1 Container (Collections)

Ein zentrales Thema in der modernen Informatik sind die "Datenstrukturen und Algorithmen". Hierbei geht es um das fundamentale Problem, mit dem sich fast jedes ernsthafte Programm konfrontiert sieht: Wie sollen die Daten abgelegt werden, so dass Operationen darauf effizient ablaufen können? Unter Effizienz versteht man dabei meistens den Begriff Schnelligkeit, aber oft genug ist es auch der Platzverbrauch, der so gering wie möglich bleiben soll. Am liebsten hätte man natürlich gerne minimalen Speicherverbrauch bei maximaler Geschwindigkeit – ein unerreichbares Ziel und in der Praxis muss man sich als Programmierer auf Kompromisse einlassen.

Nun kann im Rahmen dieses Buches keine umfassende Behandlung von Datenstrukturen und Algorithmen angeboten werden. Selbst ein zaghafter Versuch würde dieses Kapitel schnell auf mehrere hundert Seiten explodieren lassen. Aber glücklicherweise lässt sich eine brauchbare Grundlage schon mit wenigen Zutaten sprich Klassen verwirklichen, die sich im Paket `java.util` bereit halten.

Ein Java-Programm erzeugt und verarbeitet während seiner Laufzeit Daten in Form von Variablen. Teilweise handelt es sich dabei um primitive Datentypen wie `int`, `char` usw., zum größeren Teil handelt es sich um Objekte. Bei einer sehr überschaubaren Menge von Daten kann man als Programmierer diese Daten in entsprechenden Variablen ablegen, z.B.

```
int a;
int b;
String name;
String vorname;
```

Wenn die Anzahl der Daten jedoch wächst, wird dieser Ansatz unpraktikabel. Wenn beispielsweise Hunderte von Namen verwaltet werden müssen, kann man nicht einhundert Variablen `name_1`, `name_2`, ..., `name_100` anlegen und einen maßgeschneiderten Code zum Umgang mit diesen Variablen fest kodieren. Die Lösung bringt der Einsatz eines Datenbehälters, in den man beliebige Daten in Form von Objekten einfügen und bei Bedarf über geeignete Operationen wieder darauf zugreifen kann. Einen solchen Behälter nennt man bei Programmiersprachen allgemein *Container* und bei Java im besonderen *Collection* (Sammlung)[1].

[1] Da es in Java auch eine Schnittstelle `Collection` und eine Klasse `Collections` gibt, wird in diesem Kapitel der allgemeinere Begriff

Heerscharen von Forschern haben sich in den letzten Jahrzehnten Gedanken darüber gemacht, wie ein Container intern die ihm anvertrauten Daten ablegen soll und welche Operationen auf diesen Daten für den Nutzer zur Verfügung gestellt werden sollen. Das Ergebnis dieser Arbeiten können wir grob in drei Arten unterteilen:

- Eine Liste (= List) kann Elemente mehrfach enthalten und verwaltet eine Ordnung (Reihenfolge) der Elemente.
- Eine Menge (= Set) enthält jedes Element nur einmal und es gibt keine Ordnung auf den Elementen.
- Eine Wörterbuch/Hashtabelle (= Map) ist eine Menge von Schlüssel-Wert-Paaren.

Container sind in Java von einer allgemeinen Basisschnittstelle `java.util.Collection` abgeleitet. Die Spezialisierung auf die oben skizzierten Grundtypen Liste, Menge und Wörterbuch erfolgt durch zusätzliche Schnittstellen `java.util.List`, `java.util.Set` und `java.util.Map`. Davon abgeleitet sind verschiedene Klassen, die unterschiedliche interne Implementierungen dieser Schnittstellen anbieten, um besondere Anforderungen zu erfüllen.

Hinzu kommt noch die Schnittstelle `java.util.Iterator`, die definiert, wie bequem alle Elemente eines Containers besucht werden können, sowie die Schnittstellen `java.lang.Comparable` und `java.util.Comparator` die festlegen, wie Elemente miteinander verglichen werden sollen.

In Java kann ein Container immer nur Objektreferenzen vom Basistyp `Object` aufnehmen. Da alle Klassen von `Object` abgeleitet sind, bedeutet dies, dass man jede beliebige Klasseninstanz in einen Container einfügen kann. Allerdings muss man bei Ausleseoperationen einen entsprechenden Cast in die gewünschte Klasse durchführen, da aus dem Container nur Referenzen vom Typ `Object` geliefert werden. Dies ist der wesentliche Unterschied zu Containern in anderen Programmiersprachen wie C++. Dort ist es möglich, für einen Container festzulegen, welche Art von Objekten abgelegt werden dürfen, so dass Einfügeoperationen mit einem anderen Objekttyp zu einem Compiler-Fehler führen. In Java wird dies erst mit der nächsten Version J2SE 1.5 (Stichwort "*Java Generics*") möglich sein. In Kapitel 33 wird in einer kurzen Preview darauf eingegangen.

Container verwendet. Das macht es auch für Sie leichter, wenn Sie sich mit C++-Programmierern austauschen :-)

Icon INFO

Da alle Collection-Container nur auf Objekten operieren können, ist die direkte Verwaltung von einfachen Datentypen wie `int` oder `float` nicht möglich. Man muss dann mit Hilfe der entsprechenden Wrapper-Klassen `Integer`, `Float`, etc. erst ein Stellvertreter-Objekt kreieren (z.B. `new Integer(34)`), das dann als Element in einen Container eingefügt werden kann.

In zukünftigen Versionen von Java (Version 1.5 oder 1.6) ist geplant, für solche Fälle *Autoboxing* einzuführen. Hierbei übernimmt der Compiler das lästige Wrappen und man kann dann auch einfache Datentypen direkt übergeben.

Ein weiteres wichtiges Merkmal von Containern ist ihre Fähigkeit, beliebig viele Elemente aufzunehmen. Ein Container hat zwar immer eine Standardgröße beim Erzeugen, aber sie kann bei Bedarf vergrößert werden, wenn mehr Elemente eingefügt werden sollen als Platz vorhanden ist. Die Vergrößerung verläuft dabei für den Anwender völlig unsichtbar.

1.1 Listen

Der meistgenutzte Container ist sicherlich die Liste. In eine Liste kann man Elemente aufnehmen und dabei die Reihenfolge der Elemente festlegen. In einer Liste gibt es daher immer ein Anfangselement (der Listenkopf, Head) und ein Endelement (das Listenende, auch Tail genannt). Da es eine Reihenfolge der Elemente gibt, ist es außerdem möglich, auf einzelne Elemente über ihre Position in der Liste zuzugreifen.

