5 Das Collections-Framework

In diesem Kapitel stelle ich Ihnen das Collections-Framework vor, das wichtige Datenstrukturen wie Listen, Mengen und Schlüssel-Wert-Abbildungen zur Verfügung stellt. Die Lektüre dieses Kapitels soll Ihnen helfen, ein gutes Verständnis für die Arbeitsweise der genannten Datenstrukturen und mögliche Besonderheiten oder Nebenwirkungen ihres Einsatzes zu entwickeln.

Abschnitt 5.1 beschreibt in der Praxis relevante Datenstrukturen und zeigt kurze Nutzungsbeispiele. Auf gebräuchliche Anwendungsfälle wie Suchen, Sortieren und Filtern gehe ich in Abschnitt 5.2 ein. Diverse weitere nützliche Funktionalitäten werden durch die zwei Utility-Klassen `Collections` und `Arrays` im Package `java.util` bereitgestellt und in Abschnitt 5.3 beschrieben. Abschnitt 5.4 beschäftigt sich mit dem Zusammenspiel von Generics und Collections und zeigt, welche Dinge vor allem in Kombination mit Vererbung beachtet werden sollten. Abschließend werden in Abschnitt 5.5 einige Besonderheiten und Fallstricke in den Realisierungen des Collections-Frameworks dargestellt, deren Kenntnis dabei hilft, Fehler zu vermeiden.

Das Thema Datenstrukturen und Multithreading wird in diesem Kapitel nicht vertieft. Das gilt ebenso für Hinweise zur Optimierung durch die Wahl geeigneter Datenstrukturen für gewisse Einsatzkontexte. Auf Ersteres geht Abschnitt 7.6.1 ein. Letzteres wird in Kapitel 22 behandelt.

5.1 Datenstrukturen und Containerklasse

Häufig genutzte Datenstrukturen sind die eingangs erwähnten Listen, Mengen und Schlüssel-Wert-Abbildungen. Deren Realisierung erfolgt durch sogenannte ***Containerklassen***. Sie heißen so, weil sie dazu dienen, andere Klassen zu verwalten. Im Collections-Framework sind Containerklassen durch die Interfaces `List<E>`, `Set<E>` bzw. `Map<K,V>` aus dem Package `java.util` repräsentiert.

Bevor wir uns konkret mit Datenstrukturen beschäftigen, möchte ich explizit auf eine Besonderheit hinweisen. Nur Arrays können Elemente eines beliebigen Typs speichern – insbesondere können nur sie direkt primitive Typen wie `byte`, `int` oder `double` enthalten. Alle im Folgenden vorgestellten Containerklassen können lediglich Objektreferenzen speichern. Die Verwaltung primitiver Typen ist dort nur möglich, wenn diese in ein Wrapper-Objekt umgewandelt werden.

5.1.1 Wahl einer geeigneten Datenstruktur

Um Daten sinnvoll zu speichern und performant darauf zugreifen zu können, ist der Einsatz geeigneter Datenstrukturen wichtig. Das Collections-Framework stellt bereits eine qualitativ und funktional hochwertige Sammlung von Containerklassen bereit. Diese lassen sich grob in zwei Ableitungshierarchien mit den Interfaces `Collection<E>` und `Map<K,V>` als Basis unterteilen. Muss für eine gegebene Aufgabenstellung eine geeignete Datenstruktur gefunden werden, so ist zunächst basierend auf den Anforderungen und dem zu lösenden Problem die grundsätzliche Entscheidung zwischen Listen und Mengen mit den Basisinterfaces `Collection<E>` sowie Schlüssel-Wert-Abbildungen mit dem Basisinterface `Map<K,V>` zu treffen. Anschließend gilt es, eine geeignete konkrete Realisierung zu finden. Im Folgenden gebe ich einige Hinweise, wo man in der Ableitungshierarchie des Collections-Frameworks »abbiegen« sollte, wenn man auf der Suche nach einer passenden Datenstruktur ist.

Wahl einer Datenstruktur basierend auf dem Interface `Collection`

Für Sammlungen von Elementen mit dem Basistyp `E` wählt man eine Implementierung des Interface `Collection<E>` und muss sich dabei zwischen Listen und Mengen entscheiden. Für Daten, die eine Reihenfolge der Speicherung erfordern und auch (mehrfach) gleiche Einträge enthalten dürfen, setzen wir Realisierungen des Interface `List<E>` ein. Möchte man dagegen doppelte Einträge automatisch verhindern, so stellt eine Realisierung des Interface `Set<E>` die geeignete Wahl dar. Abbildung 5-1 zeigt die Typhierarchie von Listen und Mengen, wobei aus Gründen der Übersichtlichkeit die generische Definition nicht gezeigt wird.

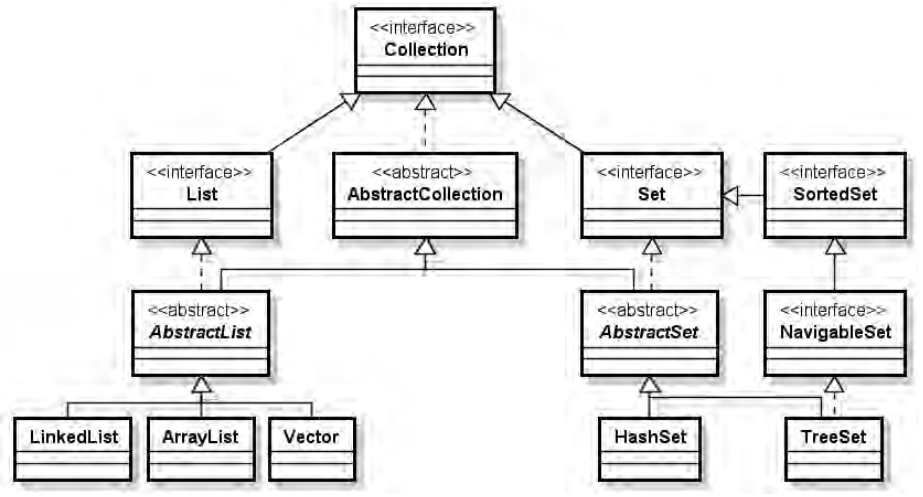


Abbildung 5-1 *`Collection`-Hierarchie*

Wahl einer Datenstruktur basierend auf dem Interface `Map`

In der Praxis findet man diverse Anwendungsfälle, in denen man Abbildungen von Objekten auf andere Objekte realisieren muss. Man spricht hier von einer Abbildung von Schlüsseln auf Werte. Dazu nutzt man sinnvollerweise das Interface `Map<K,V>`, wobei `K` dem Typ der Schlüssel und `V` demjenigen der Werte entspricht. Verschiedene Ausprägungen von Maps mit ihrer Typhierarchie, bestehend sowohl aus Klassen als auch aus weiteren, von `Map<K,V>` abgeleiteten Interfaces, zeigt Abbildung 5-2 (auch hier wieder ohne generische Typinformation).

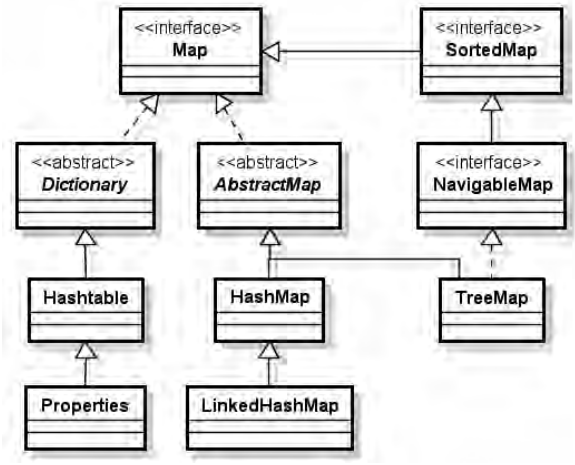


Abbildung 5-2 *`Map`-Hierarchie*

Erweiterungen in JDK 5 und 6

Mit JDK 5 und 6 wurde das Collections-Framework erweitert. Unter anderem wurden folgende Interfaces neu eingefügt:

- `Queue<E>` – Durch das Interface `Queue<E>` wird eine sogenannte Warteschlange modelliert. Diese Datenstruktur ermöglicht das Einfügen von Datensätzen am Ende und die Entnahme vom Anfang – nach dem FIFO-Prinzip (First-In-First-Out).
- `Deque<E>` – Dieses Interface definiert die Funktionalität einer doppelseitigen Queue, die Einfüge- und Löschoperationen an beiden Enden erlaubt und zudem das `Queue<E>`-Interface erfüllt.
- `NavigableSet<E>` – Dieses Interface erweitert das Interface `SortedSet<E>` um die Möglichkeit, Elemente bezüglich einer Reihenfolge zu finden. Damit ist gemeint, dass nach Elementen gesucht werden kann, die – gemäß einem Sortierkriterium – kleiner, kleiner gleich, gleich, größer gleich oder größer als übergebene Kandidaten sind. Im Speziellen können damit für bestimmte Suchbegriffe passende Elemente gefunden werden, etwa bei Eingaben in einer Combobox zur Vervollständigung.

- `NavigableMap<K,V>` – Dieses Interface erweitert die `SortedMap<K,V>`. Es gelten die für das `NavigableSet<E>` gemachten Aussagen, wobei sich die Sortierung innerhalb der Map auf die Schlüssel und nicht auf die Werte bezieht.

