründet liegt. Denn bei den gleichzeitig auszuführenden Aufgaben handelt es sich um unterschiedliche Ressourcen. Wenn die grafische Oberfläche das Fenster aufbaut, braucht sie dazu natürlich Rechenzeit. Parallel kann die Datei geöffnet werden, wobei weniger Prozessorleistung gefragt ist als vielmehr die vergleichsweise träge Festplatte angesprochen wird. Das Öffnen der Datenbank wird auf den Datenbank-Server im Netzwerk abgewälzt. Die Geschwindigkeit hängt von der Belastung des Servers und des Netzes ab. Wenn anschließend die Daten gelesen werden, muss die Verbindung zum Datenbank-Server natürlich stehen. Daher sollten wir zuerst die Verbindung aufbauen.

Ist die Verbindung hergestellt, lassen sich über das Netzwerk Daten in einen Puffer holen. Der Prozessor wird nicht belastet, vielmehr der Server auf der Gegenseite und das Netzwerk. Während der Prozessor also vor sich hindöst und sich langweilt, können wir ihn derweil besser beschäftigen, indem er alte Daten analysiert. Wir verwenden hierfür zwei Puffer. In

den einen lädt ein Thread die Daten, während ein zweiter Thread die Daten im anderen Puffer analysiert. Dann werden die Rollen der beiden Puffer getauscht. Jetzt ist der Prozessor beschäftigt. Er ist aber vermutlich fertig, bevor die neuen Daten über das Netzwerk eingetroffen sind. In der Zwischenzeit können die Report-Daten in den Report geschrieben werden; eine Aufgabe, die wieder die Festplatte belastet und weniger den Prozessor.

Wir sehen an diesem Beispiel, dass durch hohe Parallelisierung eine Leistungssteigerung möglich ist, da die bei langsamen Operationen anfallenden Wartezeiten genutzt werden können. Langsame Arbeitsschritte lasten den Prozessor nicht aus, und die anfallende Wartezeit vom Prozessor beim Netzwerkzugriff auf eine Datenbank kann für andere Aktivitäten genutzt werden. Die Tabelle gibt die Elemente zum Kombinieren noch einmal an:

Tabelle 9.1 Parallelisierbare Ressourcen

Ressource	**Belastung**
Hauptspeicherzugriffe	Prozessor
Dateioperationen	Festplatte
Datenbankzugriff	Server, Netzwerkverbindung

Das Beispiel macht auch deutlich, dass die Nebenläufigkeit gut geplant werden muss. Nur wenn verzahnte Aktivitäten unterschiedliche Ressourcen verwenden, resultiert daraus auch auf Einprozessorsystemen ein Geschwindigkeitsvorteil. Daher ist ein paralleler Sortieralgorithmus nicht sinnvoll. Das zweite Problem ist die zusätzliche Synchronisation, die das Programmieren erschwert. Wir müssen auf das Ergebnis einer Operation warten, damit wir mit der Bearbeitung fortfahren können. Diesem Problem widmen wir uns in einem eigenen Abschnitt. Doch nun zur Programmierung von Threads in Java.

---

[1] Mitunter sind die Begriffe »parallel« und »nebenläufig« nicht äquivalent definiert. Wir wollen sie aber in diesem Zusammenhang synonym benutzen.

Kapitel 11 Datenstrukturen und Algorithmen

Algorithmen[1] sind ein zentrales Thema der Informatik. Ihre Erforschung und Untersuchung nimmt dort einen bedeutenden Platz ein. Algorithmen operieren nur dann effektiv mit Daten, wenn diese geeignet strukturiert sind. Schon das Beispiel Telefonbuch zeigt, wie wichtig die Ordnung der Daten nach einem Schema ist. Die Suche nach einer Telefonnummer bei gegebenem Namen gelingt schnell, jedoch ist die Suche nach einem Namen bei bekannter Telefonnummer ein mühseliges Unterfangen. Datenstrukturen und Algorithmen sind also eng miteinander verbunden, und die Wahl der richtigen Datenstruktur entscheidet über effiziente Laufzeiten; beide erfüllen alleine nie ihren Zweck. Leider ist die Wahl der »richtigen« Datenstruktur nicht so einfach, wie es sich anhört, und eine Reihe von schwierigen Problemen

in der Informatik sind wohl noch nicht gelöst, da eine passende Datenorganisation bis jetzt nicht gefunden wurde.

Die wichtigsten Datenstrukturen wie Listen, Mengen und Assoziativspeicher sollen in diesem Kapitel vorgestellt werden. In der zweiten Hälfte des Kapitels wollen wir uns dann stärker den Algorithmen widmen, die auf diesen Datenstrukturen operieren.

11.1 Mit einem Iterator durch die Daten wandern

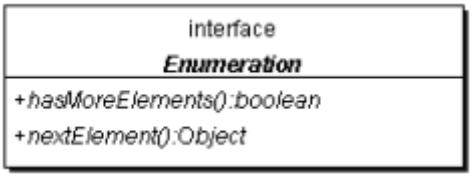
Wir wollen bei den Datenstrukturen eine Möglichkeit kennen lernen, wie sich die gespeicherten Daten unabhängig von der Implementierung immer mit derselben Technik abfragen lassen. Bei den Datenstrukturen handelt es sich meistens um Daten in Arrays, Bäumen oder Ähnlichem. Oft wird nur die Frage nach der Zugehörigkeit eines Werts zum Datenbestand gestellt, also: »Gehört das Wort dazu?«. Dieses *Wortproblem* ist durchaus wichtig, aber die Möglichkeit, die Daten in irgendeiner Weise aufzuzählen, ist nicht minder bedeutend. Bei Arrays können wir über den Index auf die Elemente zuzugreifen. Da wir jedoch nicht immer ein Array als Datenspeicher haben und uns auch die objektorientierte Programmierung verbietet, hinter die Kulisse zu sehen, benötigen wir möglichst einen allgemeineren Weg. Hier bieten sich *Enumeratoren* beziehungsweise *Iteratoren* an.

11.1.1 Die Schnittstellen Enumeration und Iterator

Für Iteratoren definiert die Java-Bibliothek zwei unterschiedliche Schnittstellen. Das hat historische Gründe. Die Schnittstelle Enumeration gibt es seit den ersten Java-Tagen; die Schnittstelle Iterator gibt es seit Java 1.2.

Die Schnittstelle Enumeration

Enumeration schreibt zwei Funktionen hasMoreElements() und nextElement() vor, mit denen durch einen Datengeber (in der Regel eine Datenstruktur) iteriert werden kann – wir sprechen in diesem Fall auch von einem *Iterator*. Bei jedem Aufruf von nextElement() erhalten wir ein weiteres Element der Datenstruktur. Im Gegensatz zum Index eines Felds können wir ein Objekt nicht noch einmal auslesen, nicht vorlaufen beziehungsweise hin und her springen. Ein Iterator gleicht anschaulich einem Datenstrom; wollten wir ein Element zweimal besuchen, zum Beispiel von rechts nach links noch einmal durchwandern, dann müssen wir wieder ein neues Enumeration-Objekt erzeugen oder uns die Elemente zwischendurch merken.



Hier klicken, um das Bild zu Vergrößern

interface java.util.**Enumeration<E>**

- boolean hasMoreElements()
  Testet, ob noch ein weiteres Element aufgezählt werden kann.[2]

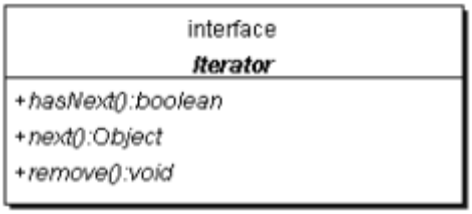
- E nextElement()
  Liefert das nächste Element der Enumeration zurück. Diese Funktion kann eine NoSuchElementException auslösen, wenn nextElement() aufgerufen wird und das Ergebnis false beim Aufruf von hasMoreElements() ignoriert wird.

Beispiel Die Aufzählung erfolgt meistens über einen Zweizeiler: Nehmen wir an, die Datenstruktur ds besitzt eine Methode elements(), die ein Enumeration-Objekt zurückgibt.

```
for ( Enumeration e = ds.elements(); e.hasMoreElements(); )
 System.out.println( e.nextElement() );
```

Die Schnittstelle Iterator

Ein Iterator ist für die neuen Collection-Klassen das, was Enumeration für die herkömmlichen Datenstruktur-Klassen ist. Die Schnittstelle Iterator besitzt kürzere Methodennamen als Enumeration. Nun heißt es hasNext() an Stelle von hasMoreElements() und next() an Stelle von nextElement(). Übergehen wir ein false von hasNext(), so erhalten wir wiederum eine NoSuchElementException. Zudem besitzt ein Iterator auch die Möglichkeit, das zuletzt aufgezählte Element aus dem zugrunde liegenden Container zu löschen. Dazu dient die optionale Methode remove(); sie lässt sich allerdings nur unmittelbar aufrufen, nachdem next() das zu löschende Element als Ergebnis geliefert hat. Eine Enumeration kann die aufgezählte Datenstruktur grundsätzlich nicht verändern.



Hier klicken, um das Bild zu Vergrößern

```
interface java.util.  Iterator<E>
```

- boolean hasNext()
  Liefert true, falls die Iteration weitere Elemente bietet.
- E next()
  Liefert das nächste Element in der Aufzählung oder NoSuchElementException, wenn keine weiteren Elemente mehr vorhanden sind.
- void remove()
  Entfernt das Element, das der Iterator zuletzt bei next() geliefert hat. Implementiert ein Iterator diese Funktion nicht, so löst er eine UnsupportedOperationException aus.

Hinweis Es ist eine interessante Frage, warum es die Methode remove() im Iterator gibt. Die Erklärung dafür ist, dass der Iterator die Stelle kennt, an der sich die Daten befinden (eine Art Cursor). Darum können die Daten auch effizient direkt dort gelöscht werden. Das erklärt jedoch nicht unbedingt, warum es keine Einfüge-Methode gibt. Ein allgemeiner Grund mag

sein, dass bei vielen Container-Typen das Einfügen an einer bestimmten Stelle keinen Sinn ergibt, etwa bei SortedSet, SortedMap, Set und Map. Dort ist die Einfügeposition durch die Sortierung vorgegeben oder belanglos (beziehungsweise bei HashSet durch die interne Realisierung bestimmt), also kein Fall für einen Iterator. Dazu wirft Einfügen weitere Fragen auf: Vor oder nach dem zuletzt per next() gelieferten Element? Soll das neue Element mit aufgezählt werden oder nicht? Auch dann nicht, wenn es in der Sortierung erst später an die Reihe käme? Eine Löschen-Methode ist problemloser und universell anwendbar.

---

[1] Das Wort »Algorithmus« geht auf den persisch-arabischen Mathematiker Ibn Mûsâ Al-Chwârismî zurück, der im 9. Jahrhundert lebte.

[2] Enumeratoren (und Iteratoren) können nicht serialisiert werden, da sie die Schnittstelle Serializable nicht implementieren.

Kapitel 12 Datenströme und Dateien

Computer sind uns so nützlich, da sie Daten bearbeiten. Dieser Bearbeitungszyklus beginnt beim Einlesen der Daten, beinhaltet das Verarbeiten und lässt die Ausgabe folgen. In der deutschen Literatur taucht dies als EVA[1]-Prinzip der Datenverarbeitungsanlagen auf. In frühen EDV-Zeiten wurde die Eingabe vom Systemoperator auf Lochkarten gestanzt, doch glücklicherweise sind diese Zeiten vorbei. Heutzutage speichern wir unsere Daten in Dateien (engl. *files*[2]) und auch in Datenbanken ab. Wichtig zu bemerken ist, dass eine Datei nur durch den Kontext interessant ist, andernfalls beinhaltet sie für uns keine Information – die Sichtweise auf eine Datei ist demnach wichtig. Auch ein Programm besteht aus Daten und wird oft in Form einer Datei repräsentiert.

Um an die Information einer Datei zu kommen, müssen wir den Inhalt auslesen können. Zudem müssen wir in der Lage sein, Dateien anzulegen, zu löschen, umzubenennen und sie in Verzeichnissen zu strukturieren. Java bietet uns eine Vielzahl von Zugriffsmöglichkeiten auf Dateien und ein wichtiges Schlagwort ist hierbei der *Datenstrom* (engl. *stream*). Dieser entsteht beim Fluss der Daten von der Eingabe zur Verarbeitung hin zur Ausgabe. Durch Datenströme können Daten sehr elegant bewegt werden, ein Programm ohne Datenfluss ist eigentlich nicht denkbar. Die Eingabeströme (engl. *input streams*) sind zum Beispiel Daten der Tastatur, Datei oder dem Netzwerk, und über die Ausgabeströme (engl. *output streams*) fließen die Daten in ein Ausgabemedium, beispielsweise Drucker oder Datei. Die Kommunikation der Threads geschieht über Pipes. Sie sind eine spezielle Variante der Datenströme.

In Java sind über dreißig Klassen zur Verarbeitung der Datenströme vorgesehen. Da die Datenströme allgemein und nicht an ein spezielles Ein- oder Ausgabeobjekt gebunden sind, können sie untereinander beliebig gemischt werden. Dies ist vergleichbar mit dem elektrischen Strom. Es gibt mehrere Stromlieferanten (Solarkraftwerke, Nutzung geothermischer Energie, Umwandlung von Meereswärmeenergie (OTEC)) und mehrere Verbraucher (Wärmedecke, Mikrowelle), die die Energie wieder umsetzen.

12.1 Datei und Verzeichnis

Da durch Datenströme keine Dateien gelöscht oder umbenannt werden können, liefert uns ein File-Objekt Informationen über Dateien und Verzeichnisse. Dieses Objekt wurde eingeführt, um Dateioperationen plattformunabhängig durchzuführen. Dies bedeutet aber leider auch eine Einschränkung, denn wie sollten Rechte vergeben werden, wenn etwa der Macintosh mit Mac OS 9 oder ein Palm-Pilot das nicht unterstützt? Auch Unix und Windows haben zwei völlig verschiedene Ansätze zur Rechteverwaltung.

12.1.1 Dateien und Verzeichnisse mit der Klasse File

Ein konkretes File-Objekt repräsentiert eine Datei oder ein Verzeichnis im Dateisystem. Der Verweis wird durch einen Pfadnamen spezifiziert. Dieser kann absolut oder relativ zum aktuellen Verzeichnis angegeben werden.[3]

Hinweis Pfadangaben sind plattformabhängig: Die Angabe des Pfads ist plattformabhängig, das heißt, auf Windows-Rechnern trennt ein Backslash die Pfade (»temp\doof«) und auf Unix-Maschinen ein normaler /, auch Divis genannt, (»temp/doof«). Glücklicherweise speichert die Klasse File im öffentlichen Attribut separatorChar den Dateitrenner.3 (Das kommt wiederum aus System.getProperty("file.separator".) Ebenso wie bei den Dateitrennern gibt es einen Unterschied in der Darstellung des Wurzelverzeichnisses. Unter Unix ist dies ein einzelnes Divis (»/«), und unter Windows ist die Angabe des Laufwerks vor den Doppelpunkt und das Backslash-Zeichen gestellt (»Z:\«).

```
class java.io.  File
implements Serializable, Comparable<File>
```

Wir können ein File-Objekt durch fünf Konstruktoren erzeugen:

- File(String pathname)
  Erzeugt ein File-Objekt mit komplettem Pfadnamen, zum Beispiel »d:\dali\die_anpassung_der_begierde«.
- File(String|File parent, String child)
  Pfadname und Dateiname sind getrennt.
- File(URI uri)
  Fragt von uri den Pfadnamen (uri.getPath()) und erzeugt ein neues File-Objekt. Ist uri null, folgt eine NullPointerException, ist die URI falsch formuliert, gibt es eine IllegalArgumentException.

URL-Objekte aus einem File-Objekt

Da es bei URL-Objekten recht häufig vorkommt, dass eine Datei die Basis ist, wurde die Methode toURL() der Klasse File aufgenommen. Es muss nur ein File-Objekt erzeugt werden, und anschließend erzeugt toURL() ein URL-Objekt, welches das Protokoll »file« trägt und eine absolute Pfadangabe zur Datei beziehungsweise zum Verzeichnis enthält. So

liefert new File("C:/Programme/").toURL() das URL-Objekt mit der URL file:/C:/Programme/.

Da diese Methode das Trennzeichen für die Pfade beachtet, ist die Angabe demnach auch passend für die Plattform. Ein Verzeichnis endet passend mit dem Pfadtrenner.

```
class java.io.  File
implements Serializable, Comparable<File>
```

- URL toURL() throws MalformedURLException
  Liefert ein URL-Objekt vom File-Objekt.
- URI toURI()
  Liefert ein URI-Objekt vom File-Objekt.

12.1.2 Dateieigenschaften und -attribute

Eine Datei oder ein Verzeichnis besitzt zahlreiche Eigenschaften, die sich mit Anfragemethoden auslesen lassen. In einigen wenigen Fällen lassen sich die Attribute auch ändern.

- boolean canRead()
  true, wenn wir lesend zugreifen dürfen
- boolean canWrite()
  true, wenn wir schreibend zugreifen dürfen
- boolean exists()
  true, wenn das File-Objekt existiert
- String getAbsolutePath()
  Liefert den absoluten Pfad. Ist das Objekt kein absoluter Pfadname, so wird ein String aus aktuellem Verzeichnis, Separator-Zeichen und Dateinamen des Objekts verknüpft.
- String getCanonicalPath() throws IOException
  File getCanonicalFile() throws IOException
  Gibt den Pfadnamen des Dateiobjekts zurück, der keine relativen Pfadangaben mehr enthält. Kann im Gegensatz zu den anderen Pfadfunktionen eine IOException aufrufen, da mitunter verbotene Dateizugriffe erfolgen.
- String getName()
  Gibt Dateinamen zurück.
- String getParent()
  Gibt Pfadnamen des Vorgängers zurück.
- String getPath()
  Gibt Pfadnamen zurück.
- boolean isAbsolute()
  true, wenn der Pfad in der systemabhängigen Notation absolut ist
- boolean isDirectory()
  Gibt true zurück, wenn es sich um ein Verzeichnis handelt.

- boolean isFile()
  true, wenn es sich um eine »normale« Datei handelt (kein Verzeichnis und keine Datei, die vom zugrunde liegenden Betriebssystem als besonders markiert wird; Blockdateien, Links unter Unix). In Java können nur normale Dateien erzeugt werden.
- long length()
  Gibt die Länge der Datei in Bytes zurück oder 0L, wenn die Datei nicht existiert oder es sich um ein Verzeichnis handelt.

Beispiel Um festzustellen, ob f das Wurzelverzeichnis ist, genügt folgende Zeile:

```
if ( f.getPath().equals(f.getParent()) )
 // File f ist root
```

Beispiel Liefere einen Dateinamen, bei dem die relativen Bezüge aufgelöst sind.

```
try
{
  System.out.println( new
File("C:/./WasNDas//..\\Programme").getCanonicalFile() );
}
catch ( IOException e ) { }
```

Das Ergebnis ist *C:\Programme*.

Änderungsdatum einer Datei

Eine Datei besitzt unter jedem Dateisystem nicht nur Attribute wie Größe und Rechte, sondern verwaltet auch das Datum der letzten Änderung. Letzteres nennt sich *Zeitstempel*. Die File-Klasse besitzt zum Abfragen dieser Zeit die Methode lastModified(). Mit der Methode setLastModified() lässt sich die Zeit auch setzen. Dabei bleibt es etwas verwunderlich, warum lastModified() nicht als veraltet ausgezeichnet ist und zu getLastModified() geworden ist, wo doch nun die passende Funktion zum Setzen der Namensgebung genügt.

```
class java.io.  File
implements Serializable, Comparable<File>
```

- long lastModified()
  Liefert den Zeitpunkt, an dem die Datei zum letzten Mal geändert wurde. Die Zeit wird in Millisekunden ab dem 1. Januar 1970, 00:00:00 GMT gemessen. Die Methode liefert null, wenn die Datei nicht existiert oder ein Ein- beziehungsweise Ausgabefehler auftritt.
- boolean setLastModified(long time)
  Setzt die Zeit, an dem die Datei zuletzt geändert wurde. Die Zeit ist wiederum in Millisekunden seit dem 1. Januar 1970 angegeben. Ist das Argument negativ, dann wird eine IllegalArgumentException ausgeworfen.

Die Methode setLastModified() ändert – wenn möglich – den Zeitstempel, und ein anschließender Aufruf von lastModified() liefert die gesetzte Zeit – womöglich gerundet – zurück. Die Funktion ist von vielfachem Nutzen, aber sicherheitsbedenklich. Denn ein Programm kann den Dateiinhalt ändern und den Zeitstempel dazu. Auf den ersten Blick ist nicht mehr sichtbar, dass eine Veränderung der Datei vorgenommen wurde. Doch die Funktion ist von größerem Nutzen bei der Programmerstellung, wo Quellcodedateien etwa mit Objektdateien verbunden sind. Nur über einen Zeitstempel ist eine einigermaßen intelligente Projektdateiverwaltung möglich.

Hinweis: Zwar erlaubt Java, die die Zeit des letzten Zugriffs zu ermitteln, das gilt aber nicht für die Erzeugungszeit. Das ist nicht böswillig, denn ein Unix-System speichert diese Zeit zum Beispiel gar nicht. Windows speichert diese Zeit schon, so dass hier grundsätzlich der Zugriff, etwa über JNI, möglich wäre. Legt ein Java-Programm die Dateien an, dessen Anlegezeiten später wichtig sind, müssen die Zeiten beim Anlegen gemessen und gespeichert werden. Falls die Datei nicht verändert wird, stimmt lastModified() mit der Anlegzeit überein.

12.1.3 Dateien berühren, neue Dateien anlegen

Unter dem Unix-System gibt es das Shell-Kommando *touch*, welches wir in einer einfachen Variante in Java umsetzen wollen. Das Programm berührt (engl. *touch*) eine Datei, indem der Zeitstempel auf das aktuelle Datum gesetzt wird. Da es mit setLastModified() einfach ist, das Zeitattribut zu setzen, muss die Datei nicht geöffnet, das erste Byte gelesen und gleich wieder geschrieben werden. Wie beim Kommando *touch* soll unser Java-Programm über alle auf der Kommandozeile übergebenen Dateien gehen und sie berühren. Falls eine Datei nicht existiert, soll sie kurzerhand angelegt werden. Gibt setLastModified() den Wahrheitswert false zurück, so wissen wir, dass die Operation fehlschlug, und geben eine Informationsmeldung aus.

Listing 12.1 Listing 12.1 Touch.java

```
import java.io.*;

public class Touch
{
  public static void main( String args[] ) throws IOException
  {
    for ( int i = 0; i < args.length; i++ )
    {
      File f = new File( args[i] );

      if ( f.exists() )
      {
        if ( f.setLastModified( System.currentTimeMillis() ) )
          System.out.println( "touched " + args[i] );
        else
          System.out.println( "touch failed on " + args[i] );
      }
      else
      {
        f.createNewFile();
        System.out.println( "create new file " + args[i] );
      }
    }
```

```
  }
}
```

```
class java.io.  File
implements Serializable, Comparable<File>
```

- boolean createNewFile() throws IOException
  Legt eine neue Datei, wenn noch keine existiert.
- static File createTempFile(String prefix, String suffix) throws IOException
  Legt eine neue Datei im temporären Verzeichnis an. Das Verzeichnis ist etwa */temp* unter Unix oder *c:/temp* unter Windows. Der Dateiname setzt sich zusammen aus einem benutzerdefinierten Präfix, einer Zufallsfolge und einem Suffix.
- static File createTempFile(String prefix, String suffix, File directory)
  throws IOException
  Legt eine neue Datei im gewünschten Verzeichnis an. Der Dateiname setzt sich zusammen aus einem benutzerdefinierten Präfix, einer Zufallsfolge und einem Suffix.

12.1.4 Umbenennen und Verzeichnisse anlegen

Mit mkdir() lassen sich Verzeichnisse anlegen und mit renameTo() Dateien oder Verzeichnisse umbenennen.

```
class java.io.  File
implements Serializable, Comparable<File>
```

- boolean mkdir()
  Legt das Unterverzeichnis an.
- boolean mkdirs()
  Legt das Unterverzeichnis inklusive weiterer Verzeichnisse an.
- boolean renameTo(File d)
  Benennt die Datei in den Namen um, der durch das File-Objekt d gegeben ist. Ging alles gut, wird true zurückgegeben. Bei zwei Dateinamen alt und neu benennt new File(alt).renameTo(new File(neu)); die Datei um.

Über renameTo() sollte noch ein Wort verloren werden: File-Objetke sind immutable, stehen also immer nur für genau eine Datei. Ändert sich der Dateiname, ist das File-Objekt ungültig und kein Zugriff mehr über dieses File-Objekt erlaubt. Auch wenn eine Laufzeitumgebung keine Exception auswirft, sind alle folgenden Ergebnisse von Anfragen unsinnig.

Zum Teil kann renameTo() auch zum Verschieben von Dateien dienen. Oftmals gilt aber die Einschränkung, das dies nur auf einem Datenträger möglich ist (also etwa von Laufwerk *C* nach *C:/*, aber nicht von *C* nach *D:/*), Links und sonstigen Dateivarianten nicht eingerechnet.

12.1.5 Die Wurzel aller Verzeichnisse/Laufwerke

Die statische Funktion listRoots() gibt ein Feld von File-Objekten zurück, die eine Auflistung der Wurzeln (engl. *root*) von Dateisystemen enthält. Dies macht es einfach, Programme zu schreiben, die etwa über dem Dateisystem eine Suche ausführen. Da es unter Unix nur eine

Wurzel gibt, ist der Rückgabewert von File.listRoots() immer »/« – ein anderes Root gibt es nicht. Unter Windows wird es aber zu einem richtigen Feld, da es mehrere Wurzeln für die Partitionen oder logischen Laufwerke gibt. Die Wurzeln tragen Namen wie »A:« oder »Z:«. Dynamisch eingebundene Laufwerke, die etwa unter Unix mit »mount« integriert werden, oder Wechselfestplatten werden mit berücksichtigt. Die Liste wird immer dann aufgebaut, wenn listRoots() aufgerufen wird. Komplizierter ist es, wenn entfernte Dateibäume mittels NFS oder SMB eingebunden sind. Denn dann kommt es darauf an, ob das zuständige Programm eine Verbindung noch aktiv hält oder nicht. Denn nach einer abgelaufenen Zeit ohne Zugriff wird das Verzeichnis wieder aus der Liste genommen. Dies ist aber wieder sehr plattformabhängig.

```
class java.io.  File
implements Serializable, Comparable<File>
```

- static File[] listRoots()
  Liefert die verfügbaren Wurzeln der Dateisysteme oder null, falls diese nicht festgestellt werden können. Jedes File-Objekt beschreibt eine Dateiwurzel. Es ist gewährleistet, dass alle kanonischen Pfadnamen mit einer der Wurzeln beginnen. Wurzeln, für die der SecurityManager den Zugriff verweigert, werden nicht aufgeführt. Das Feld ist leer, falls es keine Dateisystem-Wurzeln gibt.

Liste der Wurzeln/Laufwerke ausgeben

Im folgenden Beispiel wird ein Programm vorgestellt, das mit listRoots() eine Liste der verfügbaren Wurzeln ausgibt. Dabei berücksichtigt das Programm, ob auf das Gerät eine Zugriffsmöglichkeit besteht. Unter Windows ist etwa ein Diskettenlaufwerk eingebunden, aber wenn keine Diskette im Schacht ist, ist das Gerät nicht bereit. Das Diskettenlaufwerk taucht in der Liste auf, aber exists() liefert false.

Listing 12.2 ListRoots.java

```
import java.io.*;

public class ListRoots
{
  public static void main( String args[] )
  {
    for ( File file : File.listRoots() )
      System.out.println( file.getPath() + " ist " +
                          (file.exists() ? "" : "nicht ") + "bereit" );  }
}
```

Bei der Ausgabe mit System.out.println() entspricht root.getPath() einem root.toString(). Demnach könnte das Programm etwas abgekürzt werden, etwa mit root + " XYZ". Da aber nicht unbedingt klar ist, dass toString() auf getPath() verweist, schreiben wir getPath() direkt.

12.1.6 Verzeichnisse listen und Dateien filtern

Um eine Verzeichnisanzeige oder einen Dateiauswahldialog zu programmieren, benötigen wir eine Liste von Dateien, die in einem Verzeichnis liegen. Ein Verzeichnis kann reine Dateien

oder auch wieder Unterverzeichnisse besitzen. Die list()- und listFiles()-Funktionen der Klasse File geben ein Feld von Zeichenketten mit Dateien und Verzeichnissen beziehungsweise ein Feld von File-Objekten mit den beinhalteten Elementen zurück.

```
class java.io.  File
implements Serializable, Comparable<File>
```

- String[] list()
  Gibt eine Liste der Dateien zurück. Diese enthält weder ».« noch »..«.
- File[] listFiles()
  Gibt eine Liste der Dateien als File-Objekte zurück.

Beispiel Ein einfacher Directory-Befehl ist leicht in ein paar Zeilen programmiert.

```
String entries[] = new File(".").list();
System.out.println( Arrays.asList(entries) );
```

Die einfache Funktion list() liefert dabei nur relative Pfade, also einfach den Dateinamen oder den Verzeichnisnamen. Den absoluten Namen zu einer Dateiquelle müssen wir also erst zusammensetzen. Praktischer ist da schon die Methode listFiles(), da wir hier komplette File-Objekte bekommen, die ihre ganze Pfadangabe schon kennen. Wir können den Pfad mit getName() erfragen.

Dateien nach Kriterien filtern mit FilenameFilter und FileFilter

Sollen aus einer Liste von Dateien einige mit bestimmten Eigenschaften (zum Beispiel der Endung) herausgenommen werden, so müssen wir dies nicht selbst programmieren. Schlüssel hierzu ist die Schnittstelle FilenameFilter und FileFilter. Wenn wir etwas später den grafischen Dateiselektor kennen lernen, so können wir dort auch den FilenameFilter einsetzen. Leider ließ der Fehlerteufel seine Finger nicht aus dem Spiel und der FilenameFilter funktioniert nicht, da der FileSelector fehlerhaft ist.[4]

Ein Filter filtert aus den Dateinamen diejenigen heraus, die einem gesetzten Kriterium genügen. Eine Möglichkeit ist, nach den Endungen zu separieren. Doch auch komplexere Selektionen sind denkbar; so kann in die Datei hineingesehen werden, ob sie beispielsweise bestimmte Informationen am Dateianfang enthält. Besonders für Macintosh-Benutzer ist dies wichtig zu wissen, denn dort sind die Dateien nicht nach Endungen sortiert. Die Information liegt in den Dateien selbst. Windows versucht uns auch diese Dateitypen vorzuenthalten, aber von dieser Kennung hängt alles ab. Wer die Endung einer Grafikdatei schon einmal umbenannt hat, der weiß, warum Grafikprogramme aufgerufen werden. Von den Endungen hängt also sehr viel ab.

```
class java.io.  File
implements Serializable, Comparable<File>
```

- public String[] list(FilenameFilter filter)

Wie list(), nur filtert ein spezielles FilenameFilter-Objekt Objekte heraus.

- public File[] listFiles(FilenameFilter filter)
  Wie listFiles(), nur filtert ein spezielles FilenameFilter-Objekt Objekte heraus.
- public File[] listFiles(FileFilter filter)
  Wie list(), nur filtert ein spezielles FileFilter-Objekt bestimmte Objekte heraus.

interface java.io. **FileFilter**

- boolean accept(File pathname)
  Liefert true, wenn der Pfad in die Liste aufgenommen werden soll.

interface java.io. **FilenameFilter**

- boolean accept(File dir, String name)
  Testet, ob die Datei name im Verzeichnis dir in der Liste auftreten soll. Gibt true zurück, wenn dies der Fall ist.

Eine Filter-Klasse implementiert die Funktion accept() von FilenameFilter so, dass alle von accept() angenommenen Dateien den Rückgabewert true liefern.

Beispiel Wollen wir nur auf Textdateien reagieren, so geben wir ein true bei allen Dateien mit der Endung *.txt* zurück. Die anderen werden mit false abgelehnt.

```
class FileFilter implements FilenameFilter
{
  public boolean accept( File f, String s )
  {
    return s.toLowerCase().endsWith( ".txt" );
  }
}
```

Nun kann list() mit dem FilenameFilter aufgerufen werden. Wir bekommen eine Liste mit Dateinamen, die wir in einer Schleife einfach ausgeben. An dieser Stelle merken wir schon, dass wir nur für FilenameFilter eine neue Klasse schreiben müssen. Hier bietet sich wieder eine innere Klasse an.

Beispiel list() soll nur Verzeichnisse zurückliefern:

```
String a[] = entries.list( new FilenameFilter() {
  public boolean accept( File d, String name ) {
    return d.isDir();
  } } );
```

Die einfache Implementierung von list() mit FilenameFilter

Die Methode list() holt zunächst ein Feld von Dateinamen ein. Nun wird jede Datei mittels der accept()-Methode geprüft und in eine interne ArrayList übernommen. Nach dem Testen jeder Datei wird das Array mit ((String[])(v.toArray(new String[0])) in ein Feld von Strings kopiert. Bei listFiles() steht anstelle von String dann File.

Dateien aus dem aktuellen Verzeichnis filtern

Wir können somit ein einfaches Verzeichnisprogramm programmieren, indem wir die Funktionen von getProperty() und list() zusammenfügen. Zusätzlich wollen wir nur Dateien mit der Endung *.txt* angezeigt bekommen.

Listing 12.3 Dir.java

```
import java.io.*;

class TxtFilenameFilter implements FilenameFilter
{
  public boolean accept( File f, String s )
  {
    return s.toLowerCase().endsWith(".txt");
  }
}

public class Dir
{
  public static void main( String args[] )
  {
    File userdir = new File( System.getProperty("user.dir") );
    System.out.println( userdir );

    String entries[] = userdir.list( new TxtFilenameFilter() );

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

12.1.7 Dateien und Verzeichnisse löschen

Mit Hilfe der Funktion delete() auf einem File-Objekt lässt sich eine Datei oder ein Verzeichnis entfernen. Diese Methode löscht wirklich! Sie ist nicht so zu verstehen, dass sie true liefert, falls die Datei potenziell gelöscht werden kann. Konnte die Laufzeitumgebung delete() nicht ausführen, so sollte die Rückgabe false sein. Ein zu löschendes Verzeichnis muss leer sein, andernfalls kann das Verzeichnis nicht gelöscht werden. Unsere unten stehende Implementierung geht dieses Problem so an, dass es rekursiv die Unterverzeichnisse löscht.

Hinweis Auf manchen Systemen liefert delete() die Rückgabe true, die Datei ist aber nicht gelöscht. Grund kann eine noch geöffnete Datei sein, mit der zum Beispiel ein Eingabestrom verbunden ist.

```
class java.io.  File
implements Serializable, Comparable<File>
```

- boolean delete()
  Löscht die Datei oder das leere Verzeichnis.
- void deleteOnExit()
  Löscht die Datei/das Verzeichnis, wenn die virtuelle Maschine korrekt beendet wird. Einmal vorgeschlagen, kann das Löschen nicht mehr rückgängig gemacht werden.

Die delete()-Funktion funktioniert auf File-Objekten, die Dateien und auch Verzeichnisse repräsentieren. Doch sind Verzeichnisse nicht leer, so wird delete() nicht auch noch alle Dateien in diesem Verzeichnis inklusive aller Unterverzeichnisse löschen. Das muss von Hand gemacht werden, lässt sich aber in wenigen Programmcodezeilen rekursiv umsetzen. Eine Funktion deleteTree() soll einen Baum inklusive Unterverzeichnisse löschen. list() liefert ein Feld aller Elemente in dem Unterverzeichnis, und falls ein Element wiederum ein Verzeichnis ist, wird wieder deleteTree() auf diesem aufgerufen.

Listing 12.4 DeleteTree.java

```
import java.io.*;

public class DeleteTree
{
  public static void deleteTree( File path )
  {
    for ( File file : path.listFiles() )
    {
      if ( file.isDirectory() )
        deleteTree( file );

      file.delete();
    }

    path.delete();
  }

  public static void main( String args[] )
  {
    deleteTree( new File("c:/temp/Kopie von kai") );
  }
}
```

12.1.8 Implementierungsmöglichkeiten für die Klasse File

Wir sollten für die Klasse File noch einmal festhalten, dass sie lediglich ein Verzeichnis oder eine Datei im Verzeichnissystem repräsentiert, jedoch keine Möglichkeit bietet, auf die Daten selbst zuzugreifen. Es ist ebenso offensichtlich, dass wir mit den plattformunabhängigen Eigenschaften von Java spätestens bei Zugriffen auf das Dateisystem nicht mehr weiter kommen. Wenn zum Beispiel eine Datei gelöscht werden soll oder wir eine Liste von Dateien

im Verzeichhnis erbitten, sind Betriebssystemfunktionen mit von der Partie. Diese sind dann nativ programmiert. Es bieten sich zwei Möglichkeiten für die Implementierung an.

- Zunächst ist es denkbar, die nativen Methoden in der Klasse File selbst anzugeben. Diesen Weg ging die Sun-Implementierung eine ganze Zeit lang, und Kaffe nutzt diese Variante heute noch. Doch die Verschmelzung von einem Dateisystem in der Klasse File bietet auch Nachteile. Was ist, wenn das Dateisystem eine Datenbank ist, so dass die typischen nativen C-Funktionen unpassend sind?
- Aus dieser Beschränkung heraus hat sich Sun dazu entschlossen, eine nur paketsichtbare, abstrakte Klasse FileSystem einzufügen, die ein abstraktes Dateisystem repräsentiert.

Jedes Betriebssystem implementiert folglich eine konkrete Unterklasse für FileSystem, die dann von File genutzt werden kann. File ist dann völlig frei von nativen Methoden und leitet alles an das FileSystem-Objekt weiter, das intern mit folgender Zeile angelegt wird:

```
static private FileSystem fs = FileSystem.getFileSystem();
```

Dann ergibt sich zum Beispiel für den Zugriff auf die Länge einer Datei:

```
public long length() {
  SecurityManager security = System.getSecurityManager();
  if ( security != null )
    security.checkRead( path );
   return fs.getLength( this );
}
```

Wenn wir dies noch einmal mit dem ersten Weg vergleichen, dann finden wir in der File-Implementierung von Kaffe etwa Folgendes:

```
public class File implements Serializable, Comparable
{
  static {
    System.loadLibrary( "io" );
  }

  public long length() {
    checkReadAccess();
    return length0();
  }

  native private long length0();
}
```

Die native Methode ist selbstverständlich privat, denn sonst gäbe es ja keine Sicherheitsprüfung. Oft trägt sie die Endung 0, so dass eine Unterscheidung einfach ist.

Um das Geheimnis um die native Methode length0() zu lüften und einen Eindruck von nativen Methoden zu vermitteln, gönnen wir uns einen Blick auf die Implementierung:

```
jlong
java_io_File_length0( struct Hjava_io_File* this )
```

```
{
  struct stat buf;
  char str[MAXPATHLEN];
  int r;

  stringJava2CBuf( unhand(this)->path, str, sizeof(str) );

  r = KSTAT( str, &buf );
  if ( r != 0 )
    return (jlong)0;

  return (jlong)buf.st_size;
}
```

Der Aufruf stringJava2CBuf() konvertiert die Unicode-Zeichenfolge in eine gültige C–Zeichenkette, die mit einem Null-Byte abschließt. Es folgt der Aufruf einer ANSI-Bibliotheksfunktion, die noch über KSTAT gekapselt ist. Der Datentyp jlong ist kein C(++)-Datentyp, sondern in der javaspezifischen Header-Datei definiert.

Wenn wir uns etwas später mit dem Zugriff über Stream-Klassen beschäftigen, dann benutzt auch diese Klasse native Methoden und eine abstrakte Repräsentation eines Dateideskriptors.

12.1.9 Verzeichnisse nach Dateien rekursiv durchsuchen

Im vorausgehenden Kapitel haben wir einige Datenstrukturen kennen gelernt, unter anderem Vector und Stack. Wir wollen damit ein Programm formulieren, welches rekursiv die Verzeichnisse durchläuft und nach Dateien durchsucht. Die Vector-Klasse dient dazu, die Dateien zu speichern, und mit dem Stack merken wir uns die jeweiligen Verzeichnisse (Tiefensuche), in die wir absteigen. Anders als bei DeleteTree nutzt diese Implementierung also keine Rekursion.

Listing 12.5 FileFinder.java

```
import java.io.*;
import java.util.*;

public class FileFinder
{
  public List<File> files = new ArrayList<File>( 1024 );

  public static void main( String args[] )
  {
    String path = new File(System.getProperty("user.dir")).getParent();

    System.out.println( "Looking in path: " + path );

    FileFinder ff = new FileFinder( path,  // TODO: 1.5
                                    new String[] { ".gif", ".jpg", ".tif" }
);
    ff.print();
  }

  public FileFinder( String start, String extensions[] )  // TODO: Java 5
mit ...
  {
    Stack<File> dirs = new Stack<File>();
```

```
    File startdir = new File( start );

    if ( startdir.isDirectory() )
      dirs.push( startdir );

    while ( dirs.size() > 0 )
    {
      for ( File file : dirs.pop().listFiles() )
      {
        if ( file.isDirectory() )
          dirs.push( file );
        else
          if ( match(file.getName(), extensions) )
            files.add( file );
      }
    }
  }

  public void print()
  {
    System.out.println( "Found " + files.size() + " file" +
                        (files.size() == 1 ? "." : "s.") );

    for ( File f : files )
      System.out.println( f.getAbsolutePath() );
  }

  private static boolean match( String s, String suffixes[] )
// In Java 5 ... möglich
  {
    for ( String suffix : suffixes )
      if ( s.length() >= suffix.length() &&
           s.substring(s.length() - suffix.length(),
                       s.length()).equalsIgnoreCase(suffix) )
        return true;

    return false;
  }
}
```

12.1.10 Sicherheitsprüfung

Wir müssen uns bewusst sein, dass verschiedene Methoden eine SecurityException auslösen können, sofern ein Security-Manager die Dateioperationen überwacht. Security-Manager kommen beispielsweise bei Applets zum Zuge. Folgende Methoden sind Kandidaten für eine SecurityException: exists(), canWrite(), canRead(), canWrite(), isDirectory(), lastModified(), length(), mkdir(), mkdirs(), list(), delete() und renameFile().

12.1.11 Namen der Laufwerke

Die Namen der Laufwerksbuchstaben sind ein wenig versteckt, denn es ist nicht bei der Klasse File zu finden. Zwar liefert listRoots() schon einen passenden Anfang, um unter Windows die Laufwerke preiszugeben, aber die Namen liefert erst getSystemDisplayName() des FileSystemView-Objekts. Die Klasse gehört zu Swing und dort zum Dateiauswahldialog.

```
FileSystemView view = FileSystemView.getFileSystemView();
```

```
for ( File f : File.listRoots() )
  System.out.println( view.getSystemDisplayName(f) );
```

FileSystemView hält noch andere gute Funktionen bereit:

```
abstract class javax.swing.filechooser.FileSystemView
```

- boolean isDrive(File dir)
  Ist dir ein Laufwerk?
- boolean isFloppyDrive(File dir)
  Ist dir ein Wechsellaufwerk?
- boolean isComputerNode(File dir)
  Ist dir ein Netzwerk-Knoten?

12.1.12 Locking

Damit eine Datei gegen parallelen Zugriff gesperrt ist, kann man sie Locken. Das ist seit 1.4 auf machen Betriebssystemen möglich. Ein Betriebssystem wie Unix macht es im Allgemeinen schwer, da es keine Locks unterstützt. So kann unter Unix eine Datei auch von mehreren Seiten gelesen werden, auch wenn sie zum Beispiel aktuell beschrieben wird. Auch kann eine Datei auf dem Datensystem gelöscht werden, wenn sich noch geöffnet ist. Das Windows Betriebssystem unterstützt dagegen Locks. Wenn ein Prozess keinen Lock auf die Datei besitzt, kann der Prozess die Datei auch nicht lesen.

Um einen Lock zu erwerben, ist die Klasse FileChannel und deren Funktion lock() zu nutzen. Um zu testen, ob eine gegebene Datei gelockt ist, lässt sich tryLock() verwenden – etwa in folgender Funktion:

```
public boolean isLocked( String filename )
{
  try
  {
    new RandomAccessFile( filename,"r" ).getChannel().tryLock();
  }
  catch( IOException e ) {
     return false;
  }
  return true;
}
```

Falls nun Locking unter Unix-Systemen gewünscht ist, bleibt nichts anderes übrig, als das Locking nachzuimplementieren. Das gelingt durch Umkopieren oder Umbenennen.

---

[1] EVA ist ein Akronym für »Eingabe, Verarbeitung, Ausgabe«. Diese Reihenfolge entspricht dem Arbeitsweg. Zunächst werden die Eingaben von einem Eingabegerät gelesen, dann durch den Computer verarbeitet und anschließend ausgegeben (in welcher Form auch immer).

[2] Das englische Wort »file« geht auf das lateinische Wort filum zurück. Dies bezeichnete früher eine auf Draht aufgereihte Sammlung von Schriftstücken.

[3] Ein Wunder, warum das nicht großgeschrieben ist! Dabei ist die Variable public static final char, also eine waschechte Konstante.

[4] Obwohl die Funktionalität dokumentiert ist, findet sich unter der Bug-Nummer 4031440 kurz: »The main issue is that support for FilenameFilter in the FileDialog class was never implemented on any platform – it's not that there's a bug which needs to be fixed, but that there's no code to run nor was the design ever evaluated to see if it *could* be implemented on our target platforms«.