Eine besonders einfache Spielart der Liste verwendet jeder Programmier fast automatisch, ohne sich dessen richtig bewusst zu sein: das Array (siehe Kapitel 9)! Es erfüllt alle oben aufgeführten Merkmale und hat lediglich zwei Eigenschaften, die ein "normaler" Java-Container nicht hat: das Array hat eine feste Größe, die nicht mehr geändert werden kann, und alle Elemente haben den gleichen Datentyp, der zudem auch ein primitiver Datentyp sein darf. Für viele Zwecke ist der Einsatz eines Arrays die einfachste und auch schnellste Variante. Aber natürlich gibt es auch Nachteile, z.B. kann man aus einem Array kein Element löschen[2] und eine nachträgliche Vergrößerung ist auch nicht möglich. Glücklicherweise finden sich im Paket `java.util` eine ganze Reihe von Klassen, die eine

[2] Man könnte allenfalls als Hilfskrücke einen „Gelöscht“-Wert einer Array-Position zuweisen, die gelöscht sein soll.

Liste realisieren. Sie sind alle von der allgemeinen `List`-Schnittstelle abgeleitet.

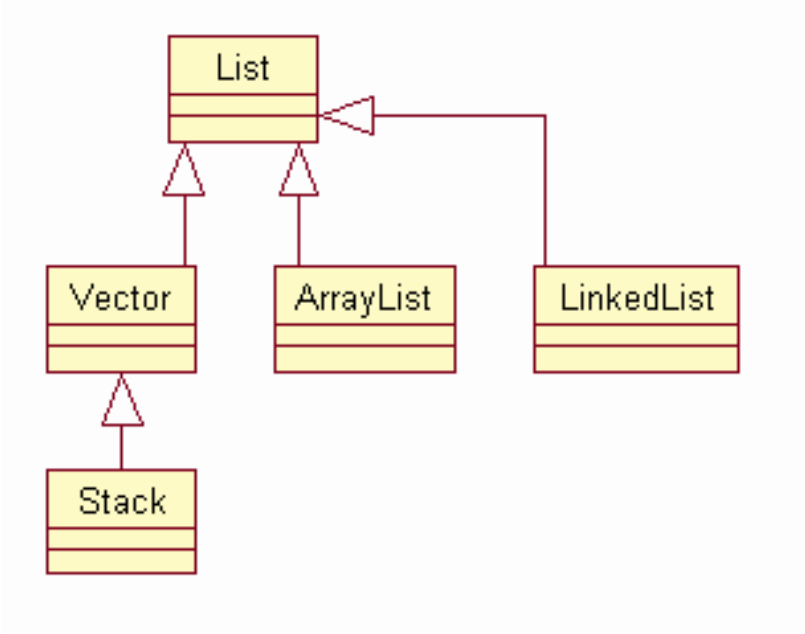

Abbildung 1.1:Vereinfachtes Klassendiagramm für Listen

Die `List`-Schnittstelle definiert eine Vielzahl von Methoden, mit deren Hilfe auf Daten in einer Liste zugegriffen werden kann bzw. Daten eingefügt werden können. Tabelle 1.1 zeigt eine Auswahl.

Methode	**Beschreibung**
`boolean add(Object obj)`	Fügt das Element `obj` an das Ende der Liste hinzu; Rückgabe von `true` bei Erfolg.
`void add(int pos, Object obj)`	Fügt das Element `obj` beim Index `pos` ein. Verschiebt das an dieser Position vorhandene Element und alle nachfolgenden um eins nach hinten.
`Object remove(int pos)`	Löscht das Element an Position `pos` und liefert es zurück.
`boolean remove(Object obj)`	Sucht und löscht das erste Auftreten von `obj` und gibt im Erfolgsfall `true`

	zurück.
`boolean contains(Object obj)`	Durchsucht die Liste nach `obj` und liefert im Erfolgsfall `true` zurück.
`Object get(int pos)`	Liefert das Element an Position `pos`.
`int indexOf(Object obj)`	Durchsucht die Liste nach `obj` und liefert im Erfolgsfall die Position zurück oder –1, wenn nicht gefunden.
`int lastIndexOf(Object obj)`	Durchsucht die Liste nach `obj` und liefert im Erfolgsfall die letzte Position zurück oder –1, wenn nicht gefunden.
`boolean isEmpty()`	Liefert `true`, wenn die Liste leer ist.
`int size()`	Liefert die Anzahl der Elemente in der Liste.
`Iterator iterator()` `ListIterator listIterator()`	Liefert ein `Iterator`- bzw. `ListIterator`-Objekt zum Durchlaufen der Liste[3].
`Object[] toArray()`	Liefert ein Array, gefüllt mit den Elementen gemäß ihrer Reihenfolge in der Liste.

Tabelle 1.1: Wichtige Methoden der Schnittstelle List

Die Klassen `Vector`, `ArrayList` und `LinkedList` implementieren diese Methoden und bieten hinsichtlich der Funktionalität keine wesentlichen Unterschiede, abgesehen von einigen Zugriffsmethoden (z.B. bietet `LinkedList` die Methoden `getFirst()`, `getLast()` an). Aber warum gibt es dann mehrere Klassen für Listen?

Die Antwort liegt im Laufzeitverhalten. Welche Klasse angebracht ist, hängt davon ab, welche Zugriffsoperationen überwiegend auf der Liste durchgeführt werden sollen:

- `Vector` und `ArrayList` sind in ihrer internen Implementierung einem normalen Array ähnlich und daher gut geeignet für das sequentielle Durchlaufen oder den direkten Zugriff auf ein Element, dessen Position innerhalb der Liste bekannt ist. Nachteil: Einfügen und Löschen innerhalb der Liste (nicht am Ende) sind sehr langsam, weil intern die Datenstruktur umgebaut werden muss. `Vector` ist übrigens

[3] Mehr dazu im Abschnitt „Iteratoren“.

eine `sychronized`-Version und historisch gesehen älter als `ArrayList`, das in der Regel vorzuziehen ist.

- `LinkedList` realisiert eine verkettete Liste: jedes Element hat einen Zeiger auf seinen Vorgänger und seinen Nachfolger. Das sequentielle Durchlaufen oder der direkte Zugriff ist daher sehr langsam, weil immer am Anfang oder Ende der Liste begonnen werden muss und man sich von einem Element zum nächsten hangelt. Dafür sind Einfüge- und Löschoperationen recht effizient.

Eine besondere Form der Liste ist ein Stack, auch als FILO (First in, last out) Datenstruktur oder Keller bezeichnet. Der Begriff „Stack“ bedeutet Stapel und veranschaulicht sehr schön die Funktionsweise: wie bei einem Tellerstapel ist das zuerst eingefügte Element ganz unten, das zuletzt eingefügte Element ganz oben und ein Zugriff auf den Stack erfolgt immer nur am Ende, also oben: man kann noch ein Element hinzufügen oder das oberste Element entfernen. Eine Implementierung stellt die Klasse `Stack` bereit, die von `Vector` abgeleitet ist[4]. Die zusätzlichen Stack-typischen Methoden sind in Tabelle 1.2 aufgeführt.

Methode	Beschreibung
`Object peek()`	Liefert das oberste Element, ohne es zu entfernen.
`Object pop()`	Liefert das oberste Element und entfernt es vom Stack.
`Object push(Object obj)`	Fügt `obj` als oberstes Element in den Stack ein. Rückgabewert ist `obj`.

Tabelle 1.2: Methoden der Klasse Stack

Während Listen recht intuitiv sind, ist ein Stack doch für viele Programmierer etwas ungewohnt. dabei kann er recht nützlich sein. Aus diesem Grund soll hier der Einsatz von Listen am Sonderfall Stack demonstriert werden.

Stacks werden häufig beim Parsen von Ausdrücken eingesetzt. Im folgenden Beispiel wird ein arithmetischer Ausdruck aus der üblichen Infix-Notation, z.B.

```
2 * (3 + 4 * 2 + 5)
```

[4] Übrigens ein Paradebeispiel für den falschen Einsatz von Vererbung. `Stack` besitzt dadurch auch Zugriffsmethoden auf Elemente, die nicht oben sind, was einem Stack völlig widerspricht.

in die Postfix-Notation umgewandelt:

```
2 3 4 2 * + 5 + *
```

Bei der Postfix-Notation kommen zuerst die Operanden, gefolgt vom Operator, während bei der Infix-Notation der Operator zwischen den Operanden steht. Postfix hat den Vorteil, dass man weder Prioritäten für Operatoren noch Klammern benötigt und ist daher für die Verarbeitung durch ein Programm viel leichter zu handhaben als die Infix-Variante.

Die Umwandlung kann mit Hilfe eines Stacks durchgeführt werden. Der Infix-Ausdruck wird nach den folgenden Regeln abgearbeitet:

- Jede Zahl wird sofort dem Postfix-Ausdruck hinzugefügt
- Eine öffnende Klammer oder ein Operator wird auf den Stack gelegt, es sei denn, auf dem Stack befindet sich oben ein Operator, der eine niedrigere Priorität hat; dann werden alle Operatoren vom Stack entfernt und zur Postfix-Darstellung hinzugefügt, bis der aktuelle Operator eine höhere Priorität hat als der noch befindliche Operator auf dem Stack.
- Wenn eine schließende Klammer gefunden wird, werden alle Operatoren vom Stack entfernt und zur Postfix-Darstellung hinzugefügt, bis eine öffnende Klammer auf dem Stack erreicht worden ist.

Listing 1.1: StackDemo.java – Umwandlung in Postfix-Notation mit einem Stack

```
01 import java.util.*;
02 import java.io.*;
03
04 // Hilfsklasse
05 class Item {
06    // Konstruktor
07    Item(char z, int w) {
08       zeichen = z;
09       prio = w;
10    }
11
12   char zeichen;
13   int prio;
14 }
15
16 public class StackDemo {
17    public static void main(String[] args) {
18
19    while(true) {
20       // Einen Ausdruck von Tastatur einlesen
21       ...
22
```

```
23       // nach jedem Zeichen ein Leerzeichen einfügen für
24       // den StringTokenizer
25       ...
26
27       // in einzelne Zeichen zerlegen
28       StringTokenizer st = new StringTokenizer(ausdruck);
29
30       // Ausdruck in Postfix umwandeln
31       Stack operatoren = new Stack();
32       StringBuffer postFix = new StringBuffer();
33
34       char z;
35       String str;
36
37       while(st.hasMoreTokens()) {
38          str = st.nextToken();
39          z = str.charAt(0);
40
41          int prioAlt; // die Priorität des im vorigen Schritt
42                       // gelesenen Operators
43
44          if(operatoren.size() > 0) {
45             Item op = (Item) operatoren.peek();
46             prioAlt = op.prio;
47          } else
48             prioAlt = 0;
49
50          Item aktItem;  // frisch gelesene Operator/Zahl
51
52          switch(z) {
            case 'q': System.exit(0);
53          case '(': aktItem = new Item(z,1);
54                    break;
55          case '+':
56          case '-': aktItem = new Item(z,2);
57                    break;
58          case '*':
59          case '/':
60          case ':': aktItem = new Item(z,3);
61                    break;
62          case ')': aktItem = new Item(z,4);
63                    break;
64          default : // kein Operator, sondern eine Zahl
65                    aktItem = new Item(z,0);
66                    break;
67          }
68
69          if(aktItem.prio == 0) {
70             // eine Zahl direkt der Postfix-Darstellung hinzufügen
71             postFix.append(aktItem.zeichen);
72             postFix.append(" ");
73             continue;
74          }
75
76          if(aktItem.prio == 1) {
77             // öffnende Klammer auf den Stack
```

```
78            operatoren.push(aktItem);
79            continue;
80         }
81
82         if(aktItem.prio == 4) {
83            // schließende Klammer gefunden
84            // solange Operatoren entfernen u. zur Postfixdarstellung
85            // hinzufügen, bis eine öffnende Klammer gefunden wird
86            while(true) {
87               Item op = (Item) operatoren.pop();
88
89               if(op.zeichen != '(')   {
90                  postFix.append(op.zeichen);
91                  postFix.append(" ");
92               } else
93                  break;
94            }
95            continue;
96         }
97
98         if(aktItem.prio > prioAlt)  {
99            // der neu gelesene Operator hat eine höher Priorität
100            // als der vorige auf dem Stack befindliche
101            operatoren.push(aktItem);
102         } else {
103            // alle vorhandenen Operatoren auf dem Stack
104            // zur Postfix-Darstellung hinzufügen, deren
105            // Priorität >=  dem gerade gelesenen Operator
106            while(true)  {
107               if(operatoren.size() > 0) {
108                  Item op = (Item) operatoren.peek();
109
110                  if(op.prio >= aktItem.prio)  {
111                     operatoren.pop();
112                     postFix.append(op.zeichen);
113                     postFix.append(" ");
114                  } else
115                     break;
116               }
117            }
118
119            operatoren.push(aktItem); // zuletzt gelesener Op.
                                         // auf Stack
120         }
121      }
122
123      // Ausdruck komplett gelesen; den Reststack leeren
124      while(operatoren.size() > 0) {
125         Item op = (Item) operatoren.pop();
126         postFix.append(op.zeichen);
127         postFix.append(" ");
128      }
129
130      System.out.println("Postfix: " + postFix.toString());
131   }
```

```
132     }
133 }
```

Das Listing zeigt nur den interessanten Ausschnitt des Programms (der vollständige Code befindet sich auf der Buch-CD als *StackDemo.java* im Programmverzeichnis zu diesem Kapitel).

Es geht los in den Zeilen 31/32, wo ein `Stack` zum Zwischenspeichern von Operatoren angelegt wird sowie ein `StringBuffer` für die zu erstellende Postfix-Darstellung. Dann beginnt in einer Schleife das Abarbeiten der Tokens aus dem übergebenen Ausdruck. In Zeile 45 wird zunächst nachgeschaut, ob etwas auf dem Stack liegt, und dessen Priorität festgehalten. Dann wird dem aktuellen Token eine bestimmte Priorität zugewiesen (Zeilen 52-67). Für normale Zahlen die geringste Priorität, für die Operatoren eine höhere gemäß der Regel „Punkt- vor Strichrechnung“.

Damit ist die Priorität des gerade gelesenen Tokens sowie die Priorität des auf dem Stack befindlichen obersten Tokens (ein Operator oder eine öffnende Klammer) bekannt, so dass nun in den Zeilen 69-121 nach den oben beschriebenen Regeln vorgegangen werden kann, d.h., das aktuelle Token wird direkt zur Postfix-Darstellung hinzugefügt, auf dem Stack zwischengespeichert oder Operatoren vom Stack heruntergenommen und zur Postfix-Darstellung hinzugefügt.

Falls Sie Lust haben, können Sie sich zur Übung nun noch überlegen, wie man die erhaltene Postfix-Darstellung mit Hilfe eines Stacks dazu verwenden kann, den Wert des Ausdrucks auszurechnen!

1.2 Mengen

Eine Menge ist eng verwandt mit einer Liste. Der wesentliche Unterschied ist die fehlende Ordnung sowie der Umstand, dass ein Element nur einmal in einer Menge auftauchen kann. In der Praxis ist allerdings die nicht vorhandene Ordnung nicht ganz so streng zu sehen, da es entsprechende Mengen-Klassen gibt, die dennoch eine Ordnung anbieten:

- `HashSet`: die Standardklasse für Mengen. `null` kann als Element eingefügt werden. Es gibt keine Ordnung auf den Elementen.
- `TreeSet`: wie `HashSet`, allerdings sind die Elemente aufsteigend geordnet. Einzufügende Elemente müssen die `Comparable`-Schnittstelle implementieren[5].

[5] Mehr dazu im Abschnitt „Suchen und Sortieren“.

- `LinkedHashSet`: wie `HashSet`, aber die Elemente sind nach der Einfüge-Reihenfolge geordnet.

Lassen Sie sich von den Klassennamen nicht unnötig verwirren. Die Java-Entwickler haben die etwas fragwürdige Entscheidung getroffen, die Klassennamen in Anlehnung an die zugrunde liegende Implementierung zu wählen und nicht nach ihrer Funktionalität (`TreeSet` ist so genannt, weil intern eine besondere Baumstruktur, so genannte Rot-Schwarz-Bäume, verwendet wird. Analog wird bei `HashSet` eine Hashtabelle eingesetzt).

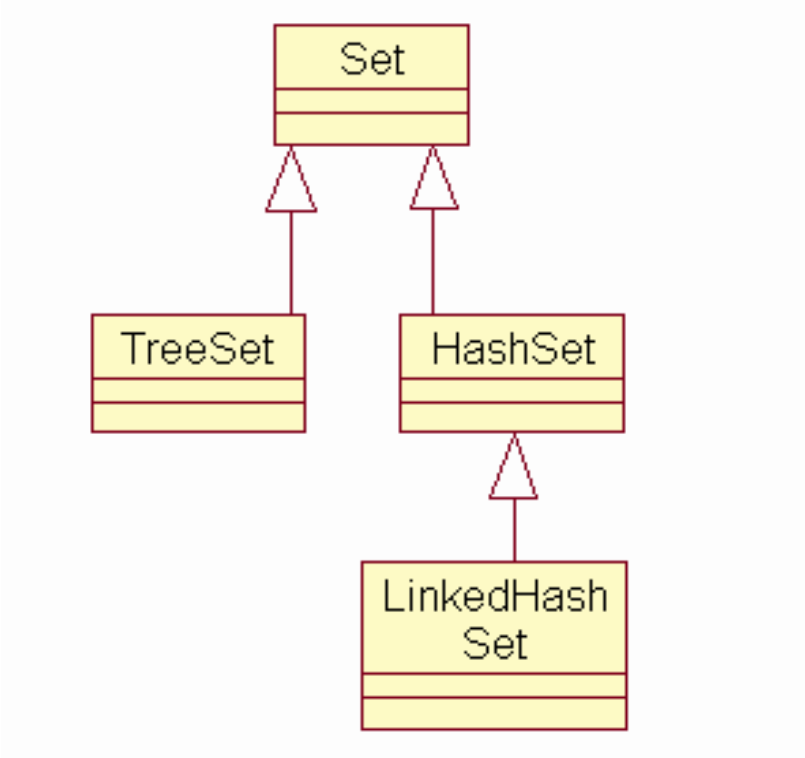

Abbildung 1.2: Vereinfachtes Klassendiagramm für Mengen

Methode	**Beschreibung**
`boolean add(Object obj)`	Fügt das Element `obj` in die Menge ein und gibt bei Erfolg `true` zurück; `false` bedeutet, dass `obj` schon vorhanden ist.

`void clear()`	Löscht alle Einträge.
`boolean remove(Object obj)`	Löscht das Element `obj` und liefert bei Erfolg `true` zurück.
`boolean contains(Object obj)`	`true`, wenn es das Element `obj` gibt.
`int size()`	Liefert die Anzahl an gespeicherten Werten.
`Object[] toArray()`	Liefert ein Array mit den Elementen der Menge.
`Iterator iterator()`	Liefert einen Iterator zum Durchlaufen der Elemente.

Tabelle 1.3: Wichtige Methoden der Schnittstelle Set

Im folgenden Beispiel sehen Sie einen exemplarischen Einsatz von Mengen in Form von `TreeSet` zur Ermittlung der Lottozahlen[6]. Die gezogenen Zahlen dürfen nur einmal vorkommen und es bietet sich daher die Modellierung als Menge an.

Listing 1.2: Lotto.java – Lottozahlen mit Hilfe von TreeSet verwalten

```
01 import java.util.*;
02
03 public class Lotto {
04    public static void main(String[] args) {
05       System.out.println("Willkommen zur Ziehung der Lottozahlen!");
06
07       TreeSet zahlen = new TreeSet();
08       Random zufall = new Random();
09
10       while(zahlen.size() != 6) {
11          int z = zufall.nextInt(50);
12
13          if(z == 0)
14             continue;
15
16          // Zahl einfügen falls nicht schon vorhanden
17          Integer aktuell = new Integer(z);
18          zahlen.add(aktuell);
19          }
20
21       // Ausgabe in aufsteigender Reihenfolge
22       Iterator it = zahlen.iterator();
23
24       while(it.hasNext()) {
```

[6] Bei Gewinn bitte an die Autoren denken!

```
25            System.out.println(it.next());
26        }
27    }
28 }
```

Das Programm zeigt hoffentlich auf, wie schnell und einfach das Programmierleben sein kann, wenn handliche und mächtige Container wie `TreeSet` verfügbar sind. Nach dem Anlegen der Menge in Zeile 7 können in einer Schleife so lange Zufallszahlen erzeugt werden, bis es gelungen ist, sechs verschiedene in die Menge einzufügen. Dies geschieht in den Zeilen 17-18. Zunächst wird aus der aktuellen Zufallszahl ein `Integer`-Objekt erzeugt und versucht es einzufügen. Wenn es ein solches Element schon gibt, schlägt die `add()`-Methode fehl (und liefert `false` zurück, was hier aber nicht weiter interessiert, da die Schleife auf jeden Fall erneut durchlaufen werden muss). Da Lottozahlen meist in aufsteigender Form präsentiert werden, kam in diesem Fall `TreeSet` und nicht `HashSet` zum Einsatz. Bei `TreeSet` sind alle Elemente aufsteigend geordnet, so dass in Zeile 25 einfach ein `Iterator` zum Ausgeben der Werte verwendet werden kann (Iteratoren werden im Abschnitt 1.4 näher besprochen.)

1.3 Wörterbücher (Hashtabellen)

Eine immer wiederkehrende Aufgabe ist die Verwaltung von Paaren (A,B), bei denen man zu einem Suchkriterium A (der Schlüssel = Key) den zugeordneten Wert B ermitteln will. Dies lässt sich beispielsweise mit einer Liste erledigen: man kreiert eine geeignete Klasse mit Feldern für Schlüssel und Wert und legt Instanzen davon in der Liste ab. Zum Suchen wandert man durch die Liste und wählt dasjenige Element, das den gewünschten Schlüssel aufweist.

So weit alles klar, aber dieser Ansatz hat einen großen Nachteil: er ist sehr langsam, insbesondere bei größeren Datenmengen! Und da jedermann nach Geschwindigkeit schreit und ein Programm per se immer zu langsam ist, hat die Informatik eine phantastische Lösung für dieses Problem gefunden: Wörterbücher oder Hashtabellen (im Englischen meist als Hashtable, Dictionary oder Map bezeichnet).

Ein Wörterbuch ist ein besonderer Container zur Ablage von (Schlüssel/Wert-)Paaren. Mit Hilfe des Schlüssels kann der zugehörige Wert gefunden werden und zwar in konstanter Zeit! Das ist ein Riesenunterschied zu einer Liste, bei der die Suchzeit immer größer wird, je mehr Elemente sie enthält.

Dieses Wunder wird durch eine besondere Technik namens *Hashing* erreicht, was soviel wie Zerhacken bedeutet. Beim Hashing wird aus dem

Schlüssel (typischerweise eine Zeichenkette) nach einem besonderen Algorithmus eine Integer-Zahl – Hashwert genannt – berechnet, die als Index in eine Tabelle (z.B. in Form eines Arrays) dient. An der berechneten Position steht dann der gesuchte Wert. Der verwendete Hashing-Algorithmus („Hashfunktion“) muss dabei sicherstellen, dass verschiedene Schlüssel zu verschiedenen Zahlen und somit Positionen in der Tabelle führen, jedenfalls mit sehr hoher Wahrscheinlichkeit. Andernfalls werden verschiedene Schlüssel/Wert-Paare unter derselben Position abgelegt („Kollision“), was die Zugriffszeit verschlechtert (man muss dann erst noch alle unter dieser Position gefundenen Werte prüfen, um den wirklich gesuchten zu ermitteln). Die Wahrscheinlichkeit für Kollisionen steigt mit dem *Ladefaktor* (load factor), der angibt, wie hoch die Auslastung der Hashtabelle im Verhältnis zur reservierten Speicherkapazität ist.

Ein typischer maximaler Wert für den Ladefaktor ist 0.75 = 75 %. Wenn eine Hashtabelle eine Anfangskapazität von 1000 hat und mittlerweile sind 750 Einträge abgespeichert worden, dann ist der Ladefaktor von 75 % erreicht und die Tabelle wird vergrößert, damit die Kollisionsrate gesenkt wird und die Laufzeit für Zugriffsoperationen gering bleiben kann. Dies erfordert einiges an Umorganisation, was als Rehashing bezeichnet wird und natürlich Zeit kostet. Man sollte daher im Vorfeld überlegen, wie viele Einträge ungefähr gespeichert werden und die Startkapazität entsprechend auslegen.

In Java steht gleich eine ganze Schar von Klassen zur Auswahl, die alle die gemeinsame Schnittstelle `Map` umsetzen. Tabelle 1.4 zeigt eine Übersicht der wichtigsten Methoden zum Einfügen, Löschen und Suchen von Einträgen.

Methode	**Beschreibung**
`void clear()`	Löscht alle Einträge.
`Object remove(Object key)`	Löscht den Eintrag mit dem Schlüssel `key` und liefert den zugehörigen Wert zurück.
`Object put(Object key, Object wert)`	Fügt `wert` unter dem Schlüssel `key` ein und liefert `wert` wieder zurück; falls es schon einen Eintrag für `key` gibt, wird dieser überschrieben.
`Object get(Object key)`	Liefert den Wert für den angegebenen Schlüssel `key` oder `null`.
`boolean containsKey(Object key)`	`true`, wenn es den Schlüssel `key` gibt.
`int size()`	Liefert die Anzahl an gespeicherten Werten.

`boolean isEmpty()`	`true`, wenn das Wörterbuch leer ist.
`Set keySet()`	Liefert alle Schlüssel als Menge zurück.

Tabelle 1.4: Wichtige Methoden der Schnittstelle Map

In der Anfangszeit von Java gab es nur eine einzige Klasse für die Umsetzung der Hashing-Technik, nämlich `Hashtable`. Mittlerweile hat sich das Angebot stark vergrößert, um auf spezielle Erfordernisse einzugehen. Die neuen Klassen sind im Gegensatz zu `Hashtable` nicht mehr `synchronized` und daher nicht threadsicher.

Allen Klassen ist gemeinsam, dass das Hashing immer nur auf Objekten ausgeführt wird, und zwar sowohl für den Schlüssel als auch den zugeordneten Wert. Die wesentlichen Unterschiede zwischen den verschiedenen Klassen sind die folgenden:

- `Hashtable` ist threadsicher, erlaubt aber keine `null`-Schlüssel oder `null`-Werte.
- `HashMap` ist nicht threadsicher, erlaubt aber auch `null` als Schlüssel oder Werte
- `WeakHashMap`: wie `HashMap`, aber gespeicherte Schlüssel können vom Garbage Collector entsorgt werden, wenn im Programm selbst der Schlüssel nicht mehr referenziert wird[7].
- `LinkedHashMap`: wie `HashMap`, zusätzlich bleibt die Reihenfolge der Schlüssel nach ihrem zeitlichen Einfügen erhalten.
- `TreeMap`: wie `HashMap` mit der Eigenschaft, dass die Schlüssel aufsteigend sortiert werden. Die Schlüssel müssen die `Comparable`-Schnittstelle implementieren[8].

[7] Bei den anderen Varianten werden Referenzen, die als Schlüssel in der Hashtabelle eingetragen sind, nicht entsorgt, da sie für den Garbage Collector als aktiv gelten. Dies kann u.U. zu Memory Leaks (Speicherleck) führen, wenn Schlüssel nicht mehr verwendet werden, aber in der Hashtabelle verbleiben.

[8] Mehr dazu im Abschnitt „Suchen und Sortieren“.

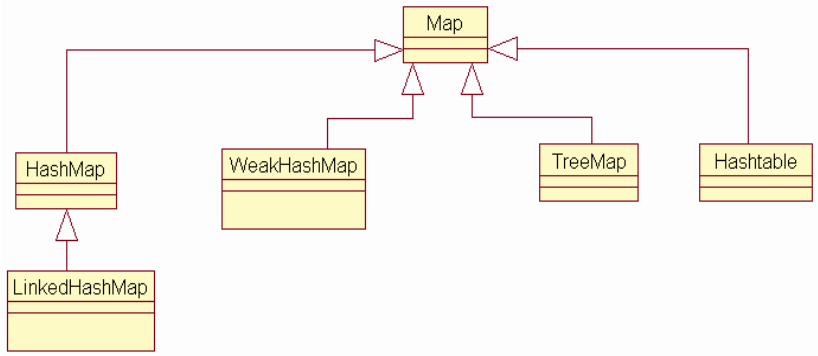

Abbildung 1.3: Vereinfachtes Klassendiagramm für Hashtabellen

Die wichtigste Klasse stellt `HashMap` dar. Sie ist für die meisten Anwendungszwecke ausreichend.

Listing 1.3: WortStatistik.java – Zählen von Worthäufigkeiten mit Hashtabellen

```
01 HashMap statistik = new HashMap(5000, 0.75F);
02
03 try {
04    BufferedReader eingabe = new
05                    BufferedReader(new FileReader(args[0]));
06
07    StreamTokenizer st = new StreamTokenizer(eingabe);
08    int typ;
09
10    while((typ = st.nextToken()) != StreamTokenizer.TT_EOF) {
11       if(st.sval == null)
12       continue;
13
14       if(statistik.containsKey(st.sval) == false)
15          statistik.put(st.sval, new Integer(1));
16       else {
17        // Wort gibt es schon -> zähler erhöhen
18          Integer wert = (Integer) statistik.get(st.sval);
19          int z = wert.intValue() + 1;
20          statistik.put(st.sval, new Integer(z));
21       }
22    }
23 }
```

In Zeile 1 wird die Hashtabelle angelegt. Neben einem parameterlosen Konstruktor mit festen Einstellungen für Startkapazität und Ladefaktor existiert für jede von `Map` abgeleitete Klasse ein zweiter Konstruktor, der Kapazität und Ladefaktor explizit erwartet. Zeilen 4-8 öffnen in vertrauter Weise die Eingabedatei und verknüpfen sie mit einem

`StreamTokenizer`, der den Gesamttext in einzelne Token (sprich die Wörter) zerlegt und zurückliefert. In der Kernschleife wird dann zunächst mit der `containsKey()`-Methode geprüft, ob es das aktuelle Wort als Schlüssel bereits gibt. Falls nein, wird das Wort selbst als Schüssel eingefügt und ihm als Wert eine 1 zugewiesen (Zeile 15). Da immer mit Objekten operiert wird, führt kein Weg an der umständlichen Variante mit dem Wrapper-Objekt vom Typ `Integer` vorbei. Falls es schon einen Eintrag zu diesem Wort gibt, wird mittels `get()` der zugehörige Zählerwert nachgeschlagen, dieser in Zeile 19 um eins erhöht und mit `put()` das aktuelle Wort und der erhöhte Zählerwert wieder eingefügt. Dabei wird der alte Eintrag überschrieben.

Wenn der Eingabestream abgearbeitet ist, befinden sich in der Hashtabelle alle gefundenen Wörter inklusive ihrer Häufigkeit. Die Ausgabe könnte dann folgendermaßen erfolgen:

```
System.out.println("Gesamte Wortanzahl: " + statistik.size());

Set wörter = statistik.keySet();
Iterator it = wörter.iterator();

while(it.hasNext()) {
   String wort  = (String) it.next();
   Integer wert = (Integer) statistik.get(wort);
   System.out.println(wort + ": " + wert.intValue());
}
```

Die Methode `keySet()` dient dazu, eine Menge aller Schlüssel der Hashtabelle zu erhalten, d.h. in unserem Fall entspricht dies genau den angetroffenen Wörtern. Mittels eines Iterators kann dann bequem die Menge durchlaufen und aus der Hashtabelle den zugehörigen Wert ausgelesen werden.

Icon NOTE

Beachten Sie, dass dieses Programm noch nicht perfekt ist, beispielsweise würde man für ernsthaftes Zählen die Eingabedatei erst noch etwas aufbereiten (z.B. Satzzeichen entfernen).

1.4 Iteratoren

Vor allem für Container des Typs Liste oder Menge will man häufig über den kompletten Inhalt iterieren, d.h. alle Elemente besuchen und auslesen oder verändern. Hierfür gibt es in modernen Programmiersprachen das Iterator-Modell.

Ein *Iterator* ist ein besonderes Objekt, das dazu dient, die Elemente eines Containers nacheinander zu durchlaufen. Man kann die Funktionsweise

eines Iterators stark vereinfacht mit einer normalen `for`-Schleife vergleichen, mit der man ein Array durchläuft: zu Beginn steht die Schleifenvariable auf der Position mit dem ersten Element, nach der Inkrementierung zeigt sie auf das nächste Element, usw. Wichtig ist der Umstand, dass man nur vom Anfang zum Ende laufen kann; hin- und herspringen oder mal vorwärts, dann rückwärts laufen ist normalerweise nicht möglich.

Auch in Java steht eine besondere Schnittstelle `java.util.Iterator` zur Verfügung mit den Methoden:

- `boolean hasNext()`: liefert `true`, wenn es noch nicht besuchte Elemente gibt
- `Object next()`: liefert das nächste Element aus dem Container
- `void remove()`: erlaubt eine Veränderung des Containers, indem das zuletzt besuchte Element gelöscht wird

Alle von `List` und `Set` abgeleiteten Klassen bieten eine besondere `iterator()`-Methode an, die ein `Iterator`-Objekt zurückliefert, das dann zum Durchlaufen des Containers verwendet werden kann. In den obigen Beispielen wurde das teilweise schon eingesetzt. Hier noch mal ein Beispiel für den Einsatz eines Iterators, wobei die besuchten Elemente aus dem Container gelöscht werden:

```
ArrayList liste = new ArrayList();

// ... Liste füllen, z.B. mit String-Objekten

// Liste auslesen und gleichzeitig löschen
Iterator it = liste.iterator();

while(it.hasNext())  {
   String str = it.next();
   System.out.println(str);
   it.remove();
}
// liste ist jetzt wieder leer
```

Zusätzlich zur `Iterator`-Schnittstelle existiert noch eine Spezialversion `ListIterator`, die von `Iterator` abgeleitet ist und wie der Name schon vermuten lässt, nur für Listen verfügbar ist. `ListIterator` bietet noch Methoden an, mit denen man die Liste in beliebiger Richtung durchlaufen und bearbeiten kann. Hierzu bieten alle von `List` abgeleiteten Klassen die Methode `listIterator()` an.

- `boolean hasPrevious()`: `true`, wenn es ein Vorgängerelement gibt

- `Object previous()`: liefert das vorangehende Elemente aus einer Liste
- `int previousIndex()`: liefert den Index des Elements, das ein nächster `previous()`-Aufruf liefern würde, oder –1, falls nicht vorhanden
- `boolean hasNext()`: `true`, wenn es einen Nachfolger gibt
- `int nextIndex()`: liefert den Index des Elements, das der nächste `next()` Aufruf liefern würde, oder –1, falls nicht vorhanden
- `void add(Object obj)`: fügt das Element `obj` in die Liste vor das Element ein, das der nächste `next()`-Aufruf liefern würde.
- `void set(Object obj)`: ersetzt das aktuell besuchte Element durch `obj`.

Die ändernden Iteratoren-Methoden `add()`, `set()` und `remove()` sind optionale Methoden. Dies bedeutet, dass ein Iterator sie zwar implementieren muss, aber eventuell nur Rumpfcode vorgibt, der nichts anderes macht als eine Exception vom Typ `UnsupportedOperationException` auszulösen.

Icon INFO

Beim Einsatz eines Iterators darf man keine Annahmen über die Reihenfolge machen, mit der die Elemente des zugrunde liegenden Containers besucht werden. Davon ausgenommen sind lediglich die Iteratoren vom Typ `ListIterator` (diese liefern die Elemente nach ihrer Position in der Liste) sowie die Iteratoren, die direkt oder indirekt von `TreeSet` und `TreeMap` erstellt worden sind (diese Container verwalten ihre Einträge in der natürlichen aufsteigenden Reihenfolge). Mehr dazu im nächsten Abschnitt.

for-Schleife der Zukunft

Die nächste Java-Version J2SE 1.5 wird eine besondere Form der `for`-Schleife unterstützen, die sich in vielen Fällen als bequeme Alternative zu Iteratoren anbietet. Das übliche Muster

```
List liste = new Liste();
...
Iterator it = liste.iterator();
while(it.hasNext())  {
   System.out.println(it.next());
}
```

vereinfacht sich dann zu:

```
for(Object o : liste)    // gelesen: "for each o in liste"
   System.out.println(o);
```

Neben der `Iterator`-Schnittstelle findet sich in der Java-Bibliothek noch eine weitere Schnittstelle `java.util.Enumeration`, die ebenfalls eine einfache Iterator-Funktionalität anbietet. Diese Schnittstelle definiert die Methoden:

```
boolean hasMoreElement()
```

```
Object nextElement()
```

zum Durchwandern der Elemente. `Enumeration` ist seit der ersten Java-Version vorhanden und wurde erst später durch das modernere `Iterator` abgelöst. Aus Kompatibilitätsgründen ist `Enumeration` jedoch weiterhin vorhanden und es gibt eine Reihe von älteren Java-Klassen, welche diese Schnittstelle implementieren oder Methoden anbieten, die ein `Enumeration`-Objekt liefern.

Für eigene Klassendefinitionen sollten Sie `Enumeration` nicht mehr verwenden und stattdessen `Iterator` einsetzen.

1.5 Suchen und Sortieren

Während die Elemente einer Liste naturgemäß in einer festen Reihenfolge vorliegen (so wie sie eingefügt worden sind), ist dies für Hashtabellen und Mengen grundsätzlich nicht der Fall. Die Ausnahmen wurden bereits in diesem Kapitel erwähnt:

- `LinkedHashSet` und `LinkedHashMap` verwalten die Elemente nach der Reihenfolge, in der sie eingefügt worden sind.
- `TreeSet` und `TreeMap` halten die eingefügten Elemente aufsteigend in ihrer natürlichen Reihenfolge geordnet.

Damit die Klassen `TreeSet` und `TreeMap` Elemente sortieren können, müssen sie natürlich wissen, wie dies geschehen soll, d.h. es wird ein Vergleichskriterium benötigt. Aus diesem Grund müssen bei diesen Klassen die eingefügten Elemente bzw. Schlüssel eine besondere Schnittstelle implementieren: `java.lang.Comparable`.

1.5.1 Die Schnittstelle Comparable

Diese Schnittstelle besteht aus einer einzigen Methode:

```
public int compareTo(Object obj)
```

und dient zum Vergleich der aktuellen Instanz mit dem Objekt `obj`. Als Rückgabewert muss ein negativer Wert für »kleiner«, 0 für »gleich« und ein positiver Wert für »größer« zurückgegeben werden.

In Kapitel 17.3.2 wurde die Schnittstelle bereits ausführlich vorgestellt. Hier soll Ihnen anhand eines einfachen Beispiels gezeigt werden, wie Sie Objekte, deren Klasse `Comparable` implementiert, in geordneter Reihenfolge in einem `TreeSet`-Container verwalten können.

Icon INFO

Die Reihenfolge der Elemente in einem Container, welche durch die `compareTo()`-Methode von `Comparable` entsteht, wird als *natürliche Reihenfolge* bezeichnet (im Unterschied zu einer über einen `Comparator` definierten, siehe unten).

Beispiel: Sie haben eine Klasse `Mitglied`, die Daten zu einem Vereinsmitglied enthält: den Namen, Vornamen sowie das Beitrittsjahr:

Listing 1.4: SortComparable.java – Verwendung der Schnittstelle Comparable mit TreeSet

```
import java.util.*;

// Klasse für einzufügende Elemente
class Mitglied implements Comparable {
   String name;
   String vorname;
   int jahr;

   // Konstruktor
   Mitglied(String n, String v, int j)  {
     name = n;
     vorname = v;
     jahr = j;
   }

   // Vergleichsmethode
   public int compareTo(Object obj)  {
     Mitglied tmp = (Mitglied) obj;

     if(jahr < tmp.jahr)
        return -1;
     if(jahr == tmp.jahr)
        return 0;
     else
        return 1;
   }
}
...
```

Diese Klasse stellt eine Implementierung von `compareTo()` bereit, so dass Instanzen in `TreeSet` eingefügt werden können und sie dort aufsteigend nach dem Beitrittsjahr sortiert vorliegen. Dies bedeutet, dass die bereits richtig sortierte Menge von einem Iterator durchlaufen werden kann:

Listing 1.5: Objekte von Mitglied in TreeSet verwalten

```
...
public class SortComparable {
  public static void main(String[] args) {
     TreeSet meinVerein = new TreeSet();

     Mitglied mg;

     mg = new Mitglied("Mueller", "Peter", 2001);
     meinVerein.add(mg);
     mg = new Mitglied("Louis", "Dirk", 2000);
     meinVerein.add(mg);
     mg = new Mitglied("Meier", "Kurt", 1998);
     meinVerein.add(mg);

     Iterator it = meinVerein.iterator();

     while(it.hasNext()) {
        mg = (Mitglied) it.next();
        System.out.println(mg.jahr + ": " + mg.name + "," + mg.vorname);
     }
  }
}
```

1.5.2 Die Schnittstelle Comparator

Neben der Schnittstelle `Comparable` existiert noch eine weitere Schnittstelle `Comparator` mit den Methoden

```
public int compare(Object o1, Object o2)
```

```
public boolean equals(Object obj)
```

`compare()` entspricht der `compareTo()`-Methode von `Comparable` und `equals()` ist die von `Object` geerbte Methode `equals()` (siehe Kapitel 17.3).

Eine wichtige Anwendungsmöglichkeit von `Comparator` liegt wieder bei den Containern mit Sortierung wie `TreeSet` und `TreeMap`. Wenn z.B. eine Reihe von `String`-Objekten eingefügt werden, dann werden sie aufgrund der `compareTo()`-Methode, welche die String-Klasse implementiert, lexikographisch aufsteigend angeordnet. Angenommen, für unsere Zwecke wäre eine absteigende Sortierung besser. Da `String` als

`final` deklariert ist, kann leider keine von `String` abgeleitete Klasse erstellt werden, welche eine neue Implementierung von `compareTo()` für absteigende Sortierung besitzt.

Hier hilft die `Comparator`-Schnittstelle weiter. Die Klassen `TreeSet` und `TreeMap` bieten neben ihren Standardkonstruktoren noch besondere Konstruktoren an, denen man eine Instanz einer Klasse mitgeben kann, die `Comparator` implementiert:

```
TreeMap(Comparator comp)
```

```
TreeSet(Comparator comp)
```

Bei Verwendung dieser Konstruktoren wird für die Sortierung während des Einfügens von Elementen die `compare()`-Methode des `Comparator`-Objekts verwendet.

Listing 1.6 zeigt ein Beispiel für die Definition einer `Comparator`-Klasse, die zur lexikographisch absteigenden Sortierung von `String`-Objekten benutzt werden kann. Das ist schnell erledigt, da bei der `Comparator`-Implementierung für `compare()` auf die `compareTo()`-Methode von `String` zurückgegriffen werden kann. Danach wird das Ergebnis negiert, so dass anstatt aufsteigend von a-z in umgekehrter Reihenfolge sortiert wird.

Listing 1.6: SortComparator.java – Beispiel für den Einsatz von Comparator

```
import java.util.*;

class StringComp implements Comparator {
   public int compare(Object o1, Object o2) {
      String str1 = (String) o1;
      String str2 = (String) o2;

      int erg =  -1 * str1.compareTo(str2);

      return erg;
   }
}


public class SortComparator {
   public static void main(String[] args) {
      StringComp comp = new StringComp();

      TreeSet menge = new TreeSet(comp);

      menge.add("Peter");
      menge.add("Adam");
      menge.add("Dirk");
```

```
        menge.add("Albert");

        Iterator it = menge.iterator();

        while(it.hasNext()) {
           String str = (String) it.next();
           System.out.println(str);
        }
     }
}
```

1.5.3 Die Klasse Collections

Ein wichtiges Hilfsmittel für den Umgang mit Containern stellt die Klasse `java.util.Collections` dar[9]. Sie bietet eine große Zahl an statischen Methoden, beispielsweise zum Suchen. Tabelle 1.5 zeigt eine kleine Auswahl.

Methode	**Beschreibung**
`static int binarySearch(List liste, Object obj)` `static int binarySearch(List liste, Object obj, Comparator comp)`	Durchsucht `liste` nach dem Element `obj` und liefert die Position des Elements oder `-1` bei Misserfolg. `liste` muss in natürlicher Reihenfolge aufsteigend sortiert sein. Alternativ kann auch ein `Comparator comp` mitgegeben werden, falls `liste` mit `comp` aufsteigend sortiert worden ist.
`static Object max(Collection coll)` `static Object max(Collection coll, Comparator comp)`	Liefert das Element mit dem maximalen Wert aus dem Container `coll` gemäß der natürlichen Reihenfolge bzw. der durch den `Comparator comp` definierten.
`static Object min(Collection coll)` `static Object min(Collection coll, Comparator comp)`	Liefert das Element mit dem minimalen Wert aus dem Container `coll` gemäß der natürlichen Reihenfolge bzw. der durch den `Comparator comp` definierten Reihenfolge.
`static void sort(List liste)` `static void sort(List liste, Comparator comp)`	Sortiert `liste` aufsteigend in natürlicher Reihenfolge bzw. gemäß der durch `comp` definierten Reihenfolge.

[9] Man beachte das »s« im Klassennamen.

`static boolean replaceAll( List liste, Object alt, Object neu)`	Ersetzt in `liste` alle Elemente, die zu `alt` identisch sind, durch das Element `neu`.

Tabelle 1.5: Wichtige Methoden der Klasse Collections

Das folgende Listing 1.7 zeigt, wie mit Hilfe der `Collections` Klasse eine Liste aufsteigend sortiert wird.

Listing 1.7: CollectionsDemo.java – Sortieren einer Liste mittels Collections.sort()

```
import java.util.*;

public class CollectionsDemo {
   public static void main(String[] args) {
      ArrayList liste = new ArrayList();

      liste.add("Hamburg");
      liste.add("Frankfurt");
      liste.add("Berlin");
      liste.add("Hannover");
      liste.add("Dortmund");

      // liste ist jetzt noch nach Einfügereihenfolge sortiert
      Collections.sort(liste);
      // liste ist jetzt alphabetisch aufsteigend

      Iterator it = liste.iterator();

      while(it.hasNext())
         System.out.println(it.next());
   }
}
```

Ausgabe:

```
Berlin
Dortmund
Frankfurt
Hamburg
Hannover
```