Mit JDK 6 wurden spezielle Implementierungen der Interfaces `SortedSet<E>` und `SortedMap<K,V>` eingeführt, die eine Sortierung mit einem hohen Grad an Parallelität kombinieren. Dies sind die Klassen `ConcurrentSkipListSet<K,V>` und `ConcurrentSkipListMap<K,V>` aus dem Package `java.util.concurrent`.

5.1.2 Arrays

Arrays sind Datenstrukturen, die in einem zusammenhängenden Speicherbereich entweder Werte eines primitiven Datentyps oder Objektreferenzen verwalten können, nachfolgend für 100 Zahlen vom Typ `int` und zwei Namen vom Typ `String` gezeigt – im letzteren Fall wird die Kurzschreibweise und syntaktische Besonderheit der direkten Initialisierung verwendet, bei der die Größe des Arrays automatisch vom Compiler durch die Anzahl der angegebenen Elemente bestimmt wird:

```
final int[] numbers = new int[100];
final String[] names1 = new String[] { "Tim", "Micha" }; // Normalschreibweise
final String[] names2 = { "Tim", "Micha" };              // Kurzschreibweise
```

Ein Array stellt lediglich einen einfachen Datenbehälter bereit, dessen Größe zur Kompilierzeit festgelegt ist und der keinerlei Containerfunktionalität bietet, d. h., es werden weder Zugriffsmethoden angeboten noch findet eine Datenkapselung statt. Diese Funktionalität muss bei Bedarf in einer nutzenden Applikation selbst programmiert werden, die wir uns an folgendem Beispiel des Einlesens von Personendaten aus einer Datenbank verdeutlichen, wobei initial Platz für 1000 Elemente bereitgestellt wird.

```
// Initiale Größenvorgabe
Person[] persons = new Person[1000];

int index = 0;
while (morePersonsAvailableInDb())
{
    if (index == persons.length)
    {
        // Größenanpassung, siehe nachfolgenden Praxistipp
        // »Anpassungen der Größe in einer Methode kapseln«
        persons = Arrays.copyOf(persons, persons.length * 2);
    }
    person[index] = readPersonFromDb();
    index++;
}
```

Eigenschaften von Arrays

Betrachten wir mögliche Auswirkungen beim Einsatz von Arrays: Vorteilhaft ist, dass indizierte Zugriffe typsicher und maximal schnell möglich sind. Auch entsteht kein

Overhead wie bei Listen, die gewisse Statusinformationen verwalten, Prüfungen vornehmen und Zugriffsmethoden auf Elemente bieten, wie es in der nachfolgenden Aufzählung angedeutet ist. Arrays eignen sich damit insbesondere dann, wenn kaum oder sogar keine Containerfunktionalität, sondern hauptsächlich ein indizierter Zugriff benötigt wird. Auch ist es nur in Arrays möglich, Werte primitiver Typen direkt zu verwalten. Auf der anderen Seite besitzen Arrays gegenüber Listen folgende Einschränkungen:

- Bei der Konstruktion eines Arrays wird zur Kompilierzeit eine fixe Größe festgelegt, die das Fassungsvermögen (auch ***Kapazität*** genannt) bestimmt. Eine sinnvolle Angabe der Kapazität ist jedoch nur dann möglich, wenn die Anzahl zu speichernder Datensätze bei der Konstruktion annähernd bekannt ist. Problematisch wird der Einsatz eines Arrays, wenn die Anzahl der zu speichernden Daten im Voraus schlecht schätzbar ist oder sich dynamisch ändern können muss, etwa bei Suchen.
- Anhand der Größe eines Arrays kann man zudem keine Aussage darüber treffen, wie viele Elemente tatsächlich gespeichert sind. Die Metainformation über den ***Füllgrad*** des Arrays, d. h. die Anzahl der dort gespeicherten Elemente, lässt sich nur aufwendig durch Iterieren über alle Einträge und durch Vergleich des gespeicherten Werts mit einem speziellen Wert, der als Indikator »kein Eintrag« dient, ermitteln. Allerdings muss auch ein solcher spezieller Wert existieren (und darf nicht Bestandteil der erlaubten Werte sein). Häufig eignet sich dazu der Wert `-1`, `0` oder `null`. Eine solche Codierung ist jedoch nicht immer möglich.
- Das Vorhalten ungenutzter Kapazität führt zu einer Verschwendung von Speicherplatz. Dies ist in der Regel für kleinere Arrays (< 1.000 Elemente) vernachlässigbar. Für große Datenstrukturen (einige 100.000 Elemente) kann sich dies aber negativ auf den belegten sowie den restlichen verfügbaren Speicher auswirken.
- Ist die gewählte Größe zu gering, so lassen sich nicht alle gewünschten Daten speichern, da keine automatische Anpassung der Größe stattfindet. Dies muss selbst programmiert werden: Im vorherigen Beispiel haben wir dazu die Methode `Arrays.copyOf()` genutzt, wodurch ein neues, größeres Array angelegt und anschließend alle Elemente des ursprünglichen Arrays in das neue kopiert werden. Dieses Vorgehen ist recht umständlich – insbesondere wenn die Größenanpassung an mehreren Stellen im Sourcecode erforderlich wird. Es bietet sich dann an, diese Funktionalität in einer Methode zu realisieren, wie dies der folgende Praxistipp »Anpassungen der Größe in einer Methode kapseln« vorstellt.
- Ein Nachbau spezifischer Containerfunktionalität ist wenig sinnvoll und erhöht die Gefahr für Probleme, etwa durch veraltete Referenzen: Das gilt, wenn einige Programmteile Referenzen auf ein Array halten und nach einer Größenänderung und einem Kopiervorgang weiterhin mit diesen arbeiten, anstatt die neue Referenz zu verwenden. Eine Lösung ist, sämtliche Zugriffe auf das Array zu kapseln. Dann beginnt man aber mit dem Nachbau einer Datenstruktur ähnlich zu der bereits existierenden `ArrayList<E>`, was aber wenig sinnvoll ist.

Wir haben nun einige Beschränkungen von Arrays kennengelernt, die besonders dann zum Tragen kommen, wenn die Zusammensetzung der Elemente einer größeren Dynamik unterliegt. Häufig lässt sich für diese Fälle durch den Einsatz von Listen oder Mengen mit dem Basisinterface `Collection<E>`, das ich im folgenden Abschnitt vorstelle, eine vereinfachte Handhabung erreichen.

Tipp: Anpassungen der Größe in einer Methode kapseln

Nehmen wir an, wir würden die Attribute `byte[] buffer` sowie `int writePos` zur Speicherung von Eingabewerten nutzen und diese über folgende Methode `storeValue(byte)` einlesen:

```
private static void storeValue(final byte byteValue)
{
    buffer[writePos] = byteValue;
    writePos++;
}
```

Werden viele Daten gespeichert, so kann die anfangs gewählte Größe nicht ausreichend sein. Es kommt dann zu einer `java.lang.ArrayIndexOutOfBoundsException`. Diese Fehlersituation lässt sich dadurch vermeiden, dass die Größe des Arrays bei Bedarf angepasst wird. Das erfordert vor dem eigentlichen Speichern eines neuen Eingabewerts eine Prüfung, ob das Ende des Arrays erreicht ist. Wird das Ende des Arrays erkannt, so muss ein größeres Array erzeugt und die zuvor gespeicherten Werte in das neue Array kopiert werden. Dazu musste man bis einschließlich JDK 5 `System.arraycopy()` wie folgt nutzen:

```
final byte[] tmp = new byte[buffer.length + GROW_SIZE];
System.arraycopy(buffer, 0, tmp, 0, buffer.length);
buffer = tmp;
```

Glücklicherweise lässt sich dies seit JDK 6 durch den Einsatz von statischen Hilfsmethoden aus der Utility-Klasse `Arrays` einfacher implementieren. Arrays und Teilbereiche können mit den für diverse Typen überladenen, statischen Hilfsmethoden `Arrays.copyOf(T[] original, int newLength)` bzw. `Arrays.copyOfRange(T[] original, int from, int to)` kopiert werden.

In der folgenden Methode `storeValueImproved(byte)` wird zur Größenanpassung die Methode `Arrays.copyOf(byte[], int)` wie folgt verwendet:

```
private static void storeValueImproved(final byte byteValue)
{
    if (writePos == buffer.length)
    {
        buffer = Arrays.copyOf(buffer, buffer.length + GROW_SIZE);
    }
    buffer[writePos] = byteValue;
    writePos++;
}
```

Die neuen Methoden in der Utility-Klasse `Arrays` sind praktisch: Sie erlauben, zu realisierende Aufgaben auf einer höheren Abstraktionsebene als mit `System.arraycopy()` zu beschreiben.

5.1.3 Das Interface `Collection`

Das Interface `Collection<E>` definiert die Basis für diverse Containerklassen, die das Interface `List<E>` bzw. `Set<E>` erfüllen und somit Listen bzw. Mengen repräsentieren. Wie bereits erwähnt, dienen Containerklassen dazu, mehrere Elemente zu speichern, auf diese zuzugreifen und gewisse Metainformationen (z. B. Anzahl gespeicherter Elemente) ermitteln zu können. Das Interface `Collection<E>` bietet *keinen* indizierten Zugriff, aber folgende Methoden:

- `int size()` – Ermittelt die Anzahl der in der Collection gespeicherten Elemente.
- `boolean isEmpty()` – Prüft, ob Elemente vorhanden sind.
- `boolean add(E element)` – Fügt ein Element zur Collection hinzu.
- `boolean addAll(Collection<? extends E> collection)` – Ist eine auf eine Menge bezogene, sogenannte Bulk-Operation (Massenoperation) und fügt alle übergebenen Elemente zur Collection hinzu.

Im Interface `Collection<E>` nutzen einige Methoden den Typparameter `Object` oder '`?`' und sind daher nicht typsicher[1]:

- `boolean remove(Object object)` – Entfernt ein Element aus der Collection.
- `boolean removeAll(Collection<?> collection)` – Entfernt mehrere Elemente aus der Collection.
- `boolean contains(Object object)` – Prüft, ob das Element in der Collection enthalten ist.
- `boolean containsAll(Collection<?> collection)` – Prüft, ob mehrere Elemente in der Collection enthalten sind.
- `boolean retainAll(Collection<?> collection)` – Behält alle Elemente einer Collection bei, die in der übergebenen Collection auch enthalten sind.

Hinweis: Typkürzel beim Einsatz von Generics

In den obigen Methodensignaturen haben wir folgende Typkürzel verwendet:

- `E` – Steht für den Typ der Elemente des Containers.
- `?` – Steht für einen unbekannten Typ.
- `? extends E` – Steht für einen unbekannten Typ, der ein Subtyp von `E` ist.

Die Buchstaben stellen lediglich Vereinbarungen dar und können beliebig gewählt werden. Ein konsistenter Einsatz erleichtert jedoch das Verständnis. Weitere gebräuchliche und in den folgenden Abschnitten genutzte Typkürzel sind:

- `T` – Steht für einen bestimmten Typ.
- `K` bzw. `V` – Steht bei Maps für den Typ des Schlüssels (`K` => key) bzw. des Werts (`V` => value).

[1] Vermutlich weil dies für eine Enthaltensein-Prüfung eine zu starke Einschränkung gewesen wäre.

Mengenoperationen auf Collections

Dieser Abschnitt verdeutlicht die Arbeitsweise einiger der zuvor genannten Methoden. Mit `contains(Object)` bzw. `containsAll(Collection<?>)` kann geprüft werden, ob ein oder mehrere gewünschte Elemente in einer Collection vorhanden sind. Man kann über `containsAll(Collection<?>)` bestimmen, ob eine Collection C eine Teilmenge einer Collection A ist. Mit `removeAll(Collection<?>)` lässt sich die Differenzmenge zweier Collections berechnen, indem z. B. aus einer Collection A alle Elemente einer anderen Collection B gelöscht werden. Mit `retainAll(Collection<?>)` berechnet man die Schnittmenge: Man behält in einer Collection A alle Elemente einer anderen Collection B. Zum leichten Verständnis ist die Arbeitsweise in Abbildung 5-3 für die Collections A, B und C visualisiert.

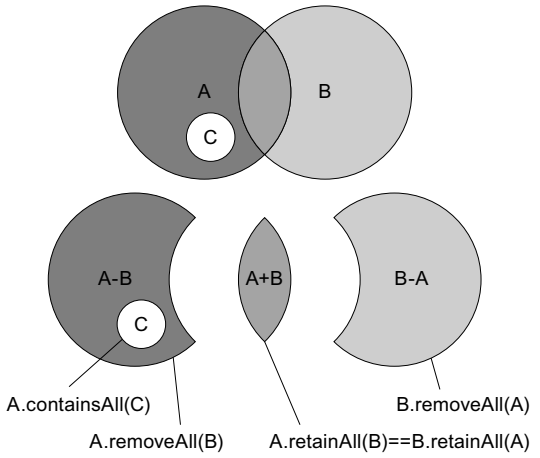


Abbildung 5-3 *Mengenoperationen auf Collections*

5.1.4 Das Interface `Iterator`

Alle Datenstrukturen, die das Interface `Collection<E>` erfüllen, bieten über die Methode `iterator()` Zugriff auf das Interface `java.util.Iterator<E>`, das einen sogenannten ***Iterator*** modelliert. Damit ist das Durchlaufen der Inhalte möglich, die in den Instanzen der Containerklassen des Collections-Frameworks gespeichert sind.

IDIOM: TRAVERSIERUNG VON COLLECTIONS MIT DEM INTERFACE ITERATOR

Zum Durchlaufen einer Collection mit Iteratoren definieren wir zunächst einen Datenbestand. Dabei nutzen wir den Trick, ein Array in eine Liste durch Aufruf der statischen Hilfsmethode `Arrays.asList(T...)` (vgl. Abschnitt 5.3.1) wandeln zu können. Das Ergebnis ist eine typisierte, aber insbesondere auch unmodifizierbare `List<T>`. Von dieser erhalten wir durch Aufruf von `iterator()` einen Iterator. Mit dessen Methode `hasNext()` kann man ermitteln, ob noch weitere Elemente zum Durchlaufen vorhan-

den sind. Ist dies der Fall, kann auf das nächste Element über die Methode `next()` zugegriffen werden. Damit ergibt sich folgendes Idiom zum Iterieren:

```
public static void main(final String[] args)
{
    final String[] textArray = { "Durchlauf", "mit", "Iterator" };
    final Collection<String> infoTexts = Arrays.asList(textArray);

    final Iterator<String> it = infoTexts.iterator();
    while (it.hasNext())
    {
        System.out.print(it.next());
        if (it.hasNext()) // Sonderbehandlung letzter Eintrag
            System.out.print(", ");
    }
}
```

Listing 5.1 *Ausführbar als* **'ITERATIONEXAMPLE'**

Wie erwartet, werden alle Elemente kommasepariert nacheinander ausgegeben:

```
Durchlauf, mit, Iterator
```

Löschfunktionalität im Interface `Iterator`

Im Interface `Iterator<E>` ist auch die parameterlose Methode `remove()` definiert, die es erlaubt, das aktuelle, d. h. das zuvor über die Methode `next()` ermittelte Element zu löschen. Allerdings muss nicht jede Realisierung eines Iterators auch tatsächlich diese Löschfunktionalität unterstützen. In diesem Fall sollte ein Aufruf von `remove()` laut JDK-Kontrakt eine `UnsupportedOperationException` auslösen. Dies gilt etwa dann, wenn es sich um eine unveränderliche Datenstruktur handelt.

Die Definition der Methode `remove()` im Interface `Iterator<E>` wirkt überflüssig, weil doch bereits im Interface `Collection<E>` eine Methode `remove(Object)` existiert. Warum diese scheinbar doppelte Definition notwendig ist, zeige ich an einem Beispiel. Nehmen wir dazu an, aus einer Liste von Stringobjekten sollen diejenigen herausgefiltert werden, die mit einer speziellen Zeichenkette beginnen. Eine intuitive Realisierung mit den zuvor vorgestellten Methoden der Interfaces `Collection<E>` und `Iterator<E>` sieht etwa folgendermaßen aus:

```
private static void removeEntriesWithPrefix(final List<String> entries,
                                            final String prefix)
{
   final Iterator<String> it = entries.iterator();
   while (it.hasNext())
   {
      final String name = it.next();
      if (name.startsWith(prefix))
      {
          // ACHTUNG: remove() der Collection ist intuitiv aber falsch
          entries.remove(name);
      }
   }
}
```

Schreiben wir ein Testprogramm, um die Funktionalität zu überprüfen. Wir definieren dazu einige Namen in einer Liste von Strings. Aus dieser sollen alle mit 'M' beginnenden Namen mit der zuvor definierten Methode gelöscht werden:

```
public static void main(final String[] args)
{
    final String[] names = { "Andreas", "Carsten", "Clemens",
                             "Merten", "Michael", "Peter" };

    final List<String> namesList = new ArrayList<>();
    namesList.addAll(Arrays.asList(names));

    removeEntriesWithPrefix(namesList, "M");
    System.out.println(namesList);
}
```

Listing 5.2 *Ausführbar als '***ITERATORCOLLECTIONREMOVEEXAMPLE***'*

Man würde erwarten, dass als Ergebnis die Namen "Andreas", "Carsten", "Clemens" und "Peter" in der Liste verbleiben und ausgegeben werden. Stattdessen kommt es zu einer java.util.ConcurrentModificationException:

```
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:781)
    at java.util.ArrayList$Itr.next(ArrayList.java:753)
   [...]
```

Eine derartige Exception deutet normalerweise auf Probleme und Veränderungen einer Datenstruktur bei parallelen Zugriffen durch mehrere Threads hin. Etwas merkwürdig ist das schon, weil hier lediglich ein Thread läuft. ***Es ist demnach möglich, eine auf Multithreading-Probleme hindeutende Exception selbst in einfachen Programmen ohne Nebenläufigkeit zu provozieren.*** Gehen wir der Sache auf den Grund. Verursacht wird die Exception dadurch, dass in jeder Collection ein Modifikationszähler zum Schutz vor konkurrierenden Zugriffen genutzt wird. Jeder Iterator ermittelt dessen Wert zu Beginn seiner Iteration und vergleicht diesen bei jedem Aufruf von next() mit dem Startwert. Weicht dieser Wert ab, so wird eine ConcurrentModificationException ausgelöst. Dieses Verhalten der Iteratoren nennt man ***fail-fast***.

Mit diesem Hintergrundwissen wird die Fehlerursache klar: Der Aufruf der Methode remove(Object) auf der Datenstruktur entries führt zu einer Änderung des Modifikationszählers! Die einzig sichere Art, während einer Iteration Elemente aus einer Collection zu löschen, ist durch Aufruf der Methode remove() aus dem Interface Iterator<E>. Wir korrigieren unsere Methode wie folgt:

```
private static void removeEntriesWithPrefix(final List<String> entries,
                                            final String prefix)
{
    final Iterator<String> it = entries.iterator();
    while (it.hasNext())
    {
        if (it.next().startsWith(prefix))
            it.remove();    // KORREKT: Zugriff über remove() des Iterators
    }
}
```

Man erkennt, dass die Methode `remove()` aus dem Interface `Iterator<E>` keinen Parameter benötigt. Der Grund ist einfach: Sie löscht immer das zuvor über die Methode `next()` ermittelte Element.

Anhand der geführten Diskussion ist verständlich, warum die Methode `remove()` nicht nur im Interface `Collection<E>`, sondern auch im Interface `Iterator<E>` definiert ist. Bei dessen Nutzung gibt es noch einen kleinen Fallstrick: Vorsicht ist geboten, wenn man beispielsweise zwei aufeinander folgende Elemente löschen möchte. Intuitiv könnte man dies folgendermaßen umsetzen:

```
it.remove();
// FALSCH: it.next()-Aufruf fehlt
it.remove();
```

Das führt zu einer `IllegalStateException`, da, wie zuvor beschrieben, einem Aufruf von `remove()` immer ein Aufruf von `next()` vorangehen muss.

Achtung: Die Methode `remove()` im Interface `Iterator`

Mein Verständnis von einem Iterator basiert auf dem Entwurfsmuster ITERATOR, das ich in Abschnitt 18.3.1 beschreibe. Laut dessen Definition sollte ein Iterator (lediglich) zum Durchlaufen einer Datenstruktur genutzt werden. Modifikationen der Datenstruktur waren dabei ursprünglich nicht vorgesehen. Man könnte daher die Existenz der Methode `remove()` im Interface `Iterator<E>` kritisieren. Aufgrund der Implementierungsentscheidung für die Fail-fast-Iteratoren wurde die Aufnahme dieser Methode in das Interface `Iterator<E>` allerdings notwendig, um Löschoperationen während einer Iteration zu erlauben.

Keine Methode `add()` Mit derselben Begründung könnte man für eine Methode `add(E)` im Interface `Iterator<E>` plädieren. Diese existiert aber aus gutem Grund nicht. Sie kann nicht angeboten werden, da das Einfügen eines Elements im Gegensatz zu dessen Entfernen nicht allgemeingültig auf Basis der aktuellen Position möglich ist: Für automatisch sortierende Container entspricht beispielsweise die momentane Position des Iterators in der Regel nicht der korrekten Einfügeposition in der Datenstruktur.

5.1.5 Listen und das Interface `List`

Unter einer Liste versteht man eine geordnete Folge von Elementen. Das Collections-Framework definiert zur Beschreibung von Listen das Interface `List<E>`. Bekannte Implementierungen sind die Klassen `ArrayList<E>` und `LinkedList<E>` sowie `Vector<E>`. Das Interface `List<E>` ermöglicht einen indizierten Zugriff und erlaubt das Hinzufügen und Entfernen von Elementen – wobei es vereinzelte Ausnahmen gibt.[2]

[2] Die von `Collections.unmodifiableList(List<? extends T>)` erzeugte Spezialisierung einer Liste stellt lediglich eine unveränderliche Sicht dar. Ein Aufruf von verändernden Methoden führt zu `UnsupportedOperationException`s.

Das Interface `List`

Das Interface `List<E>` bildet die Basis für alle Listen und bietet *zusätzlich* zu den Methoden des Interface `Collection<E>` folgende indizierte, 0-basierte Zugriffe:

- `E get(int index)` – Ermittelt das Element der Liste an der Position `index`.
- `void add(int index, E element)` – Fügt das Element `element` an der Position `index` der Liste ein.
- `E set(int index, E element)` – Ersetzt das Element an der Position `index` der Liste durch das übergebene Element `element`. Liefert das zuvor an dieser Position gespeicherte Element zurück.[3]
- `E remove(int index)` – Entfernt das Element an der Position `index` der Liste. Liefert das gelöschte Element zurück.
- `int indexOf(Object object)` und
- `int lastIndexOf(Object object)` – Mit diesen Methoden wird die Position eines gesuchten Elements zurückgeliefert. Gleichheit wird mit der Methode `equals(Object)` überprüft. Die Suche startet dabei entweder am Anfang (`indexOf(Object)`) oder am Ende der Liste (`lastIndexOf(Object)`).

Folgendes Listing zeigt einige der obigen Methoden im Einsatz. Zunächst werden einer `ArrayList<E>` verschiedene Elemente am Ende und per Positionsangabe hinzugefügt, danach wird indiziert zugegriffen. Schlussendlich werden Löschoperationen per Index ausgeführt, wobei im letzteren Fall zuvor eine Suche mit `indexOf(Object)` erfolgt:

```
public static void main(final String[] args)
{
    // Erzeugen und Hinzufügen von Elementen
    final List<String> list = new ArrayList<>();
    list.add("First");
    list.add("Last");
    list.add(1, "Middle");
    System.out.println("List: " + list);

    // Indizierter Zugriff
    System.out.println("3rd:  " + list.get(2));

    // Vorderstes Element löschen, "Last" mit indexOf() suchen und löschen
    list.remove(0);
    list.remove(list.indexOf("Last"));
    System.out.println("List: " + list);
}
```

Listing 5.3 *Ausführbar als* **'FIRSTLISTEXAMPLE'**

Startet man das Programm, so kommt es zu folgender Ausgabe:

```
List: [First, Middle, Last]
3rd:  Last
List: [Middle]
```

[3] Es kommt zu einer `IndexOutOfBoundsException`, falls kein Element an dieser Position existiert. Für `add()` ist jedoch auch der nicht existierende Index `list.size()` erlaubt.

Sublisten Die Methode `List<E> subList(int, int)` liefert einen Ausschnitt aus der Liste von Position `fromIndex` (einschließlich) bis `toIndex` (ausschließlich) und ermöglicht verschiedene Operationen auf Teillisten: Da die Rückgabe vom Typ `List<E>` ist, können tatsächlich alle Methoden des Interface `List<E>` aufgerufen werden. Man kann beispielsweise innerhalb eines gewissen Bereichs eine Suche durchführen oder diesen Bereich löschen. ***Dabei sollte man allerdings beachten, dass lediglich eine Sicht auf die ursprüngliche Liste geliefert wird***. Dadurch kommt es bei Veränderungen an der Teilliste auch zu Änderungen in der ursprünglichen Liste.

Um das zu verdeutlichen, zeige ich die Realisierung einer Hilfsmethode `truncateListToMaxSize()`, die eine übergebene Liste auf eine gewünschte Länge zurechtstutzt, sofern diese länger ist. Als Eingabe dient hier eine Liste mit fünf Fehlertexten als Strings, die auf die Länge drei gekürzt wird:

```
public static void main(final String[] args)
{
    // Merkwürdige Aufrufkombination
    final List<String> errors = new ArrayList<>(Arrays.asList(
                                        "Error1", "Error2", "Error3",
                                        "Critical Error", "Fatal Error"));

    truncateListToMaxSize(errors, 3);

    System.out.println(errors);  // [Error1, Error2, Error3]
}

// ACHTUNG: Hier Seiteneffekt: Übergebene Liste wird gegebenenfalls verkleinert
private static void truncateListToMaxSize(final List<?> listToTruncate,
                                          final int maxSize)
{
    if (listToTruncate.size() > maxSize)
    {
        final List<?> entriesAfterMaxSize = listToTruncate.subList(maxSize,
                                                    listToTruncate.size());

        // ACHTUNG: clear() wirkt sich auch in der Originalliste aus
        entriesAfterMaxSize.clear();
    }
}
```

Listing 5.4 *Ausführbar als* **'SUBLISTEXAMPLE'**

Hier kommt die im Listing fett markierte, etwas merkwürdig anmutende Kombination aus Konstruktoraufruf und Hilfsmethode zum Einsatz. Das ist notwendig, weil `Arrays.asList(...)` eine unmodifizierbare Liste zurückgibt, die dann durch den Konstruktoraufruf in eine neue veränderliche Liste überführt wird. Eleganter lässt sich das mithilfe einer Hilfsmethode `asModifiableList(T...)` wie folgt schreiben:

```
final List<String> errors = asModifiableList("Error1", "Error2", "Error3",
                                              "Critical Error", "Fatal Error");

private static <T> List<T> asModifiableList(T... items)
{
    return new ArrayList<>(Arrays.asList(items));
}
```

Anstatt derartige Hilfsmethoden selbst zu schreiben, sollte man besser einen Blick in existierende Bibliotheken wie Apache Commons oder Google Guava werfen. Insbesondere Letztere besprechen wir etwas genauer in Kapitel 6.

Das Interface `ListIterator`

Alle Datenstrukturen, die das Interface `List<E>` erfüllen, bieten über die Methode `listIterator()` Zugriff auf einen speziellen Iterator vom Typ `ListIterator<E>`, der auf die Besonderheiten des indizierten Zugriffs angepasst wurde. Mit einem solchen Iterator ist das Durchlaufen einer Liste zusätzlich zu einem normalen Iterator auch in Rückwärtsrichtung möglich. Dazu dienen die beiden Methoden `hasPrevious()` sowie `previous()`. Außerdem kann man den Index des nächsten bzw. des vorherigen Elements über die Methoden `nextIndex()` bzw. `previousIndex()` abfragen.

Achtung: Die Methoden `set()` und `add()` im Interface `ListIterator`

Zusätzlich zur Methode `remove()` wurden in das Interface `ListIterator<E>` mit den Methoden `set(E)` und `add(E)` zwei weitere Daten verändernde Methoden eingeführt. Gemäß der Argumentation aus dem vorherigen Praxistipp kann man deren Existenz kritisieren. Wenn man Listen während einer Iteration manipulieren will, muss man diese modifizierenden Methoden im `ListIterator<E>` allerdings anbieten. Ohne sie würde aufgrund der Fail-fast-Eigenschaft ein Aufruf etwa der Methode `add(E)` aus dem Interface `Collection<E>` eine Exception auslösen.

Arbeitsweise der Klassen `ArrayList` und `Vector`

Die Klassen `ArrayList<E>` und `Vector<E>` nutzen zur Datenspeicherung Arrays und erweitern diese um Containerfunktionalität: Wächst die Anzahl zu speichernder Elemente, so wird automatisch dafür gesorgt, dass ausreichend Speicher zur Verfügung steht. Bei Überschreiten der Kapazität des verwendeten Arrays wird automatisch ein neues, größeres Array angelegt und die Elemente des alten Arrays werden in das neue kopiert. Fortan wird das neue Array zur Speicherung der Elemente verwendet. Das alte Array ist daraufhin obsolet. Die Tatsache, dass intern mit Arrays gearbeitet wird, ist für den Benutzer transparent. Neben dieser Kapselung und der automatischen Größenanpassung, die für viele Anwendungen entscheidende Vorteile gegenüber dem Einsatz von Arrays darstellen, bieten die Klassen `ArrayList<E>` und `Vector<E>` einige weitere Annehmlichkeiten einer Containerklasse: Ein Beispiel dafür ist die Unterscheidung zwischen der Füllstandsabfrage (`size()`), also der Anzahl der momentan gespeicherten Elemente, und den zur Verfügung stehenden Speicherplätzen (Kapazität), die die tatsächliche Größe des Arrays, also das momentane Fassungsvermögen, bestimmt. Der Aufbau einer Array-basierten Liste wird in Abbildung 5-4 skizziert, wobei die Texte `Obj 1`, `Obj 2` usw. nicht das Objekt selbst, sondern die Referenz darauf repräsentieren.

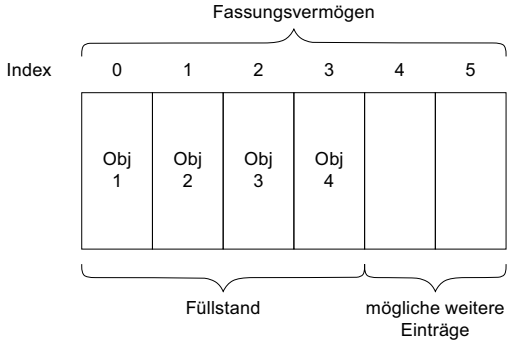
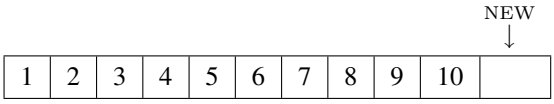


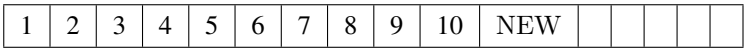
Abbildung 5-4 *Array der Klasse* `ArrayList`

Zugriff auf ein Element an Position $index$ **– `get(index)`** Der Zugriff auf beliebige Elemente wird durch einen Array-Zugriff realisiert und ist sehr performant.

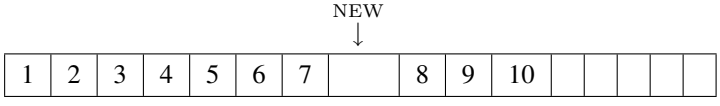
Element an letzter Position hinzufügen – `add(element)` Nehmen wir an, es ist (gerade) noch ausreichend Kapazität im Array vorhanden und es soll ein Element als letztes Element in die Datenstruktur eingefügt werden. In diesem Fall muss nur die Referenz in dem Array gespeichert werden:



Das zugrunde liegende Array ist jetzt komplett belegt und es ist kein Platz für ein weiteres Element vorhanden. Soll erneut ein Element hinzugefügt werden, muss als Folge das Array in seiner Größe angepasst werden:



Element an Position $index$ **einfügen – `add(index, element)`** Wird an einer Position $index$ ein Element eingefügt, so müssen als Folge alle Elemente mit einer Position $>= index$ nach hinten verschoben werden, um Platz für das neue Element zu schaffen. Mit $index = 0$ muss der Inhalt des gesamten Arrays verschoben werden. Reicht die Kapazität nicht mehr aus, so wird Speicher für ein neues Array alloziert und mit dem Inhalt aus dem alten Array gefüllt. Im Folgenden ist dies für das Einfügen an Position 8 und das Element NEW gezeigt:



Element an Position $index$ entfernen – `remove(index)` Wird an einer Position $index$ ein Element gelöscht, so müssen als Folge alle Elemente des Arrays mit einer Position $> index$ nach vorne verschoben werden. Im Extremfall mit $index = 0$ geschieht dies für das gesamte Array.

Größenanpassungen und Speicherverbrauch Beim Einfügen sorgen die Klassen `ArrayList<E>` und `Vector<E>` automatisch dafür, dass bei Bedarf die Größe schrittweise angepasst wird. Die Kopiervorgänge kosten mit zunehmender Größe immer mehr Zeit – aber kaum relevant. Zudem muss der Garbage Collector (vgl. Abschnitt 8.5) die obsoleten Arrays wieder wegräumen: Das ist Arbeit, die sich häufig verringern oder vermeiden lässt, indem man ***die erwartete Maximalgröße als Konstruktorparameter übergibt***. Allerdings ist die Wahl einer geeigneten Größe, wie bereits in Abschnitt 5.1.2 für Arrays erwähnt, nicht immer einfach oder gar möglich.

Bei zunehmendem Datenvolumen erhöht sich zudem die Wahrscheinlichkeit für Probleme durch die Speicherung als Array, weil immer ein zusammenhängender Speicherbereich benötigt wird. Je größer die Anzahl der Elemente wird, desto stärker wirkt sich dies aus. Zwei Probleme treten auf: Erstens kann im Extremfall eine Out-of-Memory-Situation eintreten, obwohl im Grunde noch genug Speicher vorhanden ist, aber kein zusammenhängender Speicherbereich der erforderlichen Größe bereitgestellt werden kann. Zweitens werden bei einem Vergrößerungsschritt beim Kopieren in ein neues, größeres Array temporär zwei Arrays gebraucht: einmal das alte und dann noch das neue Array. Wenn das alte Array z. B. 500 MB groß ist, dann benötigt man beim Kopieren vorübergehend ungefähr 1,25 GB.[4] Das kann ebenfalls zu einer Out-of-Memory-Situation führen, obwohl eigentlich noch genug Speicher vorhanden ist – allerdings nur für eine Version des Arrays. Besonders verwirrend ist dies, wenn man als Programmierer nicht weiß, dass im Hintergrund eine Kopie angelegt wird.

Achtung: Versteckte Memory Leaks und Abhilfemaßnahmen

Die Größe des datenspeichernden Arrays wird bei Einfügeoperationen bei Bedarf automatisch angepasst. Für das Entfernen von Elementen gilt dies allerdings nicht: Eine einmal bereitgestellte Kapazität wird dabei nicht wieder reduziert.

Wurde einmalig viel Speicher alloziert, so erzeugt man ein verstecktes ***Memory Leak***, dadurch, dass die `ArrayList<E>` bzw. der `Vector<E>` immer noch den gesamten Speicher belegt, obwohl durch Löschoperationen mittlerweile viel weniger Elemente zu speichern sind.

Mithilfe der Methode `trimToSize()` kann man in einem solchen Fall dafür sorgen, dass das Array auf die benötigte Größe verkleinert wird. Der zuvor belegte Speicher ist anschließend unreferenziert und kann vom Garbage Collector freigeräumt werden. Der Applikation steht dieser Speicher daraufhin wieder zur Verfügung.

[4] Das neu entstehende Array ist um die Hälfte größer als die ursprüngliche Größe, weil diese Vergrößerung derart in der Implementierung der `ArrayList<E>` programmiert ist. Im Beispiel wäre die neue Größe also 750 MB.

`ArrayList` oder `Vector`? Die Klassen `ArrayList<E>` und `Vector<E>` unterscheiden sich in ihrer Arbeitsweise lediglich in einem Detail: In der Klasse `Vector<E>` sind die Methoden `synchronized` definiert, um bei konkurrierenden Zugriffen für Konsistenz der gespeicherten Daten zu sorgen. Häufig möchte man in einer Anwendung eine Kombination mehrerer Aufrufe schützen, sodass diese feingranulare Art der Synchronisierung nicht ausreichend für die benötigte Art von Thread-Sicherheit ist (vgl. Kapitel 7). Somit gibt es eher selten Anwendungsfälle für einen `Vector<E>` und in der Regel sollte man die ***`ArrayList<E>` bevorzugen***.

Arbeitsweise der Klasse `LinkedList`

Die Klasse `LinkedList<E>` verwendet zur Speicherung von Elementen miteinander verbundene kleine Einheiten, sogenannte ***Knoten*** oder ***Nodes***. Jeder Knoten speichert sowohl eine Referenz auf die Daten als auch jeweils eine Referenz auf Vorgänger und Nachfolger. Dadurch wird eine Navigation vorwärts und rückwärts möglich. Diese Art der Speicherung führt dazu, dass die `LinkedList<E>` nur eine Größe (Anzahl der Knoten), aber keine Kapazität besitzt. Den schematischen Aufbau zeigt Abbildung 5-5, wobei `Obj 1`, `Obj 2` usw. Referenzen auf Objekte repräsentieren.

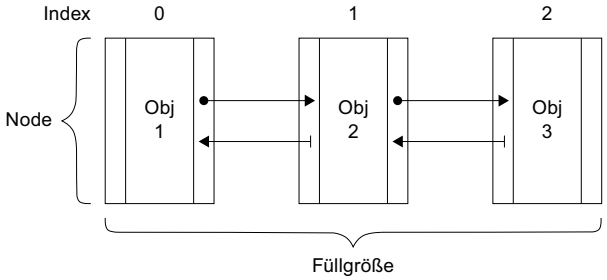


Abbildung 5-5 *Aufbau eines Objekts der Klasse* `LinkedList`

Zugriff auf ein Element an Position $index$ – `get(index)` Durch die Organisation als verkettete Liste erfordert ein indizierter Zugriff auf ein Element an einer Position $index$ immer einen Durchlauf bis zu der gewünschten Stelle. Als Optimierung werden Zugriffe entweder vom Anfang oder Ende gestartet, abhängig davon, was davon näher bei dem Zielindex liegt. Im Gegensatz zur `ArrayList<E>` mit konstanter Zugriffszeit wächst daher die Zugriffszeit bei der `LinkedList<E>` linear mit der Anzahl der gespeicherten Elemente. Dies kann bei relativ großen Datenmengen (genaue Angaben sind schwierig, in der Regel aber mehr als 500.000 Einträge) negative Auswirkungen auf die Laufzeit haben. Das zeigt sich deutlich bei der Optimierung der Darstellung umfangreicher Datenmengen mit einer `JTable` in Abschnitt 22.4.

Element an letzter Position hinzufügen – `add(element)` Wenn ein Element hinten an letzter Position angefügt werden soll, so wird zunächst ein neuer Knoten erzeugt und danach mit dem bisher letzten Knoten verbunden.

Element an Position $index$ einfügen – `add(index, element)` Um ein Element an einer beliebiger Position einzufügen, wird ein neuer Knoten erzeugt. Zudem muss die Einfügeposition bestimmt werden, was linearen Aufwand durch eine Iteration durch die Liste bis zu der gewünschten Stelle erfordert. Schließlich sind nur einige Referenzen umzusetzen, um den neuen Knoten in die Liste einzufügen.

Element an Position $index$ entfernen – `remove(index)` Für eine Löschoperation sind lediglich Referenzanpassungen nötig, um den zu löschenden Knoten aus der Liste auszuschließen. Auch hier muss zunächst mit linearem Aufwand zur Löschposition $index$ iteriert werden.

Größenanpassungen und Speicherverbrauch Die `LinkedList<E>` besitzt bezüglich des Speicherverbrauchs einige Vorteile gegenüber der `ArrayList<E>`. Erstens wird, abgesehen von einem gewissen Overhead, immer nur genau so viel Speicher für Elemente belegt, wie tatsächlich benötigt wird. Zweitens kann durch den Garbage Collector eine automatische Freigabe des Speichers an das System erfolgen, wenn Elemente gelöscht werden. Insbesondere bei großer Dynamik und temporär hohen Datenvolumina ist dies von Vorteil, da Speicherbereiche nicht unbenutzt belegt bleiben, wie dies beim Einsatz der `ArrayList<E>` der Fall sein kann. Allerdings wird das durch einen Nachteil erkauft: Während eine `ArrayList<E>` für jedes gespeicherte Element lediglich eine Objektreferenz hält, speichert jeder Knoten einer `LinkedList<E>` zusätzlich je eine Referenz auf den Vorgänger und auf den Nachfolger. Somit benötigt eine `LinkedList<E>` zur Verwaltung insgesamt mehr Speicher (in etwa Faktor drei) als eine `ArrayList<E>` mit gleicher Anzahl gespeicherter Elemente. Beachten Sie unbedingt, dass sich dieser Speicherbedarf nur auf den Verbrauch durch die Datenstruktur selbst bezieht und nicht auf den Speicherplatzbedarf der dort referenzierten Objekte, der in der Regel um Größenordnungen höher sein wird.

5.1.6 Mengen und das Interface `Set`

Das mathematische Konzept der Mengen besagt, dass diese keine Duplikate enthalten. In Java werden Mengen durch des Interface `Set<E>` beschrieben, das auf dem Interface `Collection<E>` basiert. Im Gegensatz zum Interface `List<E>` sind im Interface `Set<E>` keine Methoden zusätzlich zu denen des Interface `Collection<E>` vorhanden – allerdings wird ein anderes Verhalten für die Methoden `add(E)` und `addAll(Collection<? extends E>)` vorgeschrieben. Dieser Unterschied zwischen `Set<E>` und dem zugrunde liegende Interface `Collection<E>` ist nötig, um Duplikatfreiheit zu garantieren, selbst dann, wenn der Menge das gleiche Objekt mehrfach hinzugefügt wird.

Beispiel: Realisierungen von Mengen und ihre Besonderheiten

Um ein wenig vertraut mit Mengen zu werden, erzeugen wir mit einem `HashSet<E>` und einem `TreeSet<E>` zwei verschiedene Typen von Mengen und füllen jeweils eigene Instanzen davon mit Werten vom Typ `String` und auch `StringBuilder`:

```
public static void main(final String[] args)
{
    fillAndExploreHashSet();
    fillAndExploreTreeSet();
}

private static void fillAndExploreHashSet()
{
    // String definiert hashCode() und equals()
    final Set<String> hashSet = new HashSet<String>();
    addStringDemoData(hashSet);
    System.out.println(hashSet);

    // StringBuilder hat weder hashCode() noch equals()
    final Set<StringBuilder> hashSetSurprise = new HashSet<StringBuilder>();
    addStringBuilderDemoData(hashSetSurprise);
    System.out.println(hashSetSurprise);
}

private static void fillAndExploreTreeSet()
{
    // String implementiert Comparable
    final Set<String> treeSet = new TreeSet<String>();
    addStringDemoData(treeSet);
    System.out.println(treeSet);

    // StringBuilder implementiert Comparable nicht
    final Set<StringBuilder> treeSetSurprise = new TreeSet<StringBuilder>();
    addStringBuilderDemoData(treeSetSurprise);
    System.out.println(treeSetSurprise);
}

private static void addStringDemoData(final Set<String> set)
{
    set.add("Hallo");
    set.add("Welt");
    set.add("Welt");
}

private static void addStringBuilderDemoData(final Set<StringBuilder> set)
{
    set.add(new StringBuilder("Hallo"));
    set.add(new StringBuilder("Welt"));
    set.add(new StringBuilder("Welt"));
}
```

Listing 5.5 *Ausführbar als* **'FIRSTSETEXAMPLE'**

Starten wir das Programm FIRSTSETEXAMPLE, so kommt es zu folgenden Ausgaben:

```
[Hallo, Welt]
[Welt, Hallo, Welt]
[Hallo, Welt]
Exception in thread "main" java.lang.ClassCastException: java.lang.StringBuilder
       cannot be cast to java.lang.Comparable
```

Das Beispiel zeigt, dass man zum sicheren Umgang mit den Mengen-Datenstrukturen verstehen sollte, welche Mechanismen die Eindeutigkeit von Elementen innerhalb einer Menge bewirken: Für Strings arbeitet alles wie erwartet, für den Typ `StringBuilder` werden Duplikate nicht erkannt und für `TreeSet<StringBuilder>` sogar eine Exception ausgelöst. Für zu speichernde Klassen ist eine korrekte und den jeweiligen Kontrakten folgende Implementierung einiger Methoden erforderlich. Für die Klasse `HashSet<E>` sind dies die Methoden `hashCode()` zum Auffinden von Elementen sowie `equals(Object)` zur Garantie von Eindeutigkeit. Die Klasse `TreeSet<E>` nutzt dazu die Methoden `compareTo(T)` bzw. `compare(T, T)` aus den Interfaces `Comparable<T>` bzw. `Comparator<T>`. Abschnitt 5.1.9 geht auf das Zusammenspiel der relevanten Methoden im Detail ein. In den Abschnitten 5.1.7 und 5.1.8 werden zuvor sowohl die Grundlagen von hashbasierten Containern (zum Verständnis der Klasse `HashSet<E>`) als auch die Grundlagen automatisch sortierender Container (als Basis für die Klasse `TreeSet<E>`) vorgestellt.

Bevor wir tiefer in die Details abtauchen, wollen wir zunächst einfache Beispiele für `HashSet<E>` und `TreeSet<E>` betrachten, um ein Gefühl für die Arbeit mit Mengen zu erhalten. Komplettiert wird das Ganze durch ein Praxisbeispiel: Wir erweitern die bereits realisierte Verzeichnisüberwachung und setzen dazu Mengen ein.

Fallstrick: Fehlende Angabe eines Sortierkriteriums

Zur Kompilierzeit wird für ein `TreeSet<E>` nicht geprüft, ob nur Objekte gespeichert werden, die das Interface `Comparable<T>` erfüllen. Das ist durchaus berechtigt, da auch ein `Comparator<T>` zur Beschreibung des Sortierkriteriums dienen kann. Eine fehlende Angabe eines Sortierkriteriums macht sich daher erst zur Laufzeit beim Einfügen von Elementen durch eine `java.lang.ClassCastException` bemerkbar.

Die Klasse `HashSet`

Die Klasse `HashSet<E>` ist eine Spezialisierung der abstrakten Klasse `AbstractSet<E>` und speichert Elemente ungeordnet in einem Hashcontainer (genauer: in einer später in Abschnitt 5.1.10 vorgestellten `HashMap<K,V>`). Dadurch wird ein geringer Laufzeitbedarf für die Operationen `add(E)`, `remove(Object)`, `contains(Object)` usw. ermöglicht.

Betrachten wir ein kurzes Beispiel, in dem die Werte `1` bis `3` in absteigender Reihenfolge in ein `HashSet<Integer>` eingefügt werden:

```
public static void main(final String[] args)
{
    final Integer[] ints = new Integer[] { 3, 2, 1 };
    final Set<Integer> numberSet = new HashSet<>(Arrays.asList(ints));
    System.out.println("Initial: " + numberSet);   // 1, 2, 3
}
```

Listing 5.6 *Ausführbar als* '**HASHSETSTORAGEEXAMPLE**'

Bei einem Blick auf die Ausgabe "`1, 2, 3`" scheint ein `HashSet<Integer>` die natürliche Ordnung der eingefügten Werte herzustellen. ***Dies ist jedoch nur ein zufälliger Effekt***. Dieser wird durch kleine Datenmengen und die gewählte Abbildung der zu speichernden Daten ausgelöst. Bei der Speicherung von Werten darf man sich bei einer *ungeordneten Menge*, wie sie von der Klasse `HashSet<E>` realisiert wird, *niemals* auf eine *definierte* Reihenfolge der Elemente verlassen. Dies wird deutlich, wenn man weitere Elemente einfügt, etwa die Werte `33`, `11` und `22`:

```
public static void main(final String[] args)
{
    final Integer[] ints = new Integer[] { 3, 2, 1 };
    final Set<Integer> numberSet = new HashSet<>(Arrays.asList(ints));
    System.out.println("Initial: " + numberSet);   // 1, 2, 3

    final Integer[] moreInts = new Integer[] { 33, 11, 22 };
    numberSet.addAll(Arrays.asList(moreInts));
    System.out.println("Add: " + numberSet);       // 1, 33, 2, 3, 22, 11
}
```

Listing 5.7 *Ausführbar als* **'HASHSETSTORAGEEXAMPLE2'**

Es kommt nun zu der zufällig wirkenden Ausgabe "`1, 33, 2, 3, 22, 11`", die durch die Verteilung im Container verursacht wird.

Benötigt man eine Ordnung der Elemente, so bietet sich der Einsatz der im Folgenden beschriebenen Klasse `TreeSet<E>` zur Speicherung von Elementen an.

Die Klasse `TreeSet`

Die Klasse `TreeSet<E>` implementiert das Interface `SortedSet<E>` und speichert Elemente sortiert. Die Sortierung wird entweder durch das Interface `Comparable<T>` oder einen explizit im Konstruktor übergebenen `Comparator<T>` festgelegt. Wir schauen auf ein ähnliches Beispiel wie für die Klasse `HashSet<E>`:

```
public static void main(final String[] args)
{
    final Integer[] ints = new Integer[] { 3, 2, 1 };
    final Set<Integer> numberSet = new TreeSet<>(Arrays.asList(ints));
    System.out.println("Initial: " + numberSet); // 1, 2, 3

    final Integer[] moreInts = new Integer[] { 33, 11, 22 };
    numberSet.addAll(Arrays.asList(moreInts));
    System.out.println("Add: " + numberSet); // 1, 2, 3, 11, 22, 33
}
```

Listing 5.8 *Ausführbar als* **'TREESETSTORAGEEXAMPLE'**

Startet man das Programm TREESETSTORAGEEXAMPLE, so sieht man, dass die Elemente sortiert im Container gespeichert werden. Weil hier im Konstruktor kein `Comparator<T>` übergeben wurde, basiert die Sortierung auf `Comparable<T>`, das von der zu speichernden Klasse `Integer` erfüllt wird.

Das Interface `SortedSet<E>` Die Klasse `TreeSet<E>` bietet neben der automatischen Sortierung folgende nützliche Funktionalität aus dem Interface `SortedSet<E>`:

- `E first()` und `E last()` – Mit diesen beiden Methoden kann das erste bzw. letzte Element der Menge ermittelt werden.
- `SortedSet<E> headSet(E toElement)` – Liefert die Teilmenge der Elemente, die kleiner als das übergebene Element `toElement` sind.
- `SortedSet<E> tailSet(E fromElement)` – Liefert die Teilmenge der Elemente, die größer oder gleich dem übergebenen Element `fromElement` sind: Ein übergebenes Element ist im Gegensatz zu `headSet(E)` in der zurückgelieferten Menge enthalten, wenn es Bestandteil der Originalmenge war.
- `SortedSet<E> subSet(E fromElement, E toElement)` – Liefert die Teilmenge der Elemente, startend von inklusive `fromElement` bis exklusive `toElement`.

Wir bauen unser Beispiel ein wenig aus und integrieren zwei Änderungsaktionen, um zu zeigen, dass es sich bei den durch die obigen Methoden gelieferten Sets jeweils um Sichten handelt, die Änderungen propagieren:

```
public static void main(final String[] args)
{
    final Integer[] ints = new Integer[] { 3, 2, 1, 33, 11, 22 };
    final SortedSet<Integer> numberSet = new TreeSet<>(Arrays.asList(ints));
    System.out.println("Initial: " + numberSet);   // 1, 2, 3, 11, 22, 33

    System.out.println("first: " + numberSet.first());           // 1
    System.out.println("last: " + numberSet.last());             // 33

    final SortedSet<Integer> headSet  = numberSet.headSet(7);
    System.out.println("headSet:   " + headSet);                 // 1, 2, 3
    System.out.println("tailSet:   " + numberSet.tailSet(7));    // 11, 22, 33
    System.out.println("subSet:    " + numberSet.subSet(7, 23)); // 11, 22

    // Modifikationen an einem einzelnen Set
    headSet.remove(3);
    headSet.add(6);
    System.out.println("headSet:   " + headSet);   // 1, 2, 6
    System.out.println("numberSet: " + numberSet); // 1, 2, 6, 11, 22, 33
}
```

Listing 5.9 *Ausführbar als* '**TreeSetStorageExample2**'

Praxisbeispiel für den Einsatz von Mengen

In Abschnitt 4.6.1 wurde ein Verzeichnisüberwachungstool entwickelt, das lediglich eine Änderung der Anzahl in einem Verzeichnis enthaltener Dateien erkennen konnte. Die Kombination der Aktionen »Hinzufügen«, »Löschen« und »Umbenennen« von Dateien verändert die Anzahl der Elemente aber nicht zwingend. Die Klasse `DirectoryObserver` soll nun derart erweitert werden, dass Veränderungen spezifischer – in Form hinzugefügter bzw. gelöschter Dateien – ermittelt werden. Genauere Aussagen kann

man nur dann treffen, wenn man den aktuellen sowie den vorherigen Verzeichnisinhalt speichert und Veränderungen bestimmt.

Im folgenden Beispiel der Klasse `DirectoryCheckerReportingChanges` nutzen wir ein `TreeSet<String>` zur Datenspeicherung des Verzeichnisinhalts. Dort werden Dateinamen als Objekte vom Typ `String` gespeichert und automatisch sortiert. Die Methode `checkForContentsChanged(int)` aus der Basisklasse wird überschrieben und in ihrer Funktionalität so erweitert, dass sie Mengenoperationen verwendet. Sie ermittelt durch Aufruf der bereits bekannten Methode `getContents()` den aktuellen Verzeichnisinhalt und wandelt diesen über den Umweg einer Liste in ein `TreeSet<String>` um. Dabei nutzen wir, dass alle konkreten Realisierungen des Interface `Collection<E>` einen speziellen Konstruktor mit einem Parameter vom Typ `Collection<E>` als Eingabe anbieten und sich dadurch in beliebige andere Realisierungen dieses Interface konvertieren lassen. Die Änderungen am Verzeichnisinhalt ermitteln wir über Mengenoperationen. Damit ergibt sich folgende Implementierung:

```
public class DirectoryCheckerReportingChanges extends DirectoryObserver
{
    private final Set<String> savedContent = new TreeSet<>();

    public DirectoryCheckerReportingChanges(final String nameOfDirectoryToCheck)
           throws IOException
    {
        super(nameOfDirectoryToCheck);
    }

    @Override
    protected int checkForContentsChanged(final int numOfFiles)
    {
        final String[] content = FileUtils.getContents(getDirectoryToCheck());
        final List<String> contentAsList = Arrays.asList(content);
        final Set<String> contentAsSet = new TreeSet<>(contentAsList);

        // Neu = Differenzmenge Aktuell - Vorher
        final Set<String> newContent = new TreeSet<>(contentAsSet);
        newContent.removeAll(savedContent);

        // Gelöscht = Differenzmenge Vorher - Aktuell
        final Set<String> oldContent = new TreeSet<>(savedContent);
        oldContent.removeAll(contentAsSet);

        // Unverändert = Schnittmenge Vorher und Aktuell
        final Set<String> unchangedContent = new TreeSet<>(savedContent);
        unchangedContent.retainAll(contentAsSet);

        if (newContent.size() > 0 || oldContent.size() > 0)
        {
            onContentsChanged(contentAsSet.size(), savedContent.size());
            onFilesAdded(newContent);
            onFilesRemoved(oldContent);
        }

        savedContent.clear();
        savedContent.addAll(contentAsSet);

        return savedContent.size();
    }
```

```
    protected void onFilesAdded(final Set<String> newContent)
    {
        System.out.println("addedFiles = '" + newContent +"'");
    }

    protected void onFilesRemoved(final Set<String> removedContent)
    {
        System.out.println("removedFiles = '" + removedContent +"'");
    }

    public static void main(final String[] args) throws IOException
    {
         // Zugriff auf das systemspezifische tmp-Directory
        final String tmpDir = System.getProperty("java.io.tmpdir");
        new DirectoryCheckerReportingChanges(tmpDir).checkDirectory();
    }
}
```

Listing 5.10 *Ausführbar als* **'DIRECTORYCHECKERREPORTINGCHANGES'**

Wie aus dem Listing ersichtlich, kann man basierend auf dem aktuellen und dem zuletzt gespeicherten Verzeichnisinhalt mit den Methoden `removeAll(Collection<?>)` und `retainAll(Collection<?>)` problemlos folgende Mengen berechnen:

- Neu ***hinzugefügte*** Dateien bestimmt man aus der Differenz zwischen dem aktuellen Inhalt und dem vorherigen Inhalt.
- Dateien, die ***gelöscht*** wurden, ergeben sich aus der Differenzmenge des vorherigen Verzeichnisinhalts und dem aktuellen.
- Die ***unveränderten*** Dateien kann man ermitteln, indem die Schnittmenge zwischen dem aktuellen und dem vorherigen Inhalt berechnet wird.

Nachdem so die entsprechenden Schnitt- und Differenzmengen bestimmt sind, können als Erweiterung zu der bereits bekannten Änderungsbenachrichtigung `onContentsChanged(int, int)` die zwei zusätzlichen Benachrichtigungsmethoden `onFilesAdded(Set<String>)` und `onFilesRemoved(Set<String>)` realisiert werden. Als Eingabe erhalten diese Methoden jeweils Mengen von Dateinamen. Zur Demonstration erfolgt im Programm lediglich eine Ausgabe auf der Konsole.

5.1.7 Grundlagen von hashbasierten Containern

Arrays und Listen haben einen in manchen Situationen unangenehmen Nachteil: Die Suche nach gespeicherten Daten und der Zugriff auf diese kann sehr aufwendig sein. Im Extremfall müssen alle enthaltenen Elemente betrachtet werden. Hashbasierte Container zeichnen sich dagegen dadurch aus, dass Suchen und diverse Operationen extrem performant ausgeführt werden können. Die Laufzeiten der Operationen Einfügen, Löschen und Zugriff sind von der Anzahl gespeicherter Elemente in der Regel (weitgehend) unabhängig. Allerdings erfordern hashbasierte Container einen zusätzlichen Aufwand, weil spezielle Hashwerte berechnet werden müssen, um diese Effizienz zu erreichen. Darum sind die hashbasierten Container etwas schwieriger zu verstehen als

Arrays und Listen. Die im Folgenden beschriebenen Grundlagen helfen dabei, die hashbasierten Container gewinnbringend einzusetzen. Zum leichteren Einstieg beginne ich mit einer Analogie aus dem realen Leben und einer vereinfachten Darstellung der Arbeitsweise, die im Verlauf der Beschreibung immer weiter präzisiert wird.

Analogie aus dem realen Leben

Hashbasierte Container kann man sich wie riesige Schrankwände mit nummerierten Schubladen vorstellen. In diesen Schubladen ist wiederum Platz für beliebig viele Sachen. Diese speziellen Schubladen werden in der Informatik auch als ***Bucket*** (zu deutsch: Eimer) bezeichnet. Soll ein Objekt in der Schrankwand abgelegt werden, so wird diesem eine Schubladennummer zugeteilt – wobei diese von den Eigenschaften (Attributen) des Objekts abhängt, das abgelegt werden soll. Wenn man später wieder auf Objekte zugreifen möchte, kann man dies mit der zuvor zugewiesenen Nummer tun. Zum leichteren Verwalten von Dingen in einer Schrankwand können wir uns intuitiv folgende Auswirkungen klarmachen:

1. Benutzt man immer nur ein und dieselbe Schublade, so quillt diese bald über und man findet seine Sachen nur mühselig wieder: Erschwerend kommt hinzu, dass der Inhalt einer Schublade des Öfteren komplett zu durchsuchen ist.
2. Verteilt man die Sachen relativ gleichmäßig über möglichst viele Schubladen, so kann man Sachen (nahezu) ohne Suchaufwand finden – die Kenntnis der richtigen Schublade vorausgesetzt.
3. Wenn kein gezielter Zugriff auf die korrekte Schublade erfolgt, etwa weil man sich in der Schublade irrt, so muss man im Extremfall alle Schubladen durchsuchen, um die gewünschten Sachen zu finden.

Die Analogie erleichtert das Verständnis der Anforderungen an hashbasierte Container und vor allem an die Methode `hashCode()`:

1. Mithilfe der Methode `hashCode()` eines Objekts wird, vereinfacht gesagt, die Nummer für die Schublade berechnet, in der sich das Objekt befinden soll. Auch wenn es möglich und zulässig ist, dass `hashCode()` für unterschiedliche Objekte den gleichen Wert berechnet, sollte man das möglichst vermeiden. Wenn nämlich für zwei unterschiedliche Objekte derselbe Hashwert berechnet wird, so kommt es zu einer sogenannten ***Kollision***. Verschiedene Objekte werden dann im gleichen Bucket gespeichert und erfordern eine möglicherweise aufwendigere Suche innerhalb des Buckets.
2. Zu einer gleichmäßigen Verteilung von Objekten auf Buckets kommt es, wenn die `hashCode()`-Methode für verschiedene Objekte möglichst verschiedene Werte zurückgeben. Das erreicht man am einfachsten, wenn man die Attribute selbst wieder auf Zahlen abbildet und mit Primzahlfaktoren multipliziert, wie wir es später noch sehen werden.

3. Aus dem letzten Punkt der Analogie kann man schließen, dass man die Nummern nicht verlieren oder verwechseln sollte. ***Um Schwierigkeiten zu vermeiden, empfiehlt es sich, dass sich der über die Methode `hashCode()` für ein Objekt berechnete Hashwert zur Laufzeit möglichst nicht ändert.*** Wenn sich allerdings die Grundlagen zur Berechnung ändern, kann man natürlich Änderungen am Hashwert nicht vermeiden. Man sollte sich jedoch der möglicherweise entstehenden Probleme bewusst sein (vgl. folgenden Praxistipp).

Hinweis: Auswirkungen bei Änderungen im berechneten Hashwert

Wie gerade angedeutet, ist es teilweise der Fall, dass sich der für ein Objekt berechnete Hashwert ändert, weil sich der Wert zur Berechnung benutzter Attribute ändert. Das hat aber Konsequenzen, die man kennen sollte: Liefern zu unterschiedlichen Zeiten die Berechnungen des Hashwerts für ein Objekt unterschiedliche Ergebnisse, so kann das Element nicht mehr über seinen zuvor berechneten Wert im Hachcontainer gefunden werden, weil es durch die Wertänderung an der falschen Stelle gesucht wird. Darüber hinaus kann eine Änderung im berechneten Hashwert zu der Inkonsistenz führen, dass mehrere gleiche Elemente in unterschiedlichen Buckets eingetragen werden, was ebenfalls verschiedenste andere Probleme mit sich bringt. ***Demnach ist es – wenn möglich – zu vermeiden, dass sich der berechnete Hashwert ändert***.

Realisierung in Java

Bis jetzt haben wir nicht explizit betrachtet, dass die Anzahl von Buckets beschränkt ist. Somit muss der durch `hashCode()` berechnete `int`-Wert auf die Anzahl der tatsächlich verfügbaren Buckets abgebildet werden. Die Speicherung der Buckets erfolgt als eindimensionales Array in einer sogenannten ***Hashtabelle***. Die Anzahl der dort vorhandenen Buckets wird ***Kapazität*** genannt. Jedes Bucket kann wiederum mehrere Elemente speichern. Dazu verwaltet es eine Liste, in der Elemente abgelegt werden.

Um für ein zu speicherndes Objekt die Bucket-Nummer, also den Index innerhalb der Hashtabelle, zu bestimmen, wird das in Abbildung 5-6 angedeutete Verfahren genutzt, das folgender Berechnungsabfolge entspricht:

$$Object \xrightarrