

Kapitel 2

Objekte, Klassen, Kapselung

Dieses Kapitel erläutert, wie Objekte beschrieben werden können, und geht dazu ausführlich auf das Klassenkonzept ein. Es behandelt Kapselungsaspekte und stellt Techniken vor, mit denen sich Mengen von Klassen strukturieren und modularisieren lassen. Zu allen diesen Aspekten werden die entsprechenden Sprachkonstrukte von Java vorgestellt. Um die konzeptionelle Trennung zwischen Klassenkonzept einerseits und Vererbung andererseits deutlich zu machen, werden Subtyping und Vererbung erst im nächsten Kapitel behandelt (siehe Kap. 3).

Dieses Kapitel besteht aus zwei Abschnitten. Im Mittelpunkt des ersten Abschnitts steht die Deklaration und Verwendung von Klassen. Dabei wird auch auf den Entwurf von Klassen eingegangen, und es werden weitere in diesem Zusammenhang wichtige Aspekte behandelt, insbesondere:

- rekursive Abhängigkeiten zwischen Klassen,
- parametrische Klassen, d.h. Klassen mit Typparametern,
- die Verwendung von Klasseninformation zur Programmlaufzeit.

Der zweite Abschnitt erläutert Techniken und Konstrukte zur Kapselung von Implementierungsteilen und geht auf die Schachtelung von Klassen und die Zusammenfassung von Klassen in Pakete/Module ein.

2.1 Objekte und Klassen

Dieser Abschnitt behandelt die Frage: Was sind Objekte und wie werden sie in Programmen beschrieben? Er geht dazu vom Grundmodell der objektorientierten Programmierung aus (vgl. die Abschnitte 1.1.1 und 1.2.3) und stellt Sprachkonzepte zur Beschreibung von Objekten vor. Sein Hauptteil widmet sich dem Klassenkonzept von Java und dessen sprachlichen Erweiterungen. Schließlich erläutert er, inwiefern auch Klassen als Objekte betrachtet werden können und wie diese Sichtweise in Java unterstützt wird.

Programmiersprachen, die es ermöglichen, Objekte zu definieren, zu erzeugen und ihnen Nachrichten zu schicken, die aber weder Vererbung noch

Subtyping unterstützen, werden vielfach *objektbasierte* Sprachen genannt. Dieser Abschnitt beschäftigt sich also insbesondere mit dem objektbasierten Kern der Sprache Java.

2.1.1 Beschreibung von Objekten

Die objektorientierte Programmierung betrachtet eine Programmausführung als ein System kooperierender Objekte. Ein objektorientiertes Programm muss also im Wesentlichen beschreiben, wie die Objekte aussehen, die während der Programmausführung existieren, wie sie erzeugt bzw. gelöscht werden und wie die Kommunikation zwischen den Objekten angestoßen wird. Grundsätzlich gibt es zwei Konzepte zur programmiersprachlichen Beschreibung von Objekten:

1. *Klassenkonzept*: Der Programmierer beschreibt nicht einzelne Objekte, sondern deklariert Klassen. Eine Klasse ist eine *Beschreibung* der Eigenschaften, die die Objekte dieser Klasse haben sollen. Die Programmiersprache ermöglicht es dann, während der Programmausführung Objekte zu deklarierten Klassen zu erzeugen. Die Klassen können zur Ausführungszeit nicht verändert werden. Klassendeklarationen entsprechen damit der Deklaration von Verbundtypen imperativer Sprachen, bieten aber zusätzliche Möglichkeiten (vgl. Abschn. 1.3.2).
2. *Prototyp-Konzept*: Der Programmierer beschreibt direkt einzelne Objekte. Neue Objekte werden durch Klonen existierender Objekte und Verändern ihrer Eigenschaften zur Ausführungszeit erzeugt. *Klonen* eines Objekts *obj* meint dabei das Erzeugen eines neuen Objekts, das die gleichen Eigenschaften wie *obj* besitzt. Verändern bedeutet zum Beispiel, dass dem Objekt weitere Attribute hinzugefügt werden oder dass Methoden durch andere ersetzt werden.

Das Klassenkonzept ermöglicht eine bessere statische Überprüfung von Programmen, da zur Übersetzungszeit alle wesentlichen Eigenschaften der Objekte bekannt sind. Insbesondere kann Typkorrektheit überprüft werden und damit einhergehend, ob jedes Objekt, dem eine Nachricht *m* geschickt wird, auch eine Methode besitzt, um *m* zu bearbeiten. Beim Prototyp-Konzept können diese Prüfungen im Allgemeinen nur dynamisch durchgeführt werden, da die Eigenschaften der Objekte erst zur Ausführungszeit festgelegt werden. Dafür ist das Prototyp-Konzept tendenziell flexibler (vgl. [Bla94] bzgl. der Programmierung mit Prototypen). Beim Entwurf einer Programmiersprache müssen diese Vor- und Nachteile abgewogen werden. Wie wir in Abschn. 2.1.7 sehen werden, sind auch Mischformen zwischen den beiden Konzepten möglich. Zunächst behandeln wir allerdings das Klassenkonzept

und seine Ausprägung in Java. In Kap. 3 bzw. 4 werden wir aber auch das Erzeugen einzelner Objekte und das Klonen von Objekten kennen lernen¹.

Aus programmiersprachlicher Sicht ergeben sich die *Eigenschaften* und das *Verhalten* eines Objekts aus dessen möglichen Zuständen und daraus, wie es auf Nachrichten reagiert. Demzufolge muss die Beschreibung von Objekten – also insbesondere eine Klassendeklaration – festlegen,

- welche Zustände ein Objekt annehmen kann,
- auf welche Nachrichten es reagieren soll und kann
- und wie die Methoden aussehen, mit denen ein Objekt auf den Empfang von Nachrichten reagiert.

Die Menge der Zustände eines Objekts wird durch die Attribute beschrieben, die es besitzt. Ein Attribut ist eine objektlokale Variable (vgl. Abschn. 1.2.3). Der aktuelle *Zustand* eines Objekts *obj* ist bestimmt durch die aktuellen Werte seiner Attribute. Die Menge der Zustände entspricht dann allen möglichen Wertkombinationen, die die Attribute von *obj* annehmen können. Üblicherweise reagiert ein Objekt genau auf die Nachrichten, für die es Methoden besitzt. Außer den Attributen müssen also nur noch die Methoden beschrieben werden. Der folgende Abschnitt erläutert, wie Attribute und Methoden in Java deklariert werden, wie Objekte erzeugt werden und wie auf Attribute zugegriffen werden kann.

2.1.2 Klassen beschreiben Objekte

Eine *Klasse* beschreibt in Java einen neuen Typ von Objekten und liefert eine Implementierung für diese Objekte. Der Rumpf der Klasse besteht normalerweise aus einer Liste von Attribut-, Methoden- und Konstruktordeklarationen. Eine Attributdeklaration sieht genauso aus wie eine Variablendeklaration (vgl. Abschn. 1.3.1.2, S. 25); sie legt den Typ und den Namen des Attributs fest. Eine Methodendeklaration legt den Namen der Methode fest und beschreibt deren formale Parameter und mögliche Ergebnisse. Konstruktoren werden in Java benutzt, um Objekte zu initialisieren. Konstruktordeklarationen ähneln Methodendeklarationen, besitzen aber keinen Rückgabotyp und haben immer den Namen der umfassenden Klasse. Wir betrachten zunächst ein Beispiel und erläutern dann die Sprachkonstrukte im Einzelnen.

Als Beispiel betrachten wir eine Klasse, die Textseiten mit Titel und Inhalt beschreibt und uns als Ausgangspunkt für eine Browseranwendung dienen wird (vgl. Abb. 2.5, S. 63). Da wir dieses Beispiel im Folgenden ausbauen werden, um in einige Aspekte des World Wide Web, kurz WWW, einzuführen, nennen wir die Klasse `W3Seite`:

¹ Das Verändern der Eigenschaften von Objekten zur Ausführungszeit ist in Java nicht möglich.

```
(1)  class W3Seite {
(2)      String titel;                // Attribut
(3)      String inhalt;              // Attribut

(4)      W3Seite ( String t, String i ) {    // Konstruktor
(5)          titel = t;
(6)          this.inhalt = i;
(7)      }
(8)      String getTitel() {                // Methode
(9)          return this.titel;
(10)     }
(11)     String getInhalt() {                // Methode
(12)         return inhalt;
(13)     }
(14) }
```

Die Klasse `W3Seite` deklariert die beiden Attribute `titel` und `inhalt`, beide vom Typ `String`, einen Konstruktor mit zwei Parametern zum Erzeugen und Initialisieren von Objekten der Klasse `W3Seite` sowie die beiden Methoden `getTitel` und `getInhalt` zum Auslesen der Attributwerte (auf die unterschiedliche Syntax für den Attributzugriff in den Zeilen (5) und (6) bzw. (9) und (12) gehen wir später ein). Eine triviale Anwendung der Klasse demonstrieren wir mit folgender Klasse `W3SeitenTest`, die die Klasse `W3Seite` zu einem ausführbaren Programm ergänzt²:

```
(15) class W3SeitenTest {
(16)     public static void main( String[] argv ) {
(17)         W3Seite meineSeite;
(18)         meineSeite =
(19)             new W3Seite( "Abstraktion",
(20)                 "Abstraktion bedeutet ..." );
(21)         (System.out).println( meineSeite.getInhalt() );

(22)         W3Seite meinAlias;
(23)         meinAlias = meineSeite;
(24)         meinAlias.inhalt = "Keine Angabe";
(25)         System.out.println( meineSeite.getInhalt() );
(26)     }
(27) }
```

In Zeile (17) wird die *lokale* Variable `meineSeite` vom Typ `W3Seite` deklariert. In Zeile (18)-(20) wird ihr (die Referenz auf) ein neues Objekt vom Typ `W3Seite` zugewiesen. Der Konstruktor initialisiert dabei das Attribut `titel` zu "Abstraktion" und das Attribut `inhalt` zu "Abstraktion bedeutet ...". In Zeile (21) wird für das Objekt, das von der Variablen `meineSeite` referenziert wird, die Methode `getInhalt` aufgerufen. Sie liefert die Zeichenreihe "Abstraktion bedeutet ..." als aktuellen Parameter für die Methode `println`.

² Online im Verzeichnis `kap2/w3Seite` verfügbar.

Die Zeilen (1)-(21) enthalten alle wesentlichen syntaktischen Konstrukte für die objektbasierte Programmierung in Java. Die letzten vier Zeilen sollen auf einen wichtigen Aspekt in der objektbasierten Programmierung hinweisen und den Unterschied zwischen Variablen, Objekten und Referenzen auf Objekte nochmals verdeutlichen. In Zeile (22) wird die lokale Variable `meinAlias` deklariert; in Zeile (23) wird ihr (die Referenz auf) dasselbe Objekt zugewiesen, das auch von der Variablen `meineSeite` referenziert wird. Damit existieren zwei Variablen, die dasselbe Objekt referenzieren. Wird ein Objekt von zwei oder mehr Variablen referenziert, spricht man häufig von *Aliasing* und nennt Variablen wie `meinAlias` auch einen *Alias* für das schon referenzierte Objekt. In Zeile (24) wird dem Attribut `inhalt` des von `meinAlias` referenzierten Objekts die Zeichenreihe "Keine Angabe" zugewiesen. Die daraus resultierende Variablenbelegung ist in Abb. 2.1 demonstriert (eine Erklärung der Repräsentation von String-Objekten verschieben wir bis Abschn. 2.1.4). In Zeile (25) wird demzufolge die Zeichenreihe "Keine Angabe" ausgegeben.

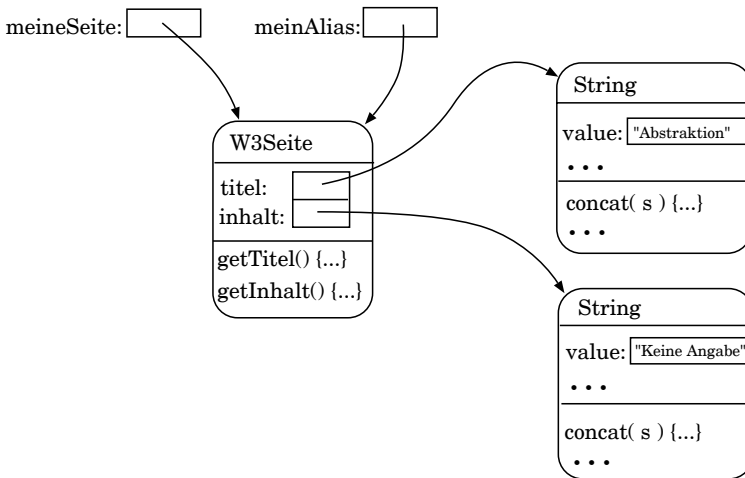


Abb. 2.1. Variablen- und Attributbelegung zum Beispiel

Die folgenden Paragraphen liefern eine genauere und allgemeinere Erläuterung der vom obigen Beispiel illustrierten Sprachkonstrukte.

Klassendeklarationen, Attribute und Objekte. Betrachtet man kein Subtyping und keine Vererbung, besteht eine Klassendeklaration in Java aus einer Modifikatorenliste, dem Schlüsselwort `class`, dem Klassennamen und einer in geschweifte Klammern eingeschlossenen Liste von Attribut-, Konstruktor- und Methodendeklarationen:

```

Modifikatorenliste class Klassenname {
    Liste von Attribut-, Konstruktor- und Methodendeklarationen
}

```

Die Modifikatorenliste ist eine Liste von Schlüsselwörtern, mit denen bestimmte Eigenschaften der Klasse beschrieben werden können; welche Eigenschaften das sind, werden wir im Laufe des Buches kennen lernen. Sie kann wie in den obigen Beispielen leer sein. Der Klassenname wird gleichzeitig als Typname für die Objekte dieser Klasse verwendet³. Typen, die durch eine Klasse deklariert sind, nennen wir *Klassentypen*. Die Attribute, Konstrukturen und Methoden einer Klasse nennen wir zusammenfassend die *Komponenten* der Klasse.

Ein Objekt einer Klasse besitzt für jedes Attribut der Klasse eine *objektlokale Variable*. Der Einfachheit halber werden wir diese objektlokalen Variablen ebenfalls als *Attribute* bezeichnen. In der objektorientierten Literatur wird allerdings zum Teil in der Begriffswahl zwischen der Klassenebene und der Objektebene unterschieden. Die Objekte einer Klasse werden dann häufig die *Instanzen* der Klasse genannt, und man spricht statt vom Erzeugen eines Objekts vom *Instanzieren* einer Klasse. Dementsprechend bezeichnet man die objektlokalen Variablen vielfach auch als *Instanzvariablen*. (Bei dem Wort „Instanz“ sollte man nicht an behördliche Instanzen denken, sondern vielmehr an das Ausstanzen eines Gegenstands gemäß einer vorgegebenen Schablone: die Klasse entspricht der Schablone, der Gegenstand dem Objekt.⁴)

Methodendeklaration. In Java besteht eine Methodendeklaration aus der *erweiterten Methodensignatur* und dem *Methodenrumpf*. Der Methodenrumpf ist ein Anweisungsblock (zur Definition von Anweisungsblöcken siehe Unterabschn. 1.3.1.3); die erweiterte Methodensignatur hat im Allgemeinen folgende Form:

```

Modifikatorenliste Ergebnistyp Methodenname (Parameterliste) throws-Deklaration

```

Wir gehen die Teile der erweiterten Methodensignatur kurz durch:

- Wie bei Klassen, ist die Modifikatorenliste eine Liste von Schlüsselwörtern, mit denen bestimmte Eigenschaften der Methoden beschrieben werden können. Sie kann leer sein wie bei den Methoden `getTitel` und `getInhalt`. Ein Beispiel für eine nichtleere Modifikatorenliste bietet die Methodendeklaration von `main`, wie wir sie in unseren Testprogrammen verwenden (vgl. Zeile (16) auf S. 52): Sie enthält die Modifikatoren `public` und `static`, deren Bedeutung wir noch in diesem Kapitel behandeln werden.

³ In Kap. 3 werden wir sehen, dass der Typ, der von einer Klasse deklariert wird, im Allgemeinen nicht nur die Objekte dieser Klasse umfasst.

⁴ Um die angedeutete falsche Analogie zu vermeiden, wird in [Goo99] der Terminus „Ausprägung“ statt „Instanz“ verwendet.

- Als Ergebnistyp ist entweder ein Typname oder, wenn die Methode kein Ergebnis zurückliefert, das Schlüsselwort `void` zulässig. Die Methoden aus obigem Beispiel liefern also (Referenzen auf) String-Objekte zurück; die Methode `main` aus dem minimalen Programmrahmen liefert kein Ergebnis.
- Nach dem Methodennamen folgt die in Klammern eingeschlossene und durch Komma getrennte Liste von Parameterdeklarationen. Eine Parameterdeklaration besteht aus dem Typ und dem Namen des Parameters; ein Beispiel zeigt die Deklaration von `main` in Zeile (16), die einen Parameter `argv` vom Typ `String[]` deklariert. Die Methodendeklarationen `getTitel` und `getInhalt` illustrieren, dass Parameterlisten auch leer sein können.
- In der *throws-Deklaration* muss deklariert werden, welche Ausnahmen, die in der Methode selbst *nicht* behandelt werden, bei ihrer Ausführung auftreten können (vgl. die Erläuterungen zur `try`-Anweisung auf den Seiten 32ff); können keine Ausnahmen auftreten, kann die *throws-Deklaration* entfallen (wie bei den obigen Methodendeklarationen). So wie der Ergebnistyp darüber Auskunft gibt, welche Ergebnisse die Methode bei normaler Terminierung liefert, gibt die *throws-Deklaration* darüber Auskunft, welche Ausnahmen bei abrupter Terminierung der Methode möglicherweise auftreten. Details zur *throws-Deklaration* werden in Kap. 4 behandelt.

Im Rumpf einer Methode *m* sind die Parameter von *m* sowie alle Attribute und Methoden der umfassenden Klasse sichtbar, also auch diejenigen, die erst weiter unten im Text der Klassendeklaration stehen. Das Objekt, für das die Methode aufgerufen wurde, wird in Methodenrümpfen mit `this` bezeichnet (vgl. Zeile (9) des Beispiels und Abschn. 1.3.2.2); häufig nennt man dieses Objekt auch einfach das *this-Objekt*, das *self-Objekt* oder den *impliziten Parameter* der Methode⁵.

Konstruktordeklaration. Eine Konstruktordeklaration besteht in Java aus der *erweiterten Konstruktorsignatur* und dem *Konstruktorrumpf*. Die erweiterte Konstruktorsignatur hat im Allgemeinen folgende Form:

Modifikatorenliste *Klassenname* (*Parameterliste*) *throws-Deklaration*

Der Klassenname muss gleich dem Namen der umfassenden Klasse sein. Er wird häufig auch als Name des Konstruktors verwendet. Der Konstruktorrumpf ist ein Anweisungsblock (zur Definition von Anweisungsblöcken siehe Unterabschn. 1.3.1.3). Er darf `return`-Anweisungen ohne Ergebnis und als erste Anweisung eine spezielle Form von Konstruktoraufrufen enthalten (darauf werden wir in Abschn. 3.3 näher eingehen). Im Rumpf eines Konstruktors sind seine Parameter sowie alle Attribute und Methoden der umfassenden Klasse sichtbar.

⁵ Beispielsweise wird es in Java und C++ als *this-Objekt* bezeichnet, in Smalltalk als *self-Objekt*; die Bezeichnung *impliziter Parameter* rührt daher, dass der Parameter in der Methodendeklaration nicht explizit erwähnt wird.

Konstruktoren dienen zum Initialisieren von Objekten (vgl. die Deklaration des Konstruktors von `W3Seite`, Zeilen (4)–(7), und dessen Aufruf in den Zeilen (19)–(20)). In Konstruktorrümpfen bezeichnet `this` das zu initialisierende Objekt. Enthält eine Klassendeklaration keinen Konstruktor, stellt Java automatisch einen Konstruktor mit leerer Parameterliste und leerem Rumpf bereit; dieser implizite Konstruktor wird *default-Konstruktor* genannt. Weitere wichtige Eigenschaften von Konstruktoren werden in Abschn. 3.3 behandelt.

Objekterzeugung, Attributzugriff und Methodenaufruf. Die obigen beiden Absätze haben die Deklaration von Klassen, Methoden und Konstruktoren behandelt. Dieser Absatz erläutert, wie deren Anwendung in Programmen beschrieben wird; d.h. wie Objekte erzeugt werden, wie auf deren Attribute zugegriffen wird und wie man deren Methoden aufruft. Dazu erweitern wir die in Abschn. 1.3.1.2 vorgestellte Ausdruckssyntax um drei zusätzliche Ausdrucksformen: die Objekterzeugung, den Attributzugriff und den Methodenaufruf.

Syntaktisch ist eine *Objekterzeugung* ein Ausdruck, der mit dem Schlüsselwort `new` eingeleitet wird und folgenden Aufbau hat, wobei die aktuelle Parameterliste eine Liste von Ausdrücken ist:

```
new Klassenname ( AktuelleParameterliste )
```

Die Objekterzeugung wird wie folgt ausgewertet: Zuerst wird eine neue Instanz der Klasse *Klassenname* erzeugt. Die Attribute dieses neu erzeugten Objekts sind mit default-Werten initialisiert (der default-Wert für ein Attribut von einem Klassentyp ist die Referenz `null`, der default-Wert von zahlenwertigen Attributen ist 0, der für boolesche Attribute `false`). Danach werden die Ausdrücke der Parameterliste ausgewertet. Anschließend wird der Konstruktor *Klassenname* aufgerufen. Bei diesem *Konstruktoraufruf* wird das neu erzeugte Objekt als impliziter Parameter übergeben. Terminiert die Auswertung einer Objekterzeugung normal, liefert sie die Referenz auf ein neu erzeugtes Objekt der Klasse *Klassenname* als Ergebnis. Abrupte Terminierung tritt auf, wenn nicht mehr genügend Speicher für die Allokation des Objekts in der Laufzeitumgebung zur Verfügung steht oder wenn die Auswertung der Parameterausdrücke oder des Konstruktoraufrufs abrupt terminiert. Die Zeilen (19) und (20) des obigen Beispiels demonstrieren eine Objekterzeugung.

Ein *Attributzugriff* hat grundsätzlich die Form *Ausdruck* . *Attributname*. Der Typ des Ausdrucks muss ein Klassentyp sein und die zugehörige Klasse muss ein Attribut des angegebenen Namens besitzen. Liefert die Auswertung des Ausdrucks den Wert `null`, terminiert der Attributzugriff abrupt und erzeugt eine `NullPointerException`. Im Normalfall liefert der Ausdruck ein Objekt *X* vom angesprochenen Klassentyp. Steht der Attributzugriff auf der linken Seite einer Zuweisung, wird das Attribut *Attributname* vom Objekt *X* durch die Zuweisung entsprechend modifiziert (schreibender Zugriff). An-

dernfalls wird der Wert des Attributs gelesen. Die Zeilen (6), (9) und (24) des Beispiels zeigen Attributzugriffe in Standardform. Der Typ der Variablen `meinAlias` und des Vorkommens von `this` in Zeile (9) ist `W3Seite`. In der Zeile (9) wird der Wert des Attributs gelesen, die Zeilen (6) und (24) illustrieren einen schreibenden Zugriff. Als abkürzende Schreibweise gestattet es Java statt `this.Attributname` einfach nur den Attributnamen zu schreiben. Die Zeilen (5) und (12) des Beispiels sind also gleichbedeutend mit `this.titel = t;` bzw. `return this.inhalt;`. Zum präzisen Verständnis der Bedeutung eines Programmes ist es wichtig, sich dieser abkürzenden Schreibweise immer bewusst zu sein. Insbesondere kann der Zugriff auf eine lokale Variable syntaktisch leicht mit einem Attributzugriff verwechselt werden.

Ein *Methodenaufruf* ähnelt syntaktisch einem Attributzugriff:

Ausdruck.*Methodenname* (*AktuelleParameterliste*)

Der Ausdruck beim Methodenaufruf beschreibt (die Referenz auf) das Objekt, für das die Methode aufgerufen werden soll, das sogenannte *Empfängerobjekt*. Der Ausdruck kann selbstverständlich zusammengesetzt und geklammert sein (beispielsweise haben wir in Zeile (21) den Ausdruck `System.out` zur Verdeutlichung geklammert). Der Typ des Ausdrucks muss eine Klasse bezeichnen, die eine Methode mit dem angegebenen Methodenname besitzt. Die aktuelle Parameterliste ist eine Liste von Ausdrücken. Die Auswertung eines Methodenaufrufs terminiert abrupt, wenn die Auswertung des Ausdrucks den Wert `null` ergibt, wenn die Auswertung eines der aktuellen Parameter abrupt terminiert oder wenn die Ausführung des Methodenrumpfes abrupt terminiert. Entsprechend der Abkürzungsregel beim Attributzugriff kann beim Methodenaufruf die Angabe von `this.` entfallen. Zeile (21) zeigt zwei Methodenaufrufe, wobei `System.out` das Objekt für die Standardausgabe beschreibt.

Objektorientierte Programme. In einer Sprache wie Java, die Klassen zur Beschreibung von Objekten verwendet, ist ein objektorientiertes Programm im Wesentlichen eine Ansammlung von Klassen. Ein Programm wird gestartet, indem man eine Methode einer der Klassen aufruft. Ein kleines Problem dabei ist, dass man nach dem bisher Gesagten ein Empfängerobjekt benötigt, um eine Methode aufzurufen, ein solches Objekt beim Programmstart aber noch nicht existiert. Java löst dieses Problem, indem es die Deklaration sogenannter statischer Methoden oder Klassenmethoden unterstützt. Das sind Methoden ohne impliziten Parameter, die man analog zu Prozeduren prozeduraler Sprachen direkt aufrufen kann. Klassenmethoden werden in Unterabschn. 2.1.4.2 behandelt. Für das Verständnis des Folgenden reicht es zu wissen, dass es solche Methoden gibt und dass sie durch Voranstellen des Schlüsselwortes `static` deklariert werden.

In Java gibt es keine als Hauptprogramm ausgezeichneten Programmteile wie etwa in Pascal. Für den Programmstart kann jede Klasse benutzt werden,

die eine Klassenmethoden mit Namen `main` und einem Parameter vom Typ `Feld` von Zeichenreihen besitzt. Wir betrachten dazu folgendes Beispiel mit den Klassen `Test1` und `Test2`:

```
class Test1 {
    public static void main( String[] argf ){
        Test2 obj = new Test2();
        obj.drucke("Ich bin ein Objekt vom Typ Test2");
    }
    public void hallo(){
        System.out.println("Jemand hat mich gegr\u00FC\u00DFt");
    }
}

class Test2 {
    public static void main( String[] argf ){
        (new Test1()).hallo();
    }
    public void drucke( String s ) {
        System.out.println( s );
    }
}
```

Befindet sich der Programmtext der Klassen in den Dateien `Test1.java` bzw. `Test2.java` (online in `kap2/programmStart` verfügbar), können die Klassen mit `javac Test1.java` bzw. `javac Test2.java` übersetzt werden. Mit jeder der beiden Klassen kann dann eine Programmausführung gestartet werden. Beispielsweise führt das Shell-Kommando `java Test1` zum Aufruf der Methode `main` von Klasse `Test1`. (Als Parameter wird dabei ein Feld ohne Einträge, d.h. mit Länge 0 übergeben. Will man beim Programmstart Parameter mitgeben, muss man sie im Shell-Kommando hinter dem Klassennamen angeben.) Im Rumpf von `Test1.main` wird eine Instanz der Klasse `Test2` erzeugt und der lokalen Variablen `obj` zugewiesen (beachte: Da Klasse `Test1` keinen Konstruktor deklariert, kommt der default-Konstruktor zum Einsatz). Für das neu erzeugte Objekt wird dann die Methode `drucke` der Klasse `Test2` aufgerufen. In entsprechender Weise kann man mit der Klasse `Test2` die Programmausführung beginnen.

Grundsätzlich besteht ein Java-Programm aus einer Ansammlung von Klassen. Wie im Laufe des Buches noch deutlich werden wird, ist es allerdings nicht immer leicht, präzise zu sagen, welche Klassen zu einem Programm gehören. Dies gilt insbesondere, wenn ein Programm sich erst zur Ausführungszeit entscheidet, welche Klassen es noch dynamisch dazu laden möchte (siehe Abschn. 2.1.7), oder wenn ein Programm auf mehrere Rechner verteilt ist (siehe Kap. 7). Darüber hinaus führt der intensive Gebrauch von Bibliotheksklassen in der objektorientierten Programmierung dazu, den statisch angelegten Programmbegriff aufzuweichen. Vielfach erscheint eine Sichtweise angemessener, in der Klassen als Komponenten von offenen Sys-

temen betrachtet werden, d.h. von Systemen, bei denen die teilnehmenden Komponenten nicht von vornherein festgelegt sind. Jede Klasse sorgt für die Erzeugung ihrer eigenen Objekte und stellt den Programmcode für die zugehörigen Methoden bereit. Sie benötigt andere Klassen, damit sie arbeiten kann. Welche Klassen das sind, lässt sich zur Übersetzungszeit bestimmen. Aber im Allgemeinen steht weder zur Übersetzungszeit, noch beim Systemstart fest, welche Klassen darüber hinaus oder stellvertretend während der Laufzeit des Systems zum Einsatz kommen.

Selbst in dem einfachen, oben angegebenen Beispiel ist nicht ohne weiteres zu erkennen, welche Klassen man zum Programm rechnen soll. Zwar ist klar, dass keine der beiden Klassen für sich genommen ein vollständiges Programm sein kann, da jede Klasse die jeweils andere benutzt: Sie erzeugt ein Objekt der anderen Klasse und ruft darauf eine Methode auf. Aber zu dem Programm gehören auch Klassen der Java-Bibliothek, z.B. die Klasse `String` und `System`, und damit alle transitiv von diesen verwendeten Klassen (beispielsweise die Klasse der Objekte, die von `System.out` referenziert werden; siehe Unterabschn. 2.1.4.2, S. 70). Da es insbesondere im Zusammenhang mit Subtyping und Vererbung nicht einfach ist, genau zu definieren, wann eine Klasse von einer anderen verwendet wird, begnügen wir uns im Folgenden mit der groben Festlegung, dass ein Java-Programm aus den benutzergeschriebenen und den Bibliotheksklassen besteht.

Zusammenfassung. Objekte lassen sich nach den Operationen, die auf ihnen definiert sind, und den Zuständen, die sie einnehmen können, klassifizieren; d.h. alle Objekte mit den gleichen Operationen und der gleichen Zustandsmenge werden in einer Klasse zusammengefasst. Andersherum besehen, legt eine Klasse fest, welche Operationen auf ihren Objekten möglich und zulässig sind und welche Zustände sie einnehmen können. Diese Sichtweise wird von den meisten objektorientierten Programmiersprachen übernommen, d.h. der Programmierer beschreibt nicht die Eigenschaften einzelner Objekte, sondern deklariert Klassen, die die Eigenschaften ihrer Objekte beschreiben, und kann dann Objekte als Instanzen zu den deklarierten Klassen erzeugen.

2.1.3 Benutzen und Entwerfen von Klassen

Klassen bilden das zentrale Sprachkonstrukt von Java. Dementsprechend stehen beim Programmentwurf zwei Fragen im Mittelpunkt:

1. Welche existierenden Klassen können für den Programmentwurf herangezogen werden?
2. Welche Klassen müssen neu entworfen werden?

Insbesondere die erste Fragestellung gewinnt in der auf Wiederverwendung ausgerichteten objektorientierten Programmierung zunehmend an Bedeutung. Im Laufe dieses Buches werden wir diesem Aspekt beim Kennenlernen der Java-Bibliothek immer wieder begegnen. Das Zusammenspiel von Benutzung existierender Klassen und Entwurf neuer Klassen soll in diesem Abschnitt an einem kleinen Beispiel betrachtet werden; dabei soll auch gezeigt werden, wie die im letzten Abschnitt vorgestellten Konstrukte im Rahmen einer Entwurfsaufgabe anzuwenden sind.

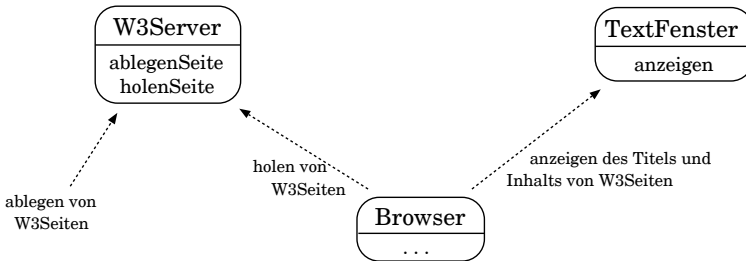


Abb. 2.2. Zusammenarbeit von Server, Textfenster und Browser

Aufgabenstellung. Ein rudimentäres Browser-Programm soll realisiert werden, mit dem Objekte vom Typ `W3Seite` bei einem Server geholt und in einem Fenster angezeigt werden können (vgl. Abb. 2.2). Der Seiten-Server ist ein Objekt, bei dem man W3-Seiten unter einer Adresse ablegen und danach unter dieser Adresse wieder holen kann. Die Adresse ist eine beliebige Zeichenreihe. Hier ist die Klassenschnittstelle des Servers:

```

class W3Server {
    W3Server() { ... }
    void ablegenSeite( W3Seite s, String sadr ) { ... }
    W3Seite holenSeite( String sadr ) { ... }
}
  
```

Zum Anzeigen von Seiten soll die Klasse `TextFenster` benutzt werden:

```

class TextFenster ... {
    ...
    TextFenster() { ... }
    void anzeigen( String kopfzeile, String text ) { ... }
}
  
```

Der Konstruktor erzeugt ein `TextFenster`-Objekt, das beim ersten Aufruf der Methode `anzeigen` mit der angegebenen Kopfzeile und dem angegebenen Text auf dem Bildschirm erscheint. Jeder weitere Aufruf von `anzeigen` auf ein `TextFenster`-Objekt führt dazu, dass die Kopfzeile und der Text entsprechend den Parametern geändert werden. Für die Darstellung wird der Text

an Wortzwischenräumen geeignet in Zeilen umbrochen. Das Textfenster wird automatisch aktualisiert, wenn es auf dem Bildschirm wieder in den Vordergrund kommt (beispielsweise durch Verschieben eines anderen Fensters, das das Textfenster verdeckt hat).

Entwerfen eines Browsers. Browser mit dem beschriebenen Verhalten sollen durch eine Klasse `Browser` implementiert werden (vgl. Abb. 2.3). Bei Erzeugung eines Browser-Objekts wird eine Referenz auf den zuständigen Server übergeben. Jedes Browser-Objekt besitzt ein Textfenster zum Anzeigen der aktuellen Seite. Nachdem das Textfenster erzeugt ist, lädt der Browser eine Startseite und startet die interaktive Steuerung des Browsers. Die Steuerung

```
class Browser {
    W3Server    meinServer;
    TextFenster oberfl;
    W3Seite     aktSeite; // aktuelle Seite

    Browser( W3Server server ){
        meinServer = server;
        oberfl      = new TextFenster();
        laden( new W3Seite("Startseite",
                           "NetzSurfer: Keiner ist kleiner") );
        interaktiveSteuerung();
    }
    void laden( W3Seite s ){
        aktSeite = s;
        oberfl.anzeigen( aktSeite.getTitel(),
                           aktSeite.getInhalt());
    }
    void interaktiveSteuerung() { ... }
}
```

Abb. 2.3. Implementierung eines rudimentären Browsers

erlaubt es dem Benutzer, interaktiv neue Seiten zu laden, indem er deren Adresse eingibt, oder den Browser zu beenden. Eine Implementierung der interaktiven Steuerung behandeln wir in Abschn. 2.1.4 zur Illustration weiterer sprachlicher Konstrukte von Java.

Abbildung 2.4 zeigt einen Testrahmen für die beschriebenen Klassen. Die vier Beispielseiten entsprechen den Anfängen der Kapitel 2 und 3 und der Abschnitte 2.1 und 2.2. Der Operator „+“ bezeichnet dabei die *Konkatenation von Zeichenreihen*. Die Zeichenreihe "
" wird zur Markierung von Zeilenumbrüchen verwendet (die Syntax mit den spitzen Klammern ist der Seitenbeschreibungs-Sprache *HTML – Hypertext Markup Language* – entlehnt; BR steht für **B**reak **R**ow). Abbildung 2.5 zeigt die Seite `kap2` dargestellt in einem Textfenster. In der bisher behandelten Ausbaustufe des Browsers muss

```

class Main {
    public static void main( String[] argf ){
        W3Seite kap2 = new W3Seite(
            "Objekte, Klassen, Kapselung",
            "Dieses Kapitel erl\u00E4utert, wie Objekte "
            +" beschrieben werden k\u00F6nnen, und geht dazu "
            +" ausf\u00Fchrlich ... deutlich zu machen, "
            +" werden Subtyping und Vererbung erst im n\u00E4chsten"
            +" Kapitel behandelt (siehe W3-Seite kap3). <BR> "
            +"Dieses Kapitel besteht aus zwei Abschnitten. Im "
            +" Mittelpunkt des ersten Abschnitts ... in Pakete /"
            +" Module ein. <BR> <BR> "
            +"Abschnitte: <BR> "
            +"2.1 Objekte und Klassen (siehe W3-Seite abs2.1) <BR> "
            +"2.2 Kapselung und Strukturierung von Klassen (siehe "
            +" W3-Seite abs2.2) "
        );
        W3Seite kap3 = new W3Seite(
            "Vererbung und Subtyping",
            "Dieses Kapitel ... "
        );
        W3Seite abs21 = new W3Seite(
            "Objekte und Klassen",
            "Dieser Abschnitt behandelt die Frage: Was ... "
        );
        W3Seite abs22 = new W3Seite(
            "Kapselung und Strukturierung von Klassen",
            "Der vorangegangene Abschnitt ... "
        );

        W3Server testServer = new W3Server();
        testServer.ablegenSeite(kap2, "kap2");
        testServer.ablegenSeite(kap3, "kap3");
        testServer.ablegenSeite(abs21, "abs2.1");
        testServer.ablegenSeite(abs22, "abs2.2");

        new Browser( testServer );
    }
}

```

Abb. 2.4. Programmrahmen für die Browser-Klasse

der Benutzer die Referenzen auf andere Seiten innerhalb des Texts (im Beispiel sind das kap3, abs2.1 und abs2.2) noch von Hand in den Browser eingeben, um diese Seiten zu laden. (Java-Klassen zum Testen des rudimentären Browsers sind online im Verzeichnis kap2/browseV1 verfügbar.)

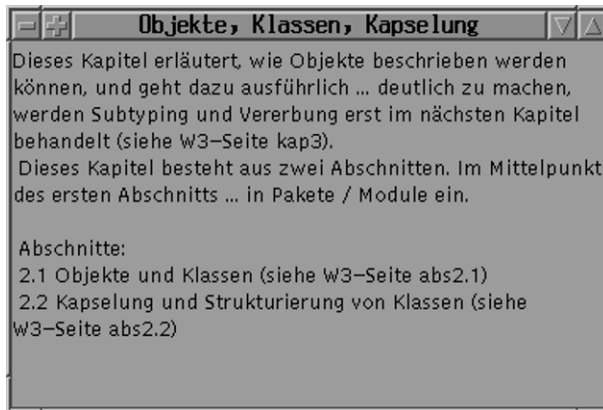


Abb. 2.5. Textfenster mit Seite „kap2“

2.1.4 Spracherweiterungen

Dieser Abschnitt behandelt Erweiterungen des in 2.1.2 vorgestellten Java-Sprachkerns. Diese Erweiterungen bringen konzeptionell zwar wenig Neues, sind aber für die effektive Programmierung unverzichtbar. Im Einzelnen stellen wir das Initialisieren von Attributen und Variablen, die Feldtypen von Java, das Überladen von Methodennamen sowie Klassenmethoden und Klassenattribute vor. Begleitend dazu werden wir erste Einblicke in die vordefinierten Java-Klassen `String` und `System` geben und das Browserbeispiel weiter vorantreiben.

2.1.4.1 Initialisierung, Felder und Überladen

Dieser Unterabschnitt zeigt zunächst, wie Variablen und Attribute bei der Deklaration initialisiert werden können, stellt dann die Realisierung von Feldern in Java vor und erläutert schließlich das Überladen von Methodennamen.

Initialisieren von Variablen und Attributen. Lokale Variablen und Attribute können an ihrer Deklarationstelle initialisiert werden. Beispielsweise ist das Programmfragment

```
class C {
    int ax, ay;
    C(){
        ax = 7;  ay = 9;    m();
    }
    void m(){
        int v1, v2;
        v1 = 4848;  v2 = -3;  ...
    }
}
```

äquivalent zu der folgenden Klassendeklaration:

```
class C {
    int ax = 7, ay = 9;
    C(){
        m();
    }
    void m(){
        int v1 = 4848, v2 = -3; ...
    }
}
```

Insbesondere ist aus dem Beispiel zu erkennen, dass die Initialisierung von Attributen vor der Ausführung der entsprechenden Konstruktorrümpfe vorgenommen wird.

Java bietet die Möglichkeit Variablen und Attribute durch Voransetzen des Schlüsselworts `final` als *unveränderlich* (engl. `final`) zu deklarieren. Oft bezeichnet man solche Variablen bzw. Attribute auch als *benannte Konstanten*. Unveränderliche Variablen müssen an der Deklarationsstelle initialisiert werden, unveränderliche Attribute entweder an der Deklarationsstelle oder in den Konstruktoren:

```
class C {
    final int ax, ay = 9;
    C(){
        ax = 7; // KORREKT: Initialisierung im Konstruktor
                // und nicht an Deklarationsstelle
        m();
    }
    void m(){
        final int v1 = 4848, v2 = -3;
        v2 = 0; // UNZULAESSIG: v2 ist unveraenderlich
        ay = 34; // UNZULAESSIG: ay ist unveraenderlich
        ...
    }
}
```

Wenn Variablen und Attribute während der Programmausführung nicht verändert werden müssen, sollten sie auch als unveränderlich deklariert werden. Das vermeidet Programmierfehler und gestattet dem Übersetzer im Allgemeinen, effizienteren Code zu erzeugen.

Felder. *Felder* (engl. *arrays*⁶) sind in Java Objekte; dementsprechend werden sie dynamisch, d.h. zur Laufzeit, erzeugt und ihre Referenzen können an Variablen zugewiesen und als Parameter an Methoden übergeben werden. Ist *T*

⁶ Zur Beachtung: „Arrays“ sind nicht mit „fields“ zu verwechseln. In der engl. Java-Literatur wird das Wort „field“ für die in einer Klasse deklarierten Attribute verwendet.

ein beliebiger Typ in Java, dann bezeichnet $T[]$ den Typ der Felder mit *Komponententyp* T . Den Operator $[]$ nennt man einen *Typkonstruktor*, da er aus einem beliebigen Typ einen neuen Typ konstruiert.

Jedes Feld(-Objekt) besitzt ein unveränderliches Attribut `length` vom Typ `int` und eine bestimmte Anzahl von Attributen vom Komponententyp. Die Attribute vom Komponententyp nennen wir im Folgenden die *Feldelemente*. Die Anzahl der Feldelemente wird bei der Erzeugung des Feldes festgelegt und im Attribut `length` gespeichert. Felder sind in Java also immer eindimensional. Allerdings können die Feldelemente andere Felder referenzieren, sodass mehrdimensionale Felder als Felder von Feldern realisiert werden können.

Das Programmfragment von Abb. 2.6 zeigt alle im Zusammenhang mit Feldern relevanten Operationen. In Zeile (1) wird die Variable `vor` für (Referenzen auf) Felder mit Komponententyp `char` deklariert und mit einem Feld der Länge 3 initialisiert. (Um syntaktische Kompatibilität zur Sprache C zu erreichen, darf der Typkonstruktor `[]` auch erst nach dem deklarierten Namen geschrieben werden, also `char vor[]` statt `char[] vor`.) In den Zeilen (2)-(4) werden die drei Feldelemente initialisiert; man beachte, dass die Feldelemente von 0 bis `length - 1` indiziert werden. In Zeile (5) wird die Variable `Fliegen` mit einem Feld der Länge 7 initialisiert; die Länge berechnet der Übersetzer dabei aus der Länge der angegebenen Liste von Ausdrücken (in dem Beispiel ist jeder Ausdruck eine `char`-Konstante). In Zeile (6) wird eine Variable `satz` für Felder mit Komponententyp `char[]` deklariert; d.h. die Feldelemente können (Referenzen auf) `char`-Felder speichern. Die Variable `satz` wird mit einem vierelementigen Feld initialisiert,

- deren erstes und viertes Element mit dem in Zeile (5) erzeugten Feld initialisiert wird,
- deren zweites Element mit einem neu erzeugten Feld initialisiert wird
- und deren drittes Element mit dem von `vor` referenzierten Feld initialisiert wird.

Das resultierende *Objektgeflecht* ist in Abb. 2.7 dargestellt. Die Anweisungen in den Zeilen (12) bis (18) bauen die Worte zu einem Satz zusammen und geben diesen aus (`i++` ist gleichbedeutend zu `i=i+1`).

Felder als Objekte zu realisieren hat den Vorteil, dass sich Felder auf diese Weise besser in objektorientierte Klassenhierarchien einbetten lassen (Genaueres dazu in Kap. 3). Außerdem gewinnt man an Flexibilität, wenn man zwei- bzw. mehrdimensionale Felder realisieren möchte. In zweidimensionalen Feldern, wie man sie beispielsweise in Pascal deklarieren kann, müssen alle Spalten bzw. alle Zeilen die gleiche Länge haben; wie das Beispiel zeigt, gilt dies nicht bei der Java-Realisierung mittels Referenzen. Auch kann man mehrfach auftretende „Spalten“ bzw. „Zeilen“ durch ein mehrfach referenziertes Feld realisieren (vgl. Beispiel). Der Preis für die größere Flexibilität ist

```

(1) char[] vor = new char[3];
(2) vor[0] = 'v';
(3) vor[1] = 'o';
(4) vor[2] = 'r';
(5) char[] Fliegen = {'F','l','i','e','g','e','n'};
(6) char[][] satz = { Fliegen,
(7)                  {'f','l','i','e','g','e','n'},
(8)                  vor,
(9)                  Fliegen };
(10) int i,j;
(11) String s = "";
(12) for( i = 0; i < satz.length; i++ ) {
(13)     for( j = 0; j < satz[i].length; j++ ) {
(14)         s = s + satz[i][j];
(15)     }
(16)     if( i+1 < satz.length ) s = s + " ";
(17) }
(18) System.out.println( s + "." );

```

Abb. 2.6. Deklaration und Anwendung von Feldern

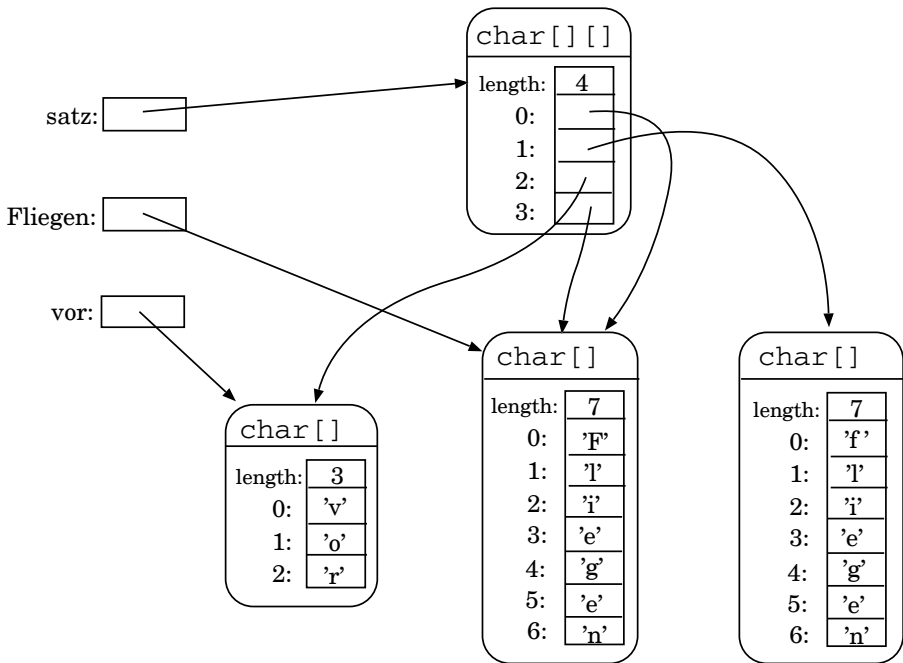


Abb. 2.7. Die Feldobjekte zum Beispiel

im Allgemeinen größerer Speicherbedarf, höhere Zugriffszeiten sowie eine etwas gestiegene Programmierkomplexität und damit Fehleranfälligkeit.

Überladen von Methodennamen und Konstruktoren. In Java ist es erlaubt, innerhalb einer Klasse mehrere Methoden mit dem gleichen Namen zu deklarieren. Eine derartige Mehrfachverwendung nennt man *Überladen* eines Namens. Methoden mit gleichen Namen müssen sich in der Anzahl oder in den Typen der Parameter unterscheiden. Dadurch ist es dem Übersetzer möglich, die Überladung *aufzulösen*, d.h. für jede Aufrufstelle zu ermitteln, welche von den Methoden gleichen Namens an der Aufrufstelle gemeint ist. Entsprechend dem Überladen von Methodennamen unterstützt Java auch das Überladen von Konstruktoren.

Gute Beispiele für das Überladen findet man in der Klasse `String` des Standard-Pakets `java.lang` der Java-Bibliothek. Ein `String`-Objekt besitzt in Java drei Attribute: eine Referenz auf ein `char`-Feld, den Index des Feld-elements mit dem ersten gültigen Zeichen und die Anzahl der Zeichen der repräsentierten Zeichenreihe. Die Klasse `String` hat elf Konstruktoren und 47 Methoden. Abbildung 2.8 präsentiert einen kleinen Auszug mit vier Konstruktoren und sieben Methoden.⁷

Der Konstruktor mit leerer Parameterliste liefert ein `String`-Objekt, das die leere Zeichenreihe repräsentiert. Der Konstruktor mit dem Parameter vom Typ `String` liefert ein neues `String`-Objekt, das die gleiche Zeichenreihe wie der Parameter repräsentiert. Der dritte Konstruktor erzeugt ein neues `String`-Objekt mit den im übergebenen Feld enthaltenen Zeichen (auf die Methode `arraycopy` gehen wir im folgenden Unterabschnitt kurz ein). Mit dem vierten Konstruktor kann ein neues `String`-Objekt zu einem Ausschnitt aus einem übergebenen Feld erzeugt werden; der Ausschnitt wird durch den `offset` des ersten Zeichens und die Anzahl der Zeichen (`count`) festgelegt.

Die Methoden mit Namen `indexOf` bieten vier Möglichkeiten, den Index des ersten Vorkommens eines Zeichens zu ermitteln. Die erste Version beginnt mit der Suche am Anfang der Zeichenreihe, die zweite beim angegebenen Index und die dritte mit Beginn des ersten Vorkommens der angegebenen Teilzeichenreihe `str`; die vierte Version verallgemeinert die dritte, indem sie mit der Suche der Teilzeichenreihe `str` erst ab Index `fromIndex` beginnt. Das Beispiel zeigt sehr schön die Anwendung von Überladen. Alle Methoden erfüllen eine ähnliche Aufgabe, sodass es gerechtfertigt ist, ihnen den gleichen Namen zu geben. Sie unterscheiden sich nur darin, wie allgemein sie sind (die zweite bzw. vierte Version sind allgemeiner als die erste bzw. dritte Version) und welche Parameter sie benötigen.

Die Methoden `length`, `charAt` und `equals` werden wir im Folgenden benötigen. Die Bedeutung von `length` und `charAt` ergibt sich aus dem angegebenen Programmcode. Die Methode `equals` dient im Wesentlichen

⁷ Gegenüber dem original Quellcode haben wir auch die Zugriffsmodifikatoren weggelassen (vgl. Abschn. 2.2).

```

class String {

    /** The value is used for character storage */
    char[] value;

    /** The offset is the first index of the used storage */
    int offset;

    /** The count is the number of characters in the String */
    int count;

    String() { value = new char[0]; }
    String( String value ) { ... }
    String( char[] value )
        this.count = value.length;
        this.value = new char[count];
        System.arraycopy(value, 0, this.value, 0, count);
    }
    String( char[] value, int offset, int count ) { ... }

    int indexOf(int ch) { return indexOf(ch, 0); }
    int indexOf(int ch, int fromIndex) { ... }
    int indexOf(String str) { return indexOf(str, 0); }
    int indexOf(String str, int fromIndex) { ... }

    int length() {
        return count;
    }
    char charAt(int index) {
        if ((index < 0) || (index >= count)) {
            throw new StringIndexOutOfBoundsException(index);
        }
        return value[index + offset];
    }
    boolean equals(Object anObject) { ... }
}

```

Abb. 2.8. Auszug aus der Klasse String

dazu, zwei String-Objekte daraufhin zu vergleichen, ob sie die gleiche Zeichenreihe repräsentieren. (Beachte: Zeichenreihengleichheit kann nicht mit dem Operator == geprüft werden, da dieser nur vergleicht, ob die String-Objekte identisch sind. Es kann aber unterschiedliche String-Objekte geben, die die gleiche Zeichenreihe repräsentieren.) Die Methode `equals` kann nicht nur String-Objekte als Parameter verkraften, sondern beliebige Objekte (jeder Klassentyp ist ein Subtyp von `Object`; dies wird in Kap. 3 erläutert). Ist der aktuelle Parameter kein String-Objekt, liefert die Methode `false`.

2.1.4.2 Klassenmethoden und Klassenattribute

Die letzte Erweiterung des bisher beschriebenen Java-Sprachkerns ist aus objektorientierter Sicht die interessanteste. Attribute und Methoden können in Java nicht nur Objekten zugeordnet werden, sondern auch Klassen. Was das im Einzelnen bedeutet, wie es deklariert wird und für welche Anwendungen dieses Konstrukt sinnvoll ist, soll im Folgenden kurz erläutert werden.

Deklaration und Bedeutung. *Klassenmethoden* bzw. *Klassenattribute* werden deklariert, indem man eine Methoden- bzw. Attributdeklaration mit dem Schlüsselwort `static` beginnt; dementsprechend werden sie häufig auch als *statische Methoden* bzw. *Attribute* bezeichnet. Diese Bezeichnung ist insofern berechtigt, als dass Klassenmethoden statisch, also zur Übersetzungszeit gebunden werden und nicht dynamisch, wie es die Regel bei normalen Methoden ist (zur dynamischen Bindung vgl. z.B. S. 42). Für Klassenattribute gilt, dass sie nicht dynamisch bei der Objekterzeugung alloziert werden; vielmehr wird statisch vom Übersetzer Speicherplatz für sie vorgesehen.

Wie wir gesehen haben, besitzt jedes Objekt für jedes normale Attribut seiner Klasse eine objektlokale Variable (vgl. die Bemerkungen zum Begriff „Instanzvariable“ auf S. 54). Im Gegensatz dazu entspricht jedem Klassenattribut genau eine Variable, unabhängig davon, wieviele Objekte der Klasse erzeugt werden; insbesondere existiert diese Variable auch, wenn gar kein Objekt der Klasse erzeugt wird. Für den Zugriff auf Klassenattribute wird die Syntax *Klassenname* . *Attributname* verwendet; innerhalb der Klassendefinition reicht die Angabe des Attributnamens.

Klassenmethoden ähneln Prozeduren der prozeduralen Programmierung. Sie besitzen keinen impliziten Parameter. Deshalb macht es auch keinen Sinn, innerhalb von Klassenmethoden auf die normalen Attribute einer Klasse zuzugreifen; denn erstens wäre nicht klar, zu welchem Objekt die Attribute gehören sollen, auf die der Zugriff erfolgt, und zweitens sollen Klassenmethoden auch ausführbar sein, wenn kein Objekt der Klasse existiert. Selbstverständlich darf eine Klassenmethode auf die zugehörigen Klassenattribute zugreifen. Ein typisches Beispiel für eine Klassenmethode ist die Methode mit Namen `main`, die zu jedem ausführbaren Programm gehört (vgl. S. 58).

Anwendungsbeispiele aus dem Standard-Paket. Klassenattribute und -methoden werden häufig verwendet, um Informationen über alle Objekte einer Klasse zentral bei der Klasse zu verwalten oder um prozedurale Aspekte in objektorientierten Programmen zu realisieren. Beispielsweise besitzt die Klasse `String` neun Klassenmethoden mit Namen `valueOf`, die u.a. zu den Werten der Basisdatentypen eine geeignete Zeichenreihenrepräsentation liefern; hier sind die Deklarationen von zweien dieser Methoden:

```
static String valueOf( long l ) { ... }  
static String valueOf( float f ) { ... }
```

Der Aufruf `String.valueOf( (float)(7./9.) )` liefert zum Beispiel die Zeichenreihe "0.7777778" und demonstriert, wie Klassenmethoden aufgerufen werden. Wiederum gilt, dass innerhalb der Klassendefinition die Angabe des Klassennamens beim Aufruf statischer Methoden entfallen kann.

Eine andere Anwendung von Klassenmethoden und -attributen ist die Realisierung von Modulen, wie sie in prozeduralen Sprachen üblich sind. Klassenattribute entsprechen dabei modullokalen Variablen, Klassenmethoden übernehmen die Rolle von modullokalen Prozeduren. Ein typisches Beispiel dafür ist die Klasse `System` der Java-Bibliothek. Sie beschreibt einen wichtigen Teil der Systemschnittstelle von Java-Programmen. Hier ist ein kurzer Ausschnitt ihrer Klassendeklaration:

```
class System {
    final static InputStream in  = ...;
    final static PrintStream out = ...;
    static void exit(int status) { ... }
    static native void arraycopy(Object src,int src_position,
                                Object dst,int dst_position, int length);
}
```

Die Klassenattribute `in` und `out` ermöglichen den Zugriff auf den Eingabe- bzw. Ausgabestrom eines Programmlaufs. Der Zugriff auf den Ausgabestrom wurde ja bereits mehrfach verwendet, indem wir für das Ausgabestrom-Objekt die Methode `print` aufgerufen haben:

```
System.out.print("Das klaert die Syntax des Printaufrufs");
```

Die Methode `exit` ermöglicht es, die Ausführung eines Java-Programms direkt abzubrechen (üblicher aktueller Parameter für `status` ist 0). Die Methode `arraycopy` ist eine System-Prozedur zum effizienten Kopieren von Feldern (sie ist uns schon bei der Implementierung eines der `String`-Konstruktoren begegnet, siehe S. 67). Sie kopiert die Elemente des Feldes `src` mit den Indizes `src_position` bis `src_position+length-1` in das Feld `dst` und zwar ab Index `dst_position`. Das Schlüsselwort `native` bei der Deklaration von `arraycopy` besagt, dass die Implementierung nicht in Java selbst erfolgt, sondern in einer anderen Programmiersprache (zur Zeit werden Programmierschnittstellen zu C und C++ unterstützt).

2.1.4.3 Zusammenwirken der Spracherweiterungen

Das Browserbeispiel soll uns hier dazu dienen, das Zusammenwirken der behandelten Spracherweiterungen in einem größeren Programmkontext zu studieren. Insbesondere werden wir sehen, wie statische Attribute und Methoden genutzt werden können, um Informationen, die allen Objekten einer Klasse gemeinsam sind, innerhalb der Klasse zu verwalten. Das spart nicht nur Speicherplatz, sondern ermöglicht auch eine Koordinierung des Verhaltens der Objekte.

Im Prinzip gestattet es die Browser-Klasse von S. 61, innerhalb einer Programmausführung mehrere Browser-Objekte zu starten. Insbesondere in realistischen Umgebungen macht dies auch Sinn, da man dann in einem Browserfenster eine Seite betrachten kann, während das andere Browser-Objekt eine neue Seite lädt, sodass man die oft viel zu langen Wartezeiten nutzen kann. Allerdings bringt die interaktive Steuerung über die Standard-Eingabe und -Ausgabe die Einschränkung mit sich, dass immer nur einer der aktiven Browser gesteuert werden kann. (In Kap. 5 werden wir zeigen, wie diese Einschränkung mittels interaktiven graphischen Bedienoberflächen überwunden werden kann.)

Die Browser-Klasse von Abschn. 2.1.3 ermöglicht zwar das Erzeugen mehrerer Browser-Objekte unterstützt es aber nicht durch interaktive Steuerung. Jedes Browser-Objekt besitzt seine eigene interaktive Steuerung. Dies führt dazu, dass jeweils nur der zuletzt gestartete Browser bedient werden kann – eine untragbare Einschränkung. Diese Einschränkung überwinden wir durch folgendes Redesign der Klasse `Browser`:

1. Wie in der Browser-Klasse von S. 61 bekommt jedes Browser-Objekt eine eigene Oberfläche und eine aktuelle Seite. Der zugehörige W3-Server wird allerdings in einem statischen Attribut gespeichert, da wir annehmen, dass er in allen Browsern einer Programmausführung gleich ist.
2. Um mit der interaktiven Steuerung zwischen den gestarteten Browsern wechseln zu können, verwalten wir die gestarteten Browser mit statischen Attributen in der Klasse `Browser`: Außer einem Feldattribut für die gestarteten Browser gibt es Klassenattribute für die maximal mögliche Anzahl gestarteter Browser, für den nächsten freien Index im Feld und für den Index des aktuellen Browsers, d.h. des Browsers, der aktuell gesteuert werden soll.
3. Die Referenz auf die Startseite speichern wir in einem statischen Attribut, damit sie für alle Browser genutzt werden kann.
4. Wir sehen zwei Konstruktoren vor: Der eine Konstruktor dient zum Erzeugen des ersten Browser-Objekts, der andere zum Erzeugen weiterer Browser. Den gemeinsamen Teil der Konstruktoren implementieren wir als Hilfsmethode namens `initialisieren`.
5. Die interaktive Steuerung realisieren wir als Klassenmethode.
6. Um die Anwendung zu vereinfachen, sehen wir eine statische Methode `start` vor, die den richtigen Konstruktor aufruft und dann die interaktive Steuerung startet⁸.

Abbildung 2.9 zeigt die so überarbeitete Browserklasse, die nochmals die in diesem Abschnitt eingeführten Spracherweiterungen illustriert: Initialisieren

⁸ Auf diese Weise vermeiden wir auch den schlechten Stil, in einem Konstruktor eine interaktiv arbeitende, nichtterminierende Methode aufzurufen.

```

class Browser {
    TextFenster oberfl;
    W3Seite      aktSeite;
    static W3Server meinServer;
    static final int MAX_ANZAHL = 4;
    static Browser[] gestarteteBrowser= new Browser[MAX_ANZAHL];
    static int      naechsterFreierIndex = 0;
    static int      aktBrowserIndex;
    static W3Seite  startseite =
        new W3Seite("Startseite","NetzSurfer: Keiner ist kleiner");

    // Konstruktor zum Starten des ersten Browsers
    Browser( W3Server server ) {
        if( naechsterFreierIndex != 0 ) {
            System.out.println("Es sind bereits Browser gestartet");
        } else {
            meinServer = server;
            initialisieren();
        }
    }
    // Konstruktor zum Starten weiterer Browser
    Browser() {
        if( naechsterFreierIndex == MAX_ANZAHL ) {
            System.out.println("Maximale Anzahl Browser erreicht");
        } else {
            initialisieren();
        }
    }

    static void start( W3Server server ) {
        new Browser(server);
        Browser.interaktiveSteuerung();
    }

    void initialisieren() {
        oberfl      = new TextFenster();
        gestarteteBrowser[ naechsterFreierIndex ] = this;
        aktBrowserIndex = naechsterFreierIndex;
        naechsterFreierIndex++;
        laden( startseite );
    }

    void laden( W3Seite s ){
        aktSeite = s;
        oberfl.anzeigen(aktSeite.getTitel(),aktSeite.getInhalt());
    }

    static void interaktiveSteuerung() {
        ... /* siehe folgende Abbildung */
    }
}

```

Abb. 2.9. Browser-Klasse nach Redesign

von Attributen, unveränderliche Attribute, Felder, Überladen von Konstruktoren sowie Klassenattribute und -methoden kombiniert mit normalen Attributen und Methoden. Die Implementierung der interaktiven Steuerung ist in Abb. 2.10 aufgeführt. Wir wollen sie uns im Folgenden etwas genauer anschauen, auch um bei der Behandlung graphischer Bedienoberflächen in Kap. 5 auf ein Beispiel für eine Menü-geführte Steuerung zurückgreifen zu können.

```
static void interaktiveSteuerung() {
    char steuerzeichen = '0';
    do {
        Konsole.writeString("Steuerzeichen eingeben [lnwe]: ");
        try {
            String eingabe = Konsole.readString();
            if( eingabe.equals("") )
                steuerzeichen = '0';
            else
                steuerzeichen = eingabe.charAt(0);
        } catch( Exception e ) {
            System.exit( 0 );
        }
        switch( steuerzeichen ){
            case 'l':
                String seitenadr;
                Konsole.writeString("Seitenadresse eingeben: ");
                seitenadr = Konsole.readString();
                gestarteteBrowser[aktBrowserIndex] .
                    laden( meinServer.holenSeite( seitenadr ) );
                break;
            case 'n':
                new Browser();
                break;
            case 'w':
                aktBrowserIndex =
                    (aktBrowserIndex+1) % naechsterFreierIndex;
                break;
            case 'e':
                System.exit( 0 );
            default:
                Konsole.writeString("undefiniertes Steuerzeichen\n");
        }
    } while( true );
}
```

Abb. 2.10. Browsersteuerung über die Konsole

Die interaktive Steuerung liest von der Eingabekonsole. Sie versteht vier Steuerzeichen: *l* zum Laden einer neuen Seite im aktuellen Browser – der Na-

me der Seite wird dann vom Benutzer abgefragt; *n* zum Starten eines neuen, weiteren Browsers; *w* zum Wechseln des aktuellen Browsers – gewechselt wird zum Browser mit dem nächsten Index bzw. zum Browser mit dem ersten Index; *e* zum Beenden des Browserprogramms. Für die Ein- und Ausgabe von Zeichenreihen nutzt die interaktive Steuerung die Klasse `Konsole` mit den Klassenmethoden `readString` und `writeString`:

```
class Konsole {
    static String readString() { ... }
    static void writeString( String s ) { ... }
}
```

Als Programmrahmen kann wieder die Klasse `Main` aus Abb. 2.4, S. 62, dienen. Dabei muss allerdings der Aufruf des Konstruktors in der letzten Zeile der `main`-Methode durch den Aufruf der statischen Methode `start` ersetzt werden, also durch die Anweisung `Browser.start(testServer);` (vgl. die online verfügbaren Klassen in `kap2/browseV2`).

2.1.5 Rekursive Klassendeklaration

Eine Definition nennt man *rekursiv*, wenn die definierte Größe im definierenden Teil vorkommt. Beispielsweise ist eine Methode *m* rekursiv definiert, wenn *m* in ihrem eigenen Rumpf aufgerufen wird; man sagt dann auch einfach, dass *m* rekursiv ist. Entsprechend dem zugrunde liegenden Programmierparadigma (vgl. Abschn. 1.2) unterstützen moderne Programmiersprachen rekursive Definitionen bzw. Deklarationen von Prozeduren/Funktionen/Prädikaten/Methoden und Datentypen. Rekursive Datentypdeklarationen sind bei vielen prozeduralen und objektorientierten Programmiersprachen allerdings nur auf dem Umweg über Zeiger möglich. Dieser Abschnitt erläutert an einem Beispiel, wie problemlos der Umgang mit rekursiven Klassen in Java ist. Als Beispiel betrachten wir eine Realisierung doppeltverketteter Listen. Damit verfolgen wir drei Ziele: Erstens soll eine kanonische und wichtige rekursive Datentypimplementierung vorgestellt werden; zweitens soll in das Java-Paket `java.util` eingeführt werden; und drittens schaffen wir uns damit Anschauungsmaterial für Abschn. 2.2.

Bei der Klasse `Browser` von Abb. 2.9 haben wir bereits von rekursiven Definitionen Gebrauch gemacht, ohne es zu erwähnen: Das Klassenattribut `gestarteteBrowser` ist vom Typ `Browser[]`, d.h. der von der Klasse deklarierte Typ wird innerhalb des Rumpfes der Klasse verwendet. In dem Beispiel können allerdings keine rekursiven Abhängigkeiten zwischen `Browser`-Objekten entstehen: Es gibt keine Kette von Referenzen von einem `Browser`-Objekt zu einem anderen. Wie wir im Laufe des Buches sehen werden, sind rekursive Abhängigkeiten in der objektorientierten Programmierung eher der Normalfall als die Ausnahme.

Doppeltverkettete Listen. Ein *Behälter* (engl. *container*) ist ein Datentyp zum Speichern von Elementen bzw. Objekt-Referenzen. Eine Liste ist ein Behälter, in dem ein Element mehrfach enthalten sein kann und in dem die Elemente geordnet sind; d.h. u.a., dass es Sinn macht, nach dem ersten und letzten Element zu fragen und die Elemente der Reihe nach zu durchlaufen. Listen implementiert man üblicherweise mit Hilfe von Feldern, durch einfache Verkettung der Elemente oder durch doppelte Verkettung.

Wir betrachten hier einen leicht vereinfachten Ausschnitt aus der Klasse `LinkedList` des Bibliothekspakets `java.util`, die doppeltverkettete Listen realisiert. Doppeltverkettete Listen ermöglichen effizientes Arbeiten am Anfang und Ende der Liste (z.B. das Anfügen von Elementen) und das flexible Durchwandern der Liste. Das Objektgeflecht zur Repräsentation einer drei-elementigen doppelt verketteten Liste ist in Abb. 2.11 dargestellt. Die entsprechenden Klassendeklarationen bietet Abbildung 2.12. Dabei gehen wir zunächst davon aus, dass Listenelemente vom Typ `ListElem` sind. Wie dieser Typ implementiert ist, spielt hier keine Rolle. (Eine mögliche Implementierung ist auf S. 92 dargestellt.)

Die Klasse `Entry` zur Beschreibung der Verkettungselemente bietet ein typisches Beispiel für eine direkt rekursive Klassendeklaration: `Entry`-Objekte

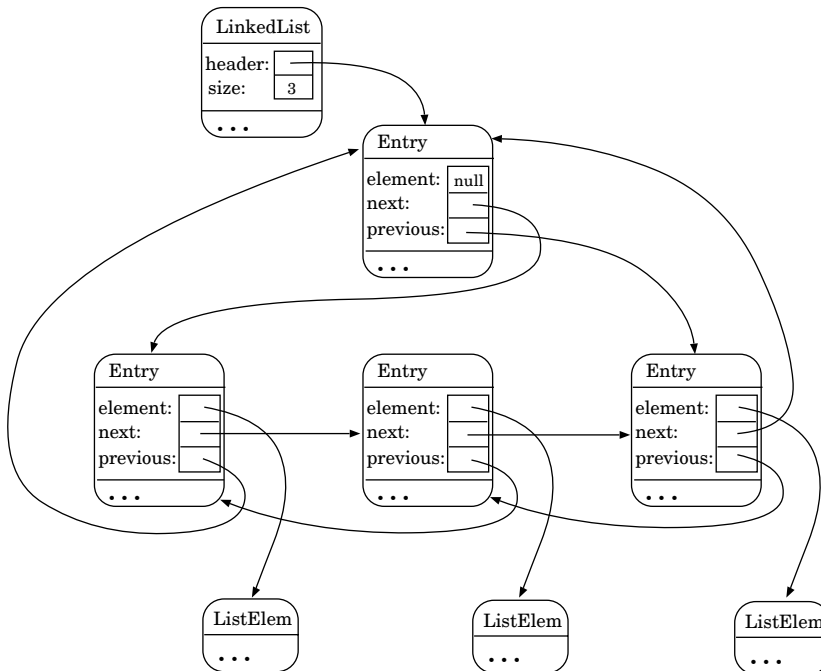


Abb. 2.11. Doppeltverkettete Liste

```

class Entry {
    ListElem element;
    Entry next;
    Entry previous;

    Entry(ListElem element, Entry next, Entry previous) {
        this.element = element;
        this.next = next;
        this.previous = previous;
    }
}

class LinkedList {
    Entry header = new Entry(null, null, null);
    int size = 0;

    /* Constructs an empty Linked List. */
    LinkedList() {
        header.next = header;
        header.previous = header;
    }
    /* Returns the last Element in this List. */
    ListElem getLast() {
        if( size==0 ) throw new NoSuchElementException();
        return header.previous.element;
    }
    /* Removes and returns the last Element from this List. */
    ListElem removeLast() {
        Entry lastentry = header.previous;
        if(lastentry==header) throw new NoSuchElementException();
        lastentry.previous.next = lastentry.next;
        lastentry.next.previous = lastentry.previous;
        size--;
        return lastentry.element;
    }
    /* Appends the given element to the end of this List. */
    void addLast(ListElem e) {
        Entry newEntry = new Entry(e, header, header.previous);
        header.previous.next = newEntry;
        header.previous = newEntry;
        size++;
    }
    /* Returns the number of elements in this List. */
    int size() {
        return size;
    }
}

```

Abb. 2.12. Die Klassen Entry und LinkedList

besitzen Attribute, die direkt Entry-Objekte referenzieren. Der Konstruktor von `LinkedList` zeigt, dass Attribute auch Referenzen auf das Objekt enthalten können, zu dem sie gehören.

2.1.6 Typkonzept und Parametrisierung von Klassen

In typisierten Programmiersprachen wird den für die Ausführung wichtigen Sprachelementen ein Typ zugeordnet, der charakterisiert, welche Operationen mit dem Sprachelement zulässig sind (vgl. dazu Abschn. 1.3.1, insbesondere die S. 25 ff, und die Erläuterungen zum Typsystem von C beginnend auf S. 36). Der Typ eines Objekts gibt darüber Auskunft, welche Attribute das Objekt besitzt und welche Signatur seine Methoden haben. Jede Klasse in Java deklariert einen neuen Typ. Aus der Klassendeklaration kann man die Attribute und deren Typen sowie die erweiterten Methodensignaturen ablesen. In Kap. 3 werden wir sehen, dass zu dem Typ, den eine Klasse K deklariert, nicht nur die Objekte der Klasse K gehören, sondern auch alle Objekte von Klassen, die von K erben.

Die Typisierung ermöglicht es, zur Übersetzungszeit die Korrektheit von Attributzugriffen und Methodenaufrufen zu überprüfen. Betrachten wir dazu die letzte Zeile der Methode `removeLast` von Abb. 2.12:

```
return lastentry.element;
```

Gemäß Typisierung kann die Variable `lastentry` nur Referenzen auf Objekte vom Typ `Entry` speichern oder aber die spezielle Referenz `null`, die zu allen Typen gehört. Zur Ausführungszeit können nur zwei Situationen eintreten: Entweder referenziert `lastentry` ein Objekt vom Typ `Entry`, sodass garantiert ist, dass das Objekt ein Attribut `element` besitzt, oder `lastentry` enthält die Referenz `null`; in diesem Fall terminiert die Ausführung abrupt mit der Ausnahme `NullPointerException` (vgl. S. 56). Darüber hinaus gewährleistet die Typisierung, dass die angegebene `return`-Anweisung bei normaler Terminierung ein Objekt vom Typ `ListElem` zurückliefert. Besäße Java kein Typkonzept, könnte die Variable `lastentry` beliebige Objekte referenzieren, insbesondere auch Objekte, die kein Attribut `element` besitzen. Um fehlerhafte Speicherzugriffe zu verhindern, müsste deshalb zur Ausführungszeit bei jedem Zugriff geprüft werden, ob das referenzierte Objekt ein Attribut `element` besitzt.

Die Diskussion der Typisierung von Variablen und Attributen lässt sich direkt auf Methoden übertragen. Die Typisierung garantiert, dass das Empfängerobjekt, das bei einem Methodenaufruf ermittelt wurde, eine Methode mit entsprechender Signatur besitzt. Die Typisierung verbessert also die Möglichkeiten, Programme automatisch zur Übersetzungszeit zu überprüfen, und führt im Allgemeinen auch zu einem Effizienzgewinn, da Prüfungen zur Ausführungszeit entfallen können. Außerdem bietet die Typisierung gleichzeitig eine rudimentäre Dokumentation der Programmelemente.

Den Vorteilen, die die Typisierung mit sich bringt, stehen im Wesentlichen zwei Nachteile gegenüber. Erstens wird der Schreibaufwand größer und der Programmtext länger. Zweitens kann die Typisierung auch wie ein Korsett wirken, das die Flexibilität des Programmiers einengt. Um dies zu illustrieren, wenden wir uns nochmals der Klasse `LinkedList` von Abb. 2.12 zu. Würde Java keine korrekte Typisierung erzwingen, könnten wir die Implementierung von `LinkedList` nutzen, um Listenelemente beliebigen Typs zu verarbeiten. Die Typisierung schreibt aber vor, dass nur Objekte vom Typ `ListElem` in die Liste eingefügt werden dürfen. Wollen wir die Listenimplementierung für String-Objekte verwenden, müssen wir den gesamten Programmtext kopieren und überall den Typbezeichner `ListElem` durch `String` ersetzen. Solche Kopier-/Ersetzungs-Operationen blähen die Programme auf und erschweren die Wartung. Im Zusammenhang mit den oft vorkommenden Behälterklassen ist ein solches Vorgehen inakzeptabel.

Parametrisierung von Klassen. Im Wesentlichen gibt es zwei Ansätze, um die geschilderte Inflexibilität zu überwinden, ohne die Vorteile der Typisierung zu verlieren: 1. Man unterstützt Subtyping (vgl. Abschn. 1.3.2, S. 41); Subtyping wird in Kap. 3 behandelt. 2. Man ermöglicht es, Typen bzw. Klassen zu parametrisieren; dieser Ansatz wird im nächsten Absatz erläutert. Beide Ansätze haben ein unterschiedliches Anwendungsprofil (siehe [Mey88], Kap. 19). Deshalb realisieren die Sprachen Eiffel, C++ und Ada95 beide Techniken. Java unterstützt die Parametrisierung von Typen und Klassen seit der JDK-Version 1.5. Wir demonstrieren im Folgenden nur das zentrale Konzept. Das parametrische Typsystem von Java ist erheblich mächtiger (siehe [NW06]). Parametrische Klassen werden oft auch generische Klassen genannt, da man durch Instanzieren des Parameters unterschiedliche Klassen „generieren“ kann.

Der Typparameter einer Klassendeklaration wird in spitze Klammern eingeschlossen und dem Klassennamen angehängt. Er kann im Rumpf der Klassen wie ein normaler Typname verwendet werden. Parametrisiert man die Klassen `Entry` und `LinkedList` bezüglich des Elementtyps, ergeben sich folgende Klassendeklarationen:

```
class Entry<ET> {
    ET element;
    Entry<ET> next;
    Entry<ET> previous;

    Entry(ET element, Entry<ET> next, Entry<ET> previous) {
        this.element = element;
        this.next = next;
        this.previous = previous;
    }
}
```

```

class LinkedList<ET> {
    Entry<ET> header = new Entry<ET>(null, null, null);
    int size = 0;

    LinkedList(){ header.next=header; header.previous=header; }

    ET getLast() {
        if( size==0 ) throw new NoSuchElementException();
        return header.previous.element;
    }
    ET removeLast() {
        Entry<ET> lastentry = header.previous;
        if(lastentry==header) throw new NoSuchElementException();
        lastentry.previous.next = lastentry.next;
        lastentry.next.previous = lastentry.previous;
        size--;
        return lastentry.element;
    }
    void addLast(ET e) {
        Entry<ET> newEntry =
            new Entry<ET>(e, header, header.previous);
        header.previous.next = newEntry;
        header.previous = newEntry;
        size++;
    }
    int size() { return size; }
}

```

Die parametrischen Klassen kann man dann bei der Anwendung mit unterschiedlichen Typen instanzieren. Dabei wird der formale Typparameter, im Beispiel also ET, an allen Stellen konsistent durch einen Typpnamen ersetzt. Die folgende ListTest-Klasse (siehe Programmverzeichnis paramliste) demonstriert die Instanzierung der parametrischen Liststenklasse LinkedList<ET> zu den beiden verschiedenen Liststenklassen LinkedList<String> und LinkedList<W3Seite> mit den Elementtypen String und W3Seite.

```

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

        LinkedList<String> ls = new LinkedList<String>();
        ls.addLast("erstes Element");
        ls.addLast("letztes Element");
        ls.getLast().indexOf("Elem"); // liefert 8

        LinkedList<W3Seite> lw3 = new LinkedList<W3Seite>();
        lw3.addLast( new W3Seite("Erste Seite","Kein Text") );
        lw3.addLast( new W3Seite("Zweite Seite","Leer") );
        lw3.getLast().indexOf("Elem"); // Programmierfehler,
        // der vom Uebersetzer automatisch entdeckt wird
    } }

```

Beachtenswert ist jeweils der Aufruf von `getLast`: Auf Grund der Typinformation von `ls` bzw. `lw3` kann der Übersetzer den Typ des Ergebnisses ermitteln. Im ersten Fall ist dies `String`; der Aufruf von `indexOf` ist also korrekt. Im zweiten Fall erkennt der Übersetzer einen Programmierfehler: Objekte des Typs `W3Seite` besitzen keine Methode `indexOf`.

Andere Formen der Parametrisierung von Klassen und eine Verfeinerung des Typsystems werden wir in Kap. 3 kennen lernen.

2.1.7 Klassen als Objekte

Bisher haben wir Klassen als (textuelle) Beschreibung von Objekten betrachtet. Gibt es Anwendungen, bei denen es sinnvoll ist, Klassen selbst wie Objekte zu behandeln? Welche Methoden würde man sich für diese „Klassenobjekte“ wünschen? Bevor wir diese Fragen angehen, wollen wir kurz eine konzeptionelle Klärung versuchen, inwieweit Klassen eher Objekte oder Werte sind. Dazu benutzen wir die Kriterien aus Unterabschn. 1.3.1.1, S. 22: Als den Zustand einer Klasse kann man die Belegung der Klassenattribute ansehen⁹. Was die Identität einer Klasse ist, d.h. wann man zwei Klassen als identisch zu betrachten hat, lässt sich unterschiedlich definieren. In Java wird die Identität einer Klasse durch deren *vollständigen Namen* bestimmt, wobei der vollständige Name einer Klasse aus dem Namen des Pakets, zum dem die Klasse gehört, und dem Klassennamen besteht (siehe Unterabschn. 2.2.2.2). Der vollständige Klassenname wird benutzt, um den Programmcode der Klasse aufzufinden; er bezeichnet quasi den Ort der Klasse. Zwei textuell identische Klassen mit verschiedenen vollständigen Namen werden als unterschiedlich betrachtet. Wie wir im Folgenden sehen werden, bietet es sich auch bei bestimmten Aufgabenstellungen an, mit Klassen über Nachrichten zu kommunizieren. Es kann also durchaus Sinn machen, eine Klasse wie ein Objekt zu behandeln.

Im Wesentlichen kann man einer Klasse drei Aufgaben zuschreiben: 1. Sie muss Instanzen/Objekte gemäß ihren eigenen Vorgaben erzeugen können. 2. Sie sollte in der Lage sein, über sich Auskunft zu erteilen. 3. Sie sollte Hilfestellung bei der Konstruktion neuer Klassen geben können. Betrachtet man eine Klasse als Objekt, müsste sie für diese Aufgaben Methoden bereitstellen. Bisher haben wir uns nur mit der ersten Aufgabe beschäftigt. Die Objekterzeugung erfolgt implizit im Zusammenhang mit einem Konstruktoraufruf. Sieht man von syntaktischen Aspekten ab (syntaktische Form der Konstruktordeklaration, Schlüsselwort `new` bei der Objekterzeugung), gibt es keine wesentlichen Unterschiede zwischen der Objekterzeugung und dem Aufruf von Klassenmethoden.

⁹ Präzise besehen, gilt das aber nur innerhalb eines Prozesses.

Die zweite Aufgabe erlangt Bedeutung, wenn man Werkzeuge bauen möchte, die in der Lage sind, unbekannte Klassen zusammenzubauen. Dafür müssen diese Werkzeuge in die Klassen hineinschauen und deren Methoden, Felder und Konstruktoren erfragen können. Derartiges Hineinschauen in Klassen nennt man *Introspektion*. Manchmal reicht es aber nicht aus, nur zu erfahren, wie die Attribute und Methoden einer fremden Klasse heißen: Man möchte sie auch anwenden können. Programmieretechniken, die beides ermöglichen, werden *reflexiv* genannt. Oft spricht man auch einfach von *Reflexion*. In diesem Abschnitt werden wir an zwei kleinen Beispielen illustrieren, wie Java Introspektion und Reflexion unterstützt.

Die dritte der genannten Aufgaben werden wir nicht behandeln. Sie gehört konzeptionell in den Zusammenhang von Kap. 3. Es sei allerdings bereits hier darauf hingewiesen, dass Java diese Aufgabe nicht unterstützt. D.h. es gibt in Java keine Nachricht, die man einer Klasse schicken kann, um sich eine erweiterte oder modifizierte Klasse als Ergebnis geben zu lassen. Eine klassische Realisierung dieses Konzepts findet man in der Sprache Smalltalk.

Introspektion. Eine typische Anwendung von Introspektion ist es zu ermitteln, welche Methoden eine Klasse zur Verfügung stellt. Anhand dieser Anwendung wollen wir uns hier die Grundlagen von Javas reflexiven Fähigkeiten erarbeiten. Im Bibliothekspaket `java.lang` stellt Java eine Klasse mit Namen `Class` zur Verfügung. Jedes Objekt der Klasse `Class` beschreibt einen Typ. Insbesondere gibt es für jeden Klassentyp bzw. jede Klasse und für jeden Basisdatentyp ein Objekt vom Typ `Class`. Die Klasse `Class` bietet naturgemäß keinen Konstruktor an, da die Menge ihrer Objekte durch die vorhandenen Klassendeklarationen und Basisdatentypen bestimmt ist. Stattdessen gibt es eine statische Methode `forName`, die zu einem Klassennamen das zugehörige `Class`-Objekt liefert. Die Klassennamen müssen dabei vollständig mit dem Paketnamen qualifiziert sein: So liefert beispielsweise der Aufruf `Class.forName("java.util.LinkedList")` das `Class`-Objekt zurück, das die Klasse `LinkedList` des Java-Bibliothekspakets `java.util` beschreibt. (Die Qualifizierung von Klassen mit Paketnamen wird in Unterabschn. 2.2.2.2 behandelt.) Zur Vereinfachung bietet Java auch die Syntax `Typ.name.class` an, um sich zu einem Typ das zugehörige `Class`-Objekt zu besorgen. Insbesondere liefert `int.class` das `Class`-Objekt für den Basisdatentyp `int`.

`Class`-Objekte besitzen Methoden, mit denen man sich alle wünschenswerten Informationen über die Klasse besorgen kann, insbesondere über die Methoden, Konstruktoren und Attribute. Darüber hinaus gibt es die Methode `newInstance`, die eine neue Instanz der Klasse erzeugt, welche das `Class`-Objekt repräsentiert; d.h. `LinkedList.class.newInstance()` ist im Wesentlichen gleichbedeutend mit `new LinkedList()`. Für unser Beispiel benötigen wir die Methode `getMethods`, die alle mit `public` deklarierten Methoden einer Klasse als Ergebnis liefert. Um dieses Ergebnis

programmtechnisch behandeln zu können, gibt es in der Java-Bibliothek ein Paket `java.lang.reflect`, das eine Klasse `Method` zur Verfügung stellt. Ein `Method`-Objekt repräsentiert eine Methode. Es stellt Methoden zur Verfügung, mit denen man sich alle wünschenswerten Informationen über die repräsentierte Methode besorgen kann. So liefert `getReturnType` den Ergebnistyp, `getName` den Methodennamen und `getParameterTypes` die Parametertypen. Damit lässt sich das angestrebte Beispiel wie folgt implementieren:

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

public class Inspektor {
    public static void main(String[] ss) {
        try{
            Class klasse = Class.forName( ss[0] );
            Method[] methoden = klasse.getMethods();
            for( int i = 0; i < methoden.length; i++ ){
                Method m = methoden[i];
                Class retType = m.getReturnType();
                String methName = m.getName();
                Class[] parTypes = m.getParameterTypes();
                System.out.print( retType.getName()+" "+methName+" (" );
                for( int j = 0; j < parTypes.length; j++ ){
                    if( j > 0 ) System.out.print(", ");
                    System.out.print( parTypes[j].getName() );
                }
                System.out.println( ");" );
            }
        } catch( ClassNotFoundException e ) {
            System.out.println("Klasse "+ss[0]+" nicht gefunden");
        }
    }
}
```

Das Beispiel illustriert nicht nur eine Anwendung von Introspektion. Es bietet auch ein hilfreiches Werkzeug. Das `Inspektor`-Programm listet nämlich nicht nur die öffentlichen Methoden, die in der Klasse deklariert sind, sondern auch die ererbten Methoden (zur Vererbung siehe Kap. 3). Insbesondere liefert der Programmaufruf `java Inspektor Inspektor` die Liste aller Methoden von Klasse `Inspektor`; und das sind zusätzlich zur Methode `main` neun Methoden, die von der Klasse `Object` geerbt sind.

Darüber hinaus demonstriert das `Inspektor`-Programm den Aspekt des dynamischen Ladens von Klassen, den wir auf S. 58 kurz angesprochen haben. Erst durch den Aufrufparameter `ss[0]`, der natürlich von Ausführung zu Ausführung verschieden sein kann, wird bestimmt, welches `Class`-Objekt der Aufruf von `forName` liefern soll. Ist die entsprechende Klasse noch nicht Teil des ausgeführten Programms, wird sie dynamisch zum Programm hinzugeladen. Dabei kann sich die zu ladende Klasse, repräsentiert durch ei-

ne .class-Datei, im Allgemeinen auf einem beliebigen Rechner irgendwo im Internet befinden; selbstverständlich muss der String-Parameter der Methode `forName` die Klasse und ihren Ort eindeutig beschreiben (vgl. Unterabschn. 2.2.2.2).

Die Technik, Klassen als Objekte zu behandeln, erleichtert es demnach auch, laufende Programme um neue Klassen zu erweitern. Mit ihr können ausführbare Programme relativ klein gehalten werden, da immer nur die aktuell benötigten Programmteile vorhanden sein müssen. Dies ist gerade im Zusammenhang mit der Programmierung von Anwendungen wichtig, die interaktiv über Netze geladen werden sollen.

Reflexion. Techniken zur Reflexion erlauben es, innerhalb von Programmen mit programmtechnischen Mitteln Programme zu analysieren und anzuwenden: „Programme reflektieren über sich selbst“. Solche reflexiven Techniken auf Programmebenen gibt es seit dem Reifen der Lisp-Programmierung in den späten sechziger Jahren. Im Zuge komponentenbasierter Softwareentwicklung erlangen sie allerdings erst in jüngster Zeit praktische Bedeutung.

Zur Einführung in reflexive Techniken betrachten wir eine Methode `filter`, die aus einem Feld von Zeichenreihen diejenigen Zeichenreihen auswählt, die einen Test bestehen. Die ausgewählten Zeichenreihen werden in einem Feld zurückgegeben. Der Test wird durch eine statische Methode beschrieben, die eine Zeichenreihe als Argument nimmt und ein Objekt vom Typ `Boolean` als Ergebnis liefert. Die `test`-Methode soll ein Parameter der Methode `filter` sein, d.h. wir wollen mit unterschiedlichen Tests filtern können. Hier ist eine mögliche Realisierung von `filter`, die wir im Folgenden genauer betrachten wollen:

```
static String[] filter( String[] strs, Method test ) {
    int i, j = 0;
    String[] aux = new String[ strs.length ];
    for( i = 0; i < strs.length; i++ ) {
        Object[] aktPars = { strs[i] };
        try{
            Boolean bobj = (Boolean) test.invoke(null, aktPars);
            if( bobj.booleanValue() ) {
                aux[j] = strs[i];
                j++;
            }
        } catch( Exception e ){
            System.out.println("Aufruf mittels invoke fehlerhaft");
        }
    }
    String[] ergebnis = new String[ j ];
    System.arraycopy( aux, 0, ergebnis, 0, j );
    return ergebnis;
}
```

Die test-Methode wird als Parameter vom Typ `Method` der filter-Methode übergeben. Sie wird mittels `invoke` für jede Zeichenreihe des übergebenen Feldes aufgerufen. Verläuft der Test positiv, wird die getestete Zeichenreihe in das Hilfsfeld `aux` eingetragen. Am Ende wird das `aux`-Feld in ein Ergebnisfeld kopiert und zurückgegeben (zur Methode `arraycopy` siehe S. 70). Der Aufruf der test-Methode mittels `invoke` birgt einige programmtechnische Schwierigkeiten. Die Methode `invoke` gehört zur Klasse `Method` und hat die Signatur

```
Object invoke( Object obj, Object[] args )
```

wobei der erste Parameter dem impliziten Parameter der aufzurufenden Methode entspricht und das Feld `args` die expliziten Parameter beherbergt. Bei statischen Methoden verwendet man wie im Beispiel `null` als impliziten Parameter. Dass die Parameter und das Ergebnis vom Typ `Object` sein müssen, bringt zwei Konsequenzen mit sich: Erstens können keine Werte der Basisdatentypen als Argumente bzw. Ergebnis verwendet werden, sondern nur Objekte. Deshalb können wir als Ergebnistyp der test-Methode nicht `boolean` verwenden, sondern müssen den Ergebniswert in Objekte der Klasse `Boolean` verpacken, die für diesen Zweck vom Bibliothekspaket `java.lang` bereitgestellt wird. Die Methode `booleanValue` holt aus einem `Boolean`-Objekt den booleschen Wert heraus. Zweitens müssen die Typüberprüfungen an den Parametern und bei der Typkonvertierung des Ergebnisses zur Laufzeit durchgeführt werden. Dies ist eine Fehlerquelle, die man im Auge behalten sollte und die eine adäquate Fehlerbehandlung notwendig macht.

Abbildung 2.13 demonstriert eine Anwendung der Methode `filter`. Beim ersten Aufruf werden mittels der Methode `prefixAoderB` diejenigen Zeichenreihen ausgewählt, die mit „A“ oder „B“ beginnen. Das `Method`-Objekt, das der Methode `prefixAoderB` entspricht, wird dabei von der Methode `getMethod` der Klasse `Class` geliefert. Der zweite Aufruf von `filter` ergibt eine lexikographisch geordnete Sequenz von Zeichenreihen, die aus der übergebenen Sequenz entsteht, wenn man alle Zeichenreihen streicht, die lexikographisch vor ihrem Vorgänger anzuordnen wären (`compareTo` geht dabei allerdings von einer Ordnung aus, in der alle Großbuchstaben vor den Kleinbuchstaben angeordnet sind).

Fazit. Es kann durchaus Sinn machen, Elemente von Programmen als Objekte zu repräsentieren und damit der („Meta“-)Programmierung zugänglich zu machen. Wir haben dies am Beispiel von Klassen und Methoden in Java demonstriert. Andere Programmiersprachen nutzen solche reflexiven Techniken wesentlich intensiver. Beispielsweise werden in `Smalltalk` sogar Anweisungsblöcke als Objekte behandelt (siehe Kap. 8). Reflexive Techniken sind sehr hilfreich, um die dynamische Konfiguration von Programmen zu beschreiben und Werkzeuge zur Komposition von Programmen zu realisieren.

```

import java.lang.reflect.*;

public class InvokeTest {

    static String[] filter( String[] strs, Method test ) {
        ... // siehe Text
    }

    static void printStringArray( String[] strs ) {
        for( int i = 0; i< strs.length; i++ )
            System.out.println( strs[i] );
    }

    public static void main(String[] args) throws Exception {
        Class cl = InvokeTest.class;
        Class[] parTypes = { String.class };
        Method m = cl.getMethod("prefixAoderB",parTypes);
        System.out.println("Ausgabe von prefixAoderB:");
        printStringArray( filter( args, m ) );

        cl= Lexik.class;
        m = cl.getMethod("lexikographisch",parTypes);
        System.out.println("Ausgabe von lexikographisch:");
        printStringArray( filter( args, m ) );
    }

    public static Boolean prefixAoderB( String s ) {
        if(      s.length() > 0
            && ( s.charAt(0)== 'A' || s.charAt(0)=='B' ) )
            return new Boolean( true );
        else return new Boolean( false );
    }
}

class Lexik {
    static String aktMax = "";
    public static Boolean lexikographisch( String s ) {
        if( s.compareTo( aktMax ) >= 0 ) {
            aktMax = s;
            return new Boolean( true );
        } else return new Boolean( false );
    }
}

```

Abb. 2.13. Anwendung der Methode filter

2.2 Kapselung und Strukturierung von Klassen

Der vorangegangene Abschnitt hat erläutert, wie man mit Klassen Objekte beschreiben kann. Dabei haben wir verschiedene Fragen zunächst zurückgestellt: Wie kann man Objekte vor unerwünschtem Zugriff schützen? Wie lassen sich Mengen von Klassen strukturieren? In welchen Beziehungen können Klassen bzw. ihre Objekte zueinander stehen? Erste Antworten auf derartige Fragestellungen enthält dieser Abschnitt.

2.2.1 Kapselung und Schnittstellenbildung: Erste Schritte

Bisher sind wir stillschweigend davon ausgegangen, dass jeder Benutzer eines Objektes auf alle Attribute zugreifen kann, dass er alle Attribute verändern kann und dass er alle Methoden und Konstruktoren aufrufen kann. Programmtechnisch gesehen heißt das, dass es an jeder Programmstelle erlaubt ist, die genannten Operationen zu nutzen; insbesondere dürfen die Attribute eines Objektes einer Klasse K auch in Methoden geändert werden, die nicht zu K gehören. Im Wesentlichen gibt es zwei Gründe, eine derart freizügige Benutzung von Objekten zu unterbinden:

1. Der Benutzer einer Klasse kennt die Implementierung selten in allen Details. Er läuft deshalb leicht Gefahr, durch unsachgemäße Benutzung von Objekten Konsistenzkriterien zu verletzen, die für das korrekte Funktionieren der Methoden notwendig sind.
2. Bestimmte Implementierungsaspekte einer Klasse sollten vor den Benutzern verborgen bleiben, sodass Änderungen dieser Aspekte keine Änderungen bei den Anwendungen der Klasse notwendig machen.

Anhand kleiner Beispiele wollen wir diese zwei Gründe etwas näher erläutern. Wir betrachten zunächst die Klasse `LinkedList` von S. 76. Damit ihre Methoden korrekt funktionieren, muss das Attribut `size` den richtigen Wert haben und die Entry-Objekte müssen wie in Abb. 2.11 miteinander verkettet sein: Ist das Attribut `size` falsch besetzt, könnte die Ausnahme `NoSuchElementException` erzeugt werden, obwohl die Liste nicht leer ist. Manipuliert ein Benutzer beispielsweise eine Liste direkt durch Verändern der Attribute (also ohne Methoden zu verwenden) und vergisst dabei, dem `previous`-Attribut des letzten Entry-Objektes die Referenz auf den Header-Entry zuzuweisen, wird die Listenrepräsentation inkonsistent; und die Methode `removeLast` wird sich im Allgemeinen nicht mehr korrekt verhalten. Die zu beachtenden Konsistenzkriterien einer Klasse entsprechen den invarianten Eigenschaften aller Objekte dieser Klasse. Deshalb spricht man vielfach auch von den *Klasseninvarianten*.

Hinter dem zweiten Grund steht das Prinzip des *Information Hiding*, im Deutschen oft als *Geheimnisprinzip* bezeichnet: Man stelle dem Benutzer

nur die Schnittstelle zur Verfügung, die er für die Anwendung der Klasse braucht. Alle hinter der Schnittstelle verborgenen Implementierungsteile können dann geändert werden, ohne dass dadurch Änderungen bei den Anwendungen notwendig werden. Natürlich gilt das nur, solange die Schnittstellen von den Änderungen nicht betroffen sind. Wäre beispielsweise garantiert, dass Benutzer der Klasse `W3Seite` niemals auf die Attribute `titel` und `inhalt` zugreifen, könnte man die Implementierung von `W3Seite` wie folgt verändern, ohne dass Benutzer davon informiert werden müssten: Titel und Inhalt werden zu einer Zeichenreihe zusammengesetzt; dabei wird der Titel wie in HTML durch "<TITLE>" und "</TITLE>" geklammert¹⁰. Die zusammengesetzte Zeichenreihe wird im Attribut `seite` gespeichert. Die Methoden `getTitel` und `getInhalt` führen dann die Zerlegung durch:

```
class W3Seite {
    private String seite;

    W3Seite( String t, String i ) {
        seite = "<TITLE>" + t + "</TITLE>" + i ;
    }
    String getTitel(){
        int trennIndex = seite.indexOf("</TITLE>");
        return new String( seite.toCharArray(),7,trennIndex-7 );
    }
    String getInhalt(){
        int trennIndex = seite.indexOf("</TITLE>");
        return new String( seite.toCharArray(), trennIndex+8,
                           seite.length() - (trennIndex+8) );
    }
}
```

Die neue Version der Klasse `W3Seite` illustriert am Beispiel des Attributs `seite`, wie Benutzern der Zugriff auf Komponenten einer Klasse verwehrt werden kann: Durch Voranstellen des Zugriffsmodifikators `private` können Attribute, Methoden und Konstruktoren als *privat* vereinbart werden. Private Programmelemente, d.h. als privat deklarierte Attribute, Methoden und Konstruktoren, dürfen nur innerhalb der Klassendeklaration angewendet werden. Dies wird vom Übersetzer geprüft.

Durch Deklaration eines privaten Konstruktors in einer Klasse *K* kann man insbesondere verhindern, dass *K*-Objekte erzeugt werden können, da der private Konstruktor außerhalb von *K* nicht aufgerufen werden kann und der default-Konstruktor (siehe S. 56) nicht mehr bereitsteht. Dies kann für Klassen sinnvoll sein, die nur statische Methoden und Attribute besitzen; ein typisches Beispiel bietet die Klasse `System`.

¹⁰ Dabei gehen wir davon aus, dass die Zeichenreihe "</TITLE>" nicht als Teilzeichenreihe in Titeln enthalten ist.

Schnittstellenbildung. Private Attribute, Methoden und Konstruktoren einer Klasse stehen dem Benutzer nicht zur Verfügung und gehören dementsprechend auch nicht zur öffentlichen Schnittstelle der Klasse. Diese explizite Trennung zwischen der öffentlichen Schnittstelle, d.h. der Benutzersicht, und der privaten Schnittstelle, d.h. der Implementierungssicht, bringt die oben angeführten Vorteile mit sich und führt häufig zu einem klareren Klassenentwurf: Der Benutzer kann sich auf das Erlernen der öffentlichen Schnittstelle konzentrieren; der Implementierer besitzt ein Sprachkonstrukt, um interne Implementierungsteile zu kapseln und klar von extern verfügbaren Teilen zu trennen. In vielen Fällen erweist es sich dabei als sinnvoll, Attribute als privat zu deklarieren und den Zugriff nur über Methoden zu ermöglichen. Die Methoden sollten dann so entworfen werden, dass klar zwischen Methoden unterschieden werden kann, die nur lesend auf Attribute zugreifen, und solchen, die den Zustand von Objekten verändern. (Beispielsweise greifen die Methoden `getTitle` und `getInhalt` der Klasse `W3Seite` nur lesend auf das Attribut `seite` zu.)

In den folgenden Abschnitten und in Kap. 3 werden wir weitere Sprachkonstrukte zur Kapselung und zur Beschreibung von Schnittstellen kennen lernen.

2.2.2 Strukturieren von Klassen

Bisher haben wir die Klassendeklarationen eines Programms als eine unstrukturierte Menge betrachtet. Es gab weder die Möglichkeit, eine Klasse einer anderen im Sinne einer Blockschachtelung unterzuordnen, noch die Möglichkeit mehrere Klassen zu Modulen zusammenzufassen. Java stellt beide Strukturierungsmöglichkeiten zur Verfügung: Eine Klasse, die innerhalb einer anderen deklariert ist, wird in Java eine innere Klasse genannt; Unterstützung bei der Modularisierung bieten die sogenannten Pakete. Hand in Hand mit der Behandlung dieser Strukturierungskonstrukte werden wir die dazugehörigen Kapselungstechniken erläutern.

2.2.2.1 Innere Klassen

Die Verwendung von Blockschachtelung in Programmiersprachen hat zwei Gründe: 1. Sie strukturiert die Programme. 2. Sie legt Gültigkeitsbereiche für Namen fest. Die Technik der Schachtelung von Blöcken wurde in Java auf Klassen übertragen: Klassen können als Komponenten anderer Klassen oder innerhalb von Blöcken deklariert werden (Beispiele der zweiten Form werden wir in Kap. 3 kennen lernen). Solche Klassen nennt man *innere Klassen*. Innere Klassen bieten u.a. die Möglichkeit, kleinere Hilfsklassen nahe bei den Programmstellen zu deklarieren, an denen sie gebraucht werden. In Kombination mit den Kapselungstechniken, die wir im letzten Abschnitt behandelt

haben, ergibt sich darüber hinaus ein sehr flexibler Mechanismus, um die Zugreifbarkeit von Implementierungsteilen zu regeln.

Innere Klassen bilden ein recht komplexes Sprachkonstrukt. In diesem Abschnitt werden wir uns auf die Anwendung innerer Klassen zu Strukturierungs- und Kapselungszwecken konzentrieren. Andere Anwendungen werden wir in späteren Kapiteln behandeln. Zunächst betrachten wir sogenannte statische innere Klassen am Beispiel von `LinkedList`. Der zweite Absatz stellt dann nicht-statische innere Klassen am Beispiel von Iteratoren vor.

Statische innere Klassen. In vielen Fällen kommt es vor, dass man zur Implementierung einer Klasse K Hilfsklassen benötigt, d.h. Klassen, die der Benutzer von K nicht kennen muss und die besser vor ihm verborgen bleiben. Eine solche Hilfsklasse ist die Klasse `Entry` für die Implementierung doppeltverketteter Listen (vgl. Abb. 2.12, S. 76). Indem wir die Klasse `Entry` zu einer privaten inneren Klasse von `LinkedList` machen und außerdem die Attribute von `LinkedList` als privat deklarieren, können wir die gesamte Implementierung von doppeltverketteten Listen kapseln. Damit ist kein Benutzer mehr in der Lage, die Invarianten der Klasse zu verletzen. Insgesamt erhält die Klasse `LinkedList` damit folgendes Aussehen (Methodenrumpfe bleiben wie in Abb. 2.12):

```
class LinkedList {
    private Entry header = new Entry(null,null,null);
    private int size = 0;

    LinkedList() {
        header.next = header;
        header.previous = header;
    }
    ListElem getLast() { ... }
    ListElem removeLast() { ... }
    void addLast(ListElem e) { ... }
    int size() { ... }

    private static class Entry {
        private ListElem element;
        private Entry next;
        private Entry previous;

        Entry(ListElem element,Entry next,Entry previous){ ... }
    }
}
```

Die innere Klasse `Entry` ist als privat deklariert; d.h. ihr Typname ist außerhalb der Klasse `LinkedList` nicht ansprechbar (ein Beispiel für eine nicht private innere Klasse werden wir gleich behandeln). Allgemeiner gilt, dass private Programmelemente überall in der am weitesten außen liegenden Klasse zugreifbar sind, aber nicht außerhalb davon; insbesondere

sind private Programmelemente in weiter innen liegenden Klassen zugreifbar. Dazu betrachten wir zwei Beispiele. Wie im obigen Programmfragment von `LinkedList` demonstriert, sind die privaten Attribute von `Entry` im Konstruktor `LinkedList` zugreifbar, d.h. außerhalb der Klassendeklaration von `Entry`. Ein komplexeres Szenario illustrieren die Klassen in Abb. 2.14. Darüber hinaus zeigen sie, wie man mit zusammengesetzten Namen der Form *InKl.ProgEl* Programmelemente *ProgEl* in inneren Klassen *InKl* ansprechen kann. Beispielsweise bezeichnet `FirstFloor.DiningRoom` die innere Klasse `DiningRoom` in Klasse `FirstFloor`. Deren Attribut `size` lässt sich mittels `FirstFloor.DiningRoom.size` ansprechen.

```
class MyClassIsMyCastle {
    private static int streetno = 169;

    private static class FirstFloor {
        private static class DiningRoom {
            private static int size = 36;
            private static void mymessage(){
                System.out.print("I can access streetno");
                System.out.println(": "+ streetno );
            }
        }
    }

    private static class SecondFloor {
        private static class BathRoom {
            private static int size = 16;
            private static void mymess(){
                System.out.print("I can access the ");
                System.out.print("dining room size: ");
                System.out.println(""+ FirstFloor.DiningRoom.size);
            }
        }
    }

    public static void main( String[] argv ) {
        FirstFloor.DiningRoom.mymessage();
        SecondFloor.BathRoom.mymess();
    }
}
```

Abb. 2.14. Schachtelung und Zugriff bei inneren Klassen

Die Klassen in Abb. 2.14 demonstrieren drei verschiedene Zugriffsverhalten: 1. Umfassende Klassen können auf private Programmelemente in tiefer geschachtelten Klassen zugreifen: In der Methode `main` wird auf die Methoden der inneren Klassen `DiningRoom` und `BathRoom` zugegriffen. 2. Innere Klassen können auf private Programmelemente umfassender Klassen zugreifen: In der Methode `mymessage` wird auf das private Attribut `streetno` der

am weitesten außen liegenden Klasse zugegriffen. 3. Innere Klassen können auf private Programmelemente anderer inneren Klassen zugreifen, wenn es eine beide umfassende Klasse gibt: In der Methode `mymess` wird auf das private Attribut `size` der Klasse `DiningRoom` zugegriffen.

Alle bisher gezeigten inneren Klassen sind mittels des Schlüsselworts `static` als statisch deklariert. Ähnlich wie bei statischen Attributen bzw. Methoden ist eine statische innere Klasse nur der umfassenden Klassendeklaration zugeordnet, aber nicht deren Instanzen. Insbesondere darf sie nur statische Attribute oder Methoden der umfassenden Klasse benutzen. Beispielsweise wäre es unzulässig, innerhalb der Konstruktordeklaration von `Entry` auf das Attribut `header` zuzugreifen. Statische innere Klassen dienen dazu, ansonsten gleichrangige Klassen zu strukturieren (dies war der erste Grund zur Verwendung von Schachtelung; siehe oben) und die Kapselungsmöglichkeiten zu verfeinern.

Nicht-statische innere Klassen. Nicht-statische innere Klassen bieten einen mächtigeren Mechanismus als statische innere Klassen: Sei IN eine nicht-statische innere Klasse einer Klasse K ; dann bekommt jedes IN -Objekt implizit eine Referenz auf ein zugehöriges K -Objekt. Das zugehörige K -Objekt kann in der inneren Klasse über den zusammengesetzten Bezeichner `K.this` angesprochen werden. Dadurch kann das Objekt der inneren Klasse insbesondere auf die Attribute und Methoden des zugehörigen K -Objekts zugreifen; sofern es keine Namenskonflikte mit den Attributen der inneren Klasse gibt, kann dafür direkt der Attribut- bzw. Methodenname verwendet werden (also z.B. statt `K.this.attr` direkt `attr`).

Durch diese Verbindung zu einem Objekt der umfassenden Klasse und die Möglichkeit auf dessen Attribute und Methoden zugreifen zu können, insbesondere auch auf die privaten, lassen sich auch komplexe Kapselungsanforderungen meistern. Eine detaillierte Erörterung der resultierenden Möglichkeiten würde in diesem Zusammenhang aber zu weit führen. Wir begnügen uns hier mit der Diskussion eines typischen Beispiels: der Realisierung von Iteratoren für unsere Listenklasse. Iteratoren stellen eine wichtige Programmier Technik dar, die jeder Programmentwickler kennen sollte. (Weitere Beispiele nicht-statischer innerer Klassen werden wir in den folgenden Kapiteln kennen lernen.)

Beim Arbeiten mit Behältern (Mengen, Listen, Schlangen, etc.) steht man häufig vor der Aufgabe, effizient über alle Elemente im Behälter zu iterieren, beispielsweise um auf jedem Element eine bestimmte Operation auszuführen oder um ein bestimmtes Element zu suchen. Diese allgemeine Aufgabenstellung wollen wir anhand eines konkreten Beispiels genauer betrachten und zwar anhand einer einfachen Implementierung der Klasse `W3Server` aus Abschn. 2.1.3, S. 60.

Bei einem `W3-Server` kann man `W3-Seiten` unter einer Adresse ablegen und holen. Die Adresse ist eine beliebige Zeichenreihe. Die Seiten werden in

einer Liste verwaltet, deren Listenelemente Paare sind, die aus einer Adresse und einer W3-Seite bestehen:

```
class ListElem {
    private String adr;
    private W3Seite seite;

    ListElem( String a, W3Seite s ){
        adr    = a;
        seite  = s;
    }
    String  getAdr  () { return adr; }
    W3Seite getSeite() { return seite; }
    void    setSeite( W3Seite s ) { seite = s; }
}
```

Die Liste solcher Paare wird beim W3-Server im Attribut `adrSeitenListe` gespeichert. Wird versucht, unter einer Adresse, zu der keine Seite vorhanden ist, eine Seite zu holen, soll eine Fehlerseite zurückgeliefert werden:

```
class W3Server {
    LinkedList adrSeitenListe      = new LinkedList();
    static final W3Seite fehlerseite =
        new W3Seite("Fehler", "Seite nicht gefunden");
    void ablegenSeite( W3Seite s, String sadr ) { ... }
    W3Seite holenSeite( String sadr ) { ... }
}
```

Wir wollen hier die Methode `holenSeite` entwickeln, wobei die Version der Klasse `LinkedList` von S. 89 verwendet werden soll. Um es uns einfach zu machen, verwenden wir lineare Suche: Die Adressen-Seiten-Liste wird schrittweise durchlaufen; dabei wird jeweils die Adresse der aktuellen Seite mit dem Parameter verglichen; wird eine Seite gefunden, wird sie zurückgegeben; ansonsten wird schließlich die Fehlerseite abgeliefert:

```
W3Seite holenSeite( String sadr ) {
    while( adrSeitenListe hat nächstes Element ) {
        ListElem adp = nächstes Element ;
        if( adp.getAdr().equals( sadr ) ) return adp.getSeite();
    }
    return fehlerseite;
}
```

Wir brauchen nur noch die informellen Anweisungen durch geeigneten Programmcode zu ersetzen. Aber wie soll das gehen? Die Methoden der Klasse `LinkedList` erlauben uns zwar, Listen auf- und abzubauen und festzustellen, ob eine Liste leer ist; das Iterieren über alle Elemente wird aber nicht unterstützt. Und außerhalb der Klasse `LinkedList` können wir das auch nicht realisieren, da die Kapselung den Zugriff auf die Entry-Objekte verbietet.

Zur Lösung dieses Problems versehen wir die Klasse `LinkedList` mit Iteratoren: Ein Listen-Iterator ist ein Objekt, das eine Position in einer Liste repräsentiert. Die Position kann sich zwischen zwei Listenelementen, am Anfang oder am Ende der Liste befinden. Für die Beschreibung der Funktionalität von Iteratoren gehen wir davon aus, dass die Listenelemente wie in Abb. 2.11 von links nach rechts geordnet sind und dass die Elemente von links nach rechts aufsteigend mit Indizes beginnend bei null durchnummeriert sind. Die Position wird in einem Attribut `nextIndex` vom Typ `int` gespeichert, wobei `nextIndex` immer den Index des Elements rechts von der Position enthält; in der Anfangsposition ist also `nextIndex = 0`. Außerdem besitzt jedes Iterator-Objekt ein Attribut `next` vom Typ `Entry`, das immer auf das Listenelement mit Index `nextIndex` verweist. Ein Iterator besitzt die Methoden `hasNext` zur Prüfung, ob die Position am Ende der Liste ist, und `next` zum Zugriff auf das nächste Element und Weiterschalten der Position. Damit Iteratoren auf die `Entry`-Objekte zugreifen können, deklarieren wir die Klasse `ListIterator` als innere Klasse von `LinkedList`:

```
class LinkedList {
    private Entry header = new Entry(null, null, null);
    private int size = 0;
    ... // wie oben

    private static class Entry {
        ... // wie oben
    }

    class ListIterator {
        private int nextIndex = 0;
        private Entry next = header.next;

        boolean hasNext() {
            return nextIndex != size;
        }
        ListElem next() {
            if( nextIndex==size )
                throw new NoSuchElementException();
            ListElem elem = next.element;
            next = next.next;
            nextIndex++;
            return elem;
        }
    }

    ListIterator listIterator() {
        return new ListIterator();
    }
}
```

Die Klasse `ListIterator` ist keine statische Klasse. Damit hat sie Zugriff auf die Attribute der umfassenden Klasse, im Beispiel auf die Attribute `header` und `size` der Klasse `LinkedList`. Um solche Zugriffe zur Ausführungszeit zu ermöglichen, besitzt jedes Objekt einer (nicht-statischen) inneren Klasse eine Referenz auf ein Objekt der umfassenden Klasse. Diese Referenz wird in einem implizit deklarierten Attribut gespeichert. Jedes `ListIterator`-Objekt besitzt also eine implizite Referenz auf das zugehörige `LinkedList`-Objekt. Diese implizite Referenz wird bei der Erzeugung des Objekts der inneren Klasse hergestellt. In unserem Beispiel bekommt dazu der default-Konstruktor `ListIterator` immer implizit das aktuelle `this`-Objekt seiner Aufrufstelle als Argument mitgeliefert. Die Methode `listIterator` liefert also einen Iterator für das `LinkedList`-Objekt, für das sie aufgerufen wurde.

Damit der Programmierer nicht immer eine Methode wie `listIterator` schreiben muss, um Konstruktoren nicht-statischer innerer Klassen aufrufen zu können, wird von Java auch eine spezielle syntaktische Variante dafür angeboten, bei der dem `new`-Ausdruck wie bei einem Methodenaufruf ein implizites Objekt mitgegeben wird. Bezeichnet beispielsweise `meineListe` eine initialisierte Variable vom Typ `LinkedList`, dann ist der Ausdruck `meineListe.new ListIterator()` äquivalent zu dem Ausdruck `meineListe.listIterator()`.

Die Programmierung und das Verstehen der Implementierung innerer Klassen ist zum Teil recht komplex (insbesondere wenn zusätzlich weitere Sprachmittel verwendet werden; vgl. dazu die vollständige Version der Klasse `LinkedList` im Bibliothekspaket `java.util`). Am Beispiel der Listeniteratoren werden wir aber sehen, dass die Anwendung trotzdem sehr einfach sein kann. Man vergleiche dazu die obige unvollständige Version der Methode `holenSeite` mit der vollständigen, die hier nochmals in ihrem Klassenkontext gezeigt ist:

```
class W3Server {
    LinkedList adrSeitenListe = new LinkedList();
    ...
    W3Seite holenSeite( String sadr ) {
        LinkedList.ListIterator lit =
            adrSeitenListe.listIterator();
        while( lit.hasNext() ) {
            ListElem adp = lit.next();
            if( adp.getAdr().equals(sadr) ) return adp.getSeite();
        }
        return fehlerseite;
    }
}
```

Man beachte das Erzeugen eines Iterators für die Adressen-Seiten-Liste mittels der Methode `listIterator`. Im Übrigen sei nochmals auf die Verwendung der Punkt-Notation bei `LinkedList.ListIterator` hingewiesen;

auf diese Weise kann der Typname `ListIterator` außerhalb der Klassendeklaration von `LinkedList` benutzt werden. Einen ähnlichen Mechanismus werden wir im Zusammenhang mit Paketen kennen lernen. (Die vollständigen Klassen `LinkedList`, `ListElem` und `W3Server` in der besprochenen Version finden sich online im Verzeichnis `kap2/browseV2`.)

2.2.2.2 Modularisierung von Programmen: Pakete

Größere Programme und vor allem Programmbibliotheken bestehen oft aus einer Vielzahl von Klassen. Es ist von großer Bedeutung für die Softwareentwicklung, diese Klassen geeignet zu organisieren. Dabei sind die folgenden Aspekte zu beachten:

1. Auffinden, Verstehen: Die gute Strukturierung der Klassen eines großen Programms erleichtert es, den Zusammenhang zwischen Softwareentwurf und Programmcode herzustellen und damit das Programm zu verstehen. Dem Benutzer einer Bibliothek sollte es leicht gemacht werden, Klassen in der Bibliothek ausfindig zu machen, die ihm bei seiner Implementierung helfen können.
2. Pflege: Die Strukturierung der Klassen sollte die Abhängigkeiten zwischen den Klassen widerspiegeln können, um Wartung und Pflege der Programme zu verbessern.
3. Zugriffsrechte, getrennte Namensräume: Klassen sollten zu Modulen zusammengefasst werden können, sodass man den Zugriff von einem Modul auf ein anderes einschränken kann und in verschiedenen Modulen getrennte Namensräume hat.
4. Getrennte Übersetzung: Im Zusammenhang mit der Strukturierung muss auch festgelegt werden, welche Programmteile unabhängig voneinander übersetzt werden können.

Java benutzt sogenannte *Pakete* zur Modularisierung von Programmen. Pakete besitzen eindeutige Namen, werden aber nicht explizit deklariert. Ein Paket umfasst eine Menge sogenannter *Übersetzungseinheiten*. Wie die Bezeichnung suggeriert, lässt sich eine Übersetzungseinheit getrennt von anderen Übersetzungseinheiten übersetzen. Syntaktisch hat eine Übersetzungseinheit in Java die folgende Form:

```
package Paketname ;  
Liste von import-Anweisungen  
Liste von Typdeklarationen
```

Der Paketname gibt an, zu welchem Paket die Übersetzungseinheit gehört. Die import-Anweisungen ermöglichen es, Klassen aus anderen Paketen in der Übersetzungseinheit direkt sichtbar zu machen. Die Typdeklarationen

bilden den eigentlichen Inhalt einer Übersetzungseinheit. Eine *Typdeklaration* ist entweder eine Klassen- oder eine Schnittstellendeklaration (die Deklaration von Schnittstellen behandeln wir in Kap. 3). Wir gehen im Folgenden davon aus, dass jede Übersetzungseinheit genau einer Datei des benutzten Rechners entspricht. Alle Übersetzungseinheiten eines Pakets müssen sich in einem Dateiverzeichnis befinden. Dabei muss der Name des Dateiverzeichnisses dem Paketnamen entsprechen (was das genau heißt, wird weiter unten erläutert). Demnach können wir jedes Paket mit dem Dateiverzeichnis identifizieren, in dem sich seine Übersetzungseinheiten befinden.

Im Folgenden werden wir anhand unseres Browser-Beispiels zeigen, wie Übersetzungseinheiten aussehen, welche Möglichkeiten der Kapselung es bei Paketen gibt, wie auf Bestandteile anderer Pakete zugegriffen werden kann und wie Pakete hierarchisch strukturiert werden können.

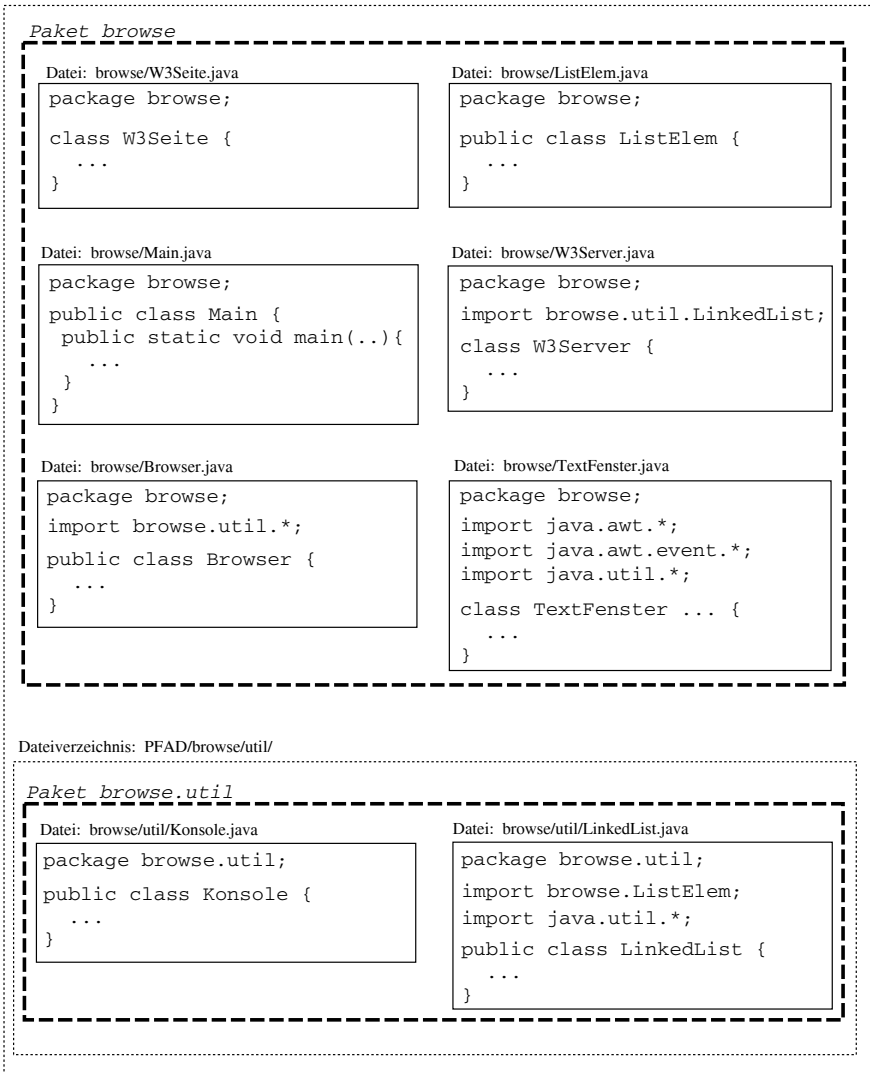
Die acht behandelten Klassen des Beispiels teilen wir auf zwei Pakete auf: das Paket `browse` und das Paket `browse.util`. Das Paket `browse.util` enthält die Klassen, die allgemeine Hilfsaufgaben erledigen, nämlich `Konsole` und `LinkedList`; die restlichen Klassen bilden das Paket `browse`. Wie üblich in Java, sehen wir für jede Klasse eine eigene Übersetzungseinheit, sprich Datei, vor. Die Dateien für das Paket `browse` liegen alle in einem Dateiverzeichnis `browse` mit einem absoluten Namen `PFAD/browse/`, wobei `PFAD` für ein Dateiverzeichnis auf dem benutzten Rechner steht¹¹. Die Dateien für das Paket `browse.util` stehen im Dateiverzeichnis `util` mit absolutem Namen `PFAD/browse/util/`. Abbildung 2.15 zeigt den Zusammenhang zwischen Paketen und Dateiverzeichnissen sowie zwischen Übersetzungseinheiten, Klassen und Dateien. Die durchgezogenen Linien rahmen Übersetzungseinheiten ein und entsprechen damit jeweils einer Datei. Die fetten gestrichelten Linien bezeichnen Paketgrenzen. Die dünnen unterbrochenen Linien markieren die Bereiche von Dateiverzeichnissen. Die Abbildung zeigt insbesondere den Zusammenhang zwischen der Deklaration des Paketnamens am Anfang einer Übersetzungseinheit und dem Namen des entsprechenden Dateiverzeichnisses.

Zum Arbeiten mit kleineren Programmen gestattet es Java, die Deklaration des Paketnamens in der Übersetzungseinheit wegzulassen. Dann werden die Typendeklarationen der Einheit einem unbenannten Paket zugeordnet. Von diesem Sonderfall haben wir bisher Gebrauch gemacht.

Java nutzt die Baumstruktur von Dateisystemen, um Pakete hierarchisch anzuordnen. Die Pakete selber sind aber nicht strukturiert. Insbesondere ist das Paket `browse.util` nicht Teil des Pakets `browse`; beide Pakete sind unabhängig voneinander und die gegenseitige Benutzung muss auf die gleiche

¹¹ Wir benutzen hier, wie unter Unix üblich, den Schrägstrich „/“ als Trenner in Dateipfaden; bei Verwendung anderer Betriebssysteme ist die Pfadsyntax geeignet anzupassen.

Dateiverzeichnis: PFAD/browse/


 *Paket*
 Dateiverzeichnis

 Datei

Abb. 2.15. Strukturierung von Klassen in Pakete

Weise geregelt werden, wie dies bei Paketen erforderlich ist, die in getrennten Dateiverzeichnissen stehen.

Benutzung von Paketen. Grundsätzlich kann jede Klasse alle Klassen in anderen Java-Paketen dieser Welt benutzen. Es gibt allerdings drei Einschränkungen:

1. Kapselung auf der Ebene von Paketen: Java ermöglicht es, Klassen innerhalb von Paketen zu kapseln und damit dem Zugriff durch andere Pakete zu entziehen (siehe nächsten Absatz).
2. Erreichbarkeit: Das gewünschte Paket/Dateiverzeichnis muss dem benutzten Rechner bzw. dem an ihm arbeitenden Benutzer unter einem geeigneten Namen zugänglich sein.
3. Eindeutigkeit: Die vollständigen Klassennamen bestehend aus Paket- und Klassennamen müssen eindeutig sein.

Auf eine Klasse K eines fremden Pakets q kann man zugreifen, indem man den Paketnamen dem Klassennamen voranstellt, d.h. den sogenannten *vollständigen* Namen $q.K$ verwendet. Nehmen wir beispielsweise an, dass auf unserem Rechner die Pakete `testpaket` und `nocheinpaket.namen` existieren. Dann können wir in der Klasse `testpaket.Main` die Klasse `Konflikt` des Pakets `nocheinpaket.namen` und damit ihre statische Methode `slogan` wie folgt benutzen:

```
package testpaket;

public class Main {
    public static void main(String[] args) {
        nocheinpaket.namen.Konflikt.slogan();
    }
}
```

wobei die Klasse `Konflikt` beispielsweise wie folgt aussehen könnte:

```
package nocheinpaket.namen;

public class Konflikt {
    public static void slogan() {
        System.out.println("Ein Traum von Namensraum");
    }
}
```

Damit Java-Übersetzer und Java-Laufzeitumgebungen Pakete und Klassen auf dem lokalen oder einem entfernten Rechner finden können, muss man ihnen mitteilen, in welchen Dateiverzeichnissen sie danach suchen sollen. Diese Information erhalten sie üblicherweise über die Umgebungsvariable `CLASSPATH` oder einen ähnlichen Mechanismus, der angibt, wie auf die Pakete bzw. auf die Klassen zugegriffen werden soll. Der `CLASSPATH`-Mechanismus, den wir im Rest des Buches zugrunde legen, funktioniert auf folgende

Weise: Enthält die Umgebungsvariable CLASSPATH beispielsweise bei der Übersetzung die Namen der Verzeichnisse `/home/gosling/myjava/` und `/jdk1.2/classes/`, würde ein Übersetzer die Klasse `Konflikt` in den Dateiverzeichnissen `/home/gosling/myjava/nocheinpaket/namen/` und `/jdk1.2/classes/nocheinpaket/namen/` suchen. Entsprechendes gilt für Laufzeitumgebungen. Um Namenskonflikte zu vermeiden, sollte man darauf achten, dass Pakete in unterschiedlichen Verzeichnissen verschiedene Namen besitzen. Eine andere Quelle von Namenskonflikten hängt mit der Punktnotation bei der Benutzung innerer Klassen zusammen. Beispielsweise würde die Existenz der folgenden Klasse dazu führen, dass der Aufruf von `slogan` in der obigen Klasse `Main` mehrdeutig wird:

```
package nocheinpaket;

public class namen {
    public static class Konflikt {
        public static void slogan() {
            System.out.println("Pakete zollfrei importieren");
        }
    }
}
```

Diese Art von Mehrdeutigkeit kann verhindert werden, wenn man sich an die Konvention hält, Paketnamen mit Klein- und Klassennamen mit Großbuchstaben zu beginnen.

Da es in der Praxis umständlich und der Lesbarkeit abträglich wäre, Programmelementen aus anderen Paketen immer den Paketnamen voranstellen zu müssen, unterstützt Java einen import-Mechanismus. Nach der Deklaration des Paketnamens am Anfang einer Übersetzungseinheit kann eine Liste von import-Anweisungen angegeben werden, wobei es zwei Varianten von import-Anweisungen gibt:

```
import Paketname.Typname ;
import Paketname.* ;
```

Mit der ersten Deklaration importiert man die Typdeklaration mit Namen *Typname* aus Paket *Paketname*. Der Typ, also insbesondere ein Klassentyp, kann dann in der Übersetzungseinheit direkt mit seinem Namen angesprochen werden, ohne dass der *Paketname* vorangestellt werden muss. Der importierte Typ muss allerdings als öffentlich deklariert sein (der folgende Paragraph beschreibt den Unterschied zwischen öffentlichen und nicht öffentlichen Typen). Mit der zweiten Deklaration importiert man alle öffentlichen Typen des Pakets *Paketname*. Beispielsweise werden durch die Deklaration

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

alle öffentlichen Typen des Pakets `java.lang.reflect` importiert, also insbesondere der Typ `Method` (vgl. Abb. 2.13, S. 85). Bei dieser zweiten Form

von import-Anweisung werden nur die Typen importiert, für die es ansonsten keine Deklaration in der Übersetzungseinheit gibt. Enthielte eine Übersetzungseinheit mit obiger import-Anweisung also eine eigene Typdeklaration `Method`, würde diese in der Übersetzungseinheit verwendet und nicht diejenige aus dem Paket `java.lang.reflect`.

Das Standard-Paket `java.lang` der Java-Bibliothek wird automatisch von jeder Übersetzungseinheit importiert. Es sei nochmals darauf hingewiesen, dass zu diesem Paket nur die Typen gehören, die direkt im `java.lang` entsprechenden Verzeichnis stehen. Insbesondere werden also die Typen des Pakets `java.lang.reflect` nicht automatisch in jede Übersetzungseinheit importiert.

Kapselung auf Paketebene. In Abschn. 2.2.1 wurde erläutert, warum man Teile einer Klassenimplementierung dem Zugriff der Benutzer entziehen möchte und wie Java dieses Kapselungskonzept unterstützt. Aus den gleichen Gründen macht es Sinn, nur ausgewählte Klassen und nur Teile dieser Klassen der Benutzung in anderen Paketen zugänglich zu machen, d.h. Teile von Paketen zu kapseln.

Programmelemente, d.h. Klassen, Attribute, Methoden und Konstruktor, deren Benutzung in allen Paketen gestattet sein soll, müssen als *öffentlich* deklariert werden. Dazu stellt man ihrer Deklaration den Modifikator `public` voran. Ein kleines Beispiel bietet die obige Deklaration der öffentlichen Klasse `Konflikt` mit der öffentlichen Methode `slogan`. Programmelemente, die weder öffentlich noch privat sind, können überall in dem Paket benutzt werden, in dem sie deklariert sind; ohne weitere Angabe ist ein Zugriff von außerhalb des Pakets nicht zulässig¹². Wir sagen, dass solche Programmelemente *paketlokalen* Zugriff erlauben und sprechen abkürzend von *paketlokalen* Programmelementen. Im Java-Jargon ist auch der Terminus *default access* verbreitet, da paketlokaler Zugriff gilt, wenn bei der Deklaration eines Programmelementes keine Zugriffsmodifikatoren angegeben sind.

Als ein interessanteres Beispiel zur Diskussion von Kapselung betrachten wir die beiden Pakete des Browsers, insbesondere die Klasse `LinkedList` aus Unterabschn. 2.2.2.1. Gemäß der Paketstruktur von Abb. 2.15 befindet sich die Klasse im Paket `browse.util`. Der Programmtext von Abbildung 2.16 zeigt eine sinnvolle Regelung des Zugriffs auf die beteiligten Programmelemente. Da die Klasse `LinkedList` gerade für die Benutzung in anderen Paketen bestimmt ist, müssen die Klasse selbst, der Konstruktor, die wesentlichen Methoden und die innere Klasse `ListIterator` öffentlich sein. Aus den in Abschn. 2.2.1 genannten Gründen ist es sinnvoll, die Attribute zur Repräsentation der Listenstruktur sowie die dafür benötigte innere Klassendeklaration `Entry` vollständig zu kapseln. Damit sind sie auch vor dem Zugriff von anderen Klassen des Pakets `browse.util` geschützt.

¹² In Kap. 3 werden wir noch den Modifikator `protected` kennen lernen, der zusätzlich zur paketlokalen Benutzung die Benutzung in Unterklassen zulässt.

```
package browse.util;

import browse.ListElem;
import java.util.NoSuchElementException;

public class LinkedList {
    private Entry header = new Entry(null, null, null);
    private int size = 0;

    public LinkedList() { ... }
    public ListElem getLast() { ... }
    public ListElem removeLast() { ... }
    public void addLast(ListElem e) { ... }
    public int size() { ... }

    private static class Entry { ... }

    public class ListIterator {
        private int nextIndex = 0;
        private Entry next = header.next;

        public boolean hasNext() { ... }
        public ListElem next() { ... }
    }

    public ListIterator listIterator() {
        return new ListIterator();
    }
}
```

Abb. 2.16. Kapselung der Klasse `LinkedList` und ihrer inneren Klassen

Typische Beispiele für paketlokale Programmelemente sind die Klassen `TextFenster` und `W3Seite`. Sie müssen für die Browser-Klasse zugreifbar sein. Ein Zugriff von außerhalb des Pakets ist aber weder nötig noch erwünscht. Deshalb sollten sie – wie in Abb. 2.15 gezeigt – als paketlokal vereinbart werden, d.h. ohne Zugriffsmodifikator.

Durch die Auszeichnung der öffentlichen Programmelemente und die Kapselung aller anderen Elemente wird für jedes Paket eine Benutzungsschnittstelle definiert, die aus den öffentlichen Typen und deren öffentlichen Methoden, Konstruktoren und Attributen besteht. Eine Verfeinerung dieser Schnittstellenbildung und der zugehörigen Kapselungsaspekte werden wir im Zusammenhang mit der Vererbung in Kap. 3 behandeln.

2.2.3 Beziehungen zwischen Klassen

In den letzten beiden Abschnitten haben wir gesehen, wie man eine Menge von Klassen strukturieren kann, indem man sie ineinander schachtelt bzw. mehrere Klassen zu Paketen zusammenfasst. In diesem Abschnitt resümieren wir diese Strukturierungsbeziehungen zwischen Klassen und betrachten dann die Beziehungen zwischen Klassen bzw. ihren Objekten von einem allgemeineren Standpunkt.

Eine gute Strukturierung von Klassen richtet sich natürlich nach den Beziehungen, die die Klassen zueinander haben. Typischerweise wird man innere Klassen verwenden, wenn man für die Implementierung einer Klasse Hilfsklassen benötigt, die von außerhalb nicht sichtbar sein sollen. Eine andere typische Anwendung innerer Klassen besteht darin, Objekte bereitzustellen, mit denen man einen komplexeren, aber gekapselten Zugang zu einem Objekt oder einem Objektgeflecht ermöglichen möchte. Als Beispiel dazu haben wir Iteratoren betrachtet, die es gestatten eine Listenstruktur schrittweise zu durchlaufen, ohne dem Benutzer allgemeinen Zugriff auf die Listenstruktur zu gewähren. Eine Menge von Klassen wird man zu einem Paket zusammenfassen, wenn diese Klassen inhaltlich bzw. anwendungsspezifisch verwandt sind oder einen hinreichend gut abgrenzbaren Teil eines größeren Anwendungsprogramms darstellen.

Wie wir gesehen haben, ist eine gute Strukturierung von Klassen abhängig von den Beziehungen, die zwischen den Klassen und ihren Objekten existieren. Die Festlegung der Klassen und ihrer Beziehungen ist eine der wichtigen Aufgaben des *objektorientierten Entwurfs*. Etwas genauer betrachtet geht es dabei darum,

1. die relevanten Objekte des Aufgabenbereichs zu identifizieren, geeignet zu klassifizieren und von allen unnötigen Details zu befreien und
2. ihre Beziehungen so zu entwerfen, dass sich die Dynamik des modellierten Aufgabenbereichs möglichst intuitiv und leicht realisierbar beschreiben lässt.

Der erste der beiden Schritte wird dabei häufig als *Auffinden der Schlüsselabstraktionen* bezeichnet.

Beide Schritte bilden die Grundlage für den Klassenentwurf und haben dementsprechend gewichtigen Einfluss auf die objektorientierte Programmierung. Insbesondere ist es wichtig, sich die Beziehungen zwischen Klassen bzw. ihren Objekten bewusst zu machen, da sie die Strukturierbarkeit, Lesbarkeit und Wartbarkeit von Programmen wesentlich beeinflussen können. Betrachten wir zunächst einmal nur die Beziehungen auf Objektebene. Die einfachste Beziehung zwischen Objekten besteht darin, dass ein Objekt ein anderes Objekt benutzt. Beispielsweise benutzt ein Browser-Objekt einen W3-Server. Man spricht von einer *Benutzungsbeziehung* oder engl. *uses relation*. Wie im Browser-Beispiel kann die Benutzungsbeziehung so angelegt sein, dass ein

Objekt von mehreren Objekten benutzt wird. Grundsätzlich kann man sagen, dass ein Objekt X ein anderes Objekt Y benutzt, wenn

- eine Methode von X eine Nachricht an Y schickt oder
- eine Methode von X das Objekt Y erzeugt, als Parameter übergeben bekommt oder als Ergebnis zurückgibt.

In vielen Fällen besitzt ein Objekt X , das ein anderes Objekt Y benutzt, eine Referenz auf dieses Objekt. Implementierungstechnisch heißt das für Java, dass X ein Attribut *hat*, das eine Referenz auf Y enthält. Man sagt deshalb auch oft, dass die Objekte X und Y in der *hat-ein-Beziehung* oder englisch *has-a relation* stehen. Eine spezielle Form der hat-ein-Beziehung ist die *Teil-von-Beziehung* oder engl. *part-of relation*. In diesem Fall betrachtet man das referenzierte Objekt als (physikalischen) Teil des referenzierenden Objekts, beispielsweise so, wie ein Motor Teil genau eines Autos ist. Insbesondere kann ein Objekt nicht direkter Teil zweier unterschiedlicher Objekte sein. Da man die Teil-von-Beziehung in Java nur durch Referenzen realisieren kann, muss der Programmierer diese Eigenschaft gewährleisten. In Sprachen wie C++ und BETA ist es möglich, diese Beziehung auch ohne den Umweg über Referenzen zu realisieren.

Eine Verallgemeinerung der hat-ein-Beziehung ist die *Erreichbarkeitsbeziehung*: Ein Objekt X *erreicht* ein Objekt Y , wenn man durch wiederholtes Selektieren eines Attributs von X nach Y kommen kann. Die Erreichbarkeitsbeziehung ist wichtig, um abzuschätzen, welche Fernwirkungen die Modifikation eines Objekts haben kann: Wird ein Objekt Y verändert, kann sich das Verhalten aller Objekte ändern, die Y erreichen (beispielsweise, weil sie bei einem Methodenaufruf den Zustand von Y lesen). Andersherum gilt, dass ein Methodenaufruf maximal diejenigen Objekte verändern kann, die vom impliziten und von den expliziten Parametern sowie den zugreifbaren Klassenattributen aus erreichbar sind.

Viele Aspekte der Beziehungen zwischen Objekten lassen sich auf ihre Klassen übertragen. So sagen wir, dass eine Klasse A eine andere Klasse B *benutzt*, wenn eine Methode von A eine Nachricht an B -Objekte schickt oder wenn eine Methode von A B -Objekte erzeugt, als Parameter übergeben bekommt oder als Ergebnis zurückgibt. Entsprechend kann man die hat-ein- und die Erreichbarkeitsbeziehung auf Klassen definieren. Ein guter Klassendesign zeichnet sich dadurch aus, dass die Anzahl der Benutzungsbeziehungen gering und überschaubar gehalten wird und weitgehend Gebrauch von Kapselungstechniken gemacht wird, um die Benutzungsschnittstellen sauber herauszuarbeiten.

Neben der Benutzungs- und hat-ein-Beziehung spielt in der objektorientierten Programmierung die *ist-ein-Beziehung* zwischen Objekten und die Subtypbeziehung auf den Typen eine zentrale Rolle. Diese werden wir im folgenden Kap. 3 kennen lernen.

Konzepte objektorientierter Programmierung

Mit einer Einführung in Java

Poetzsch-Heffter, A.

2009, XIV, 352 S. 107 Abb., Softcover

ISBN: 978-3-540-89470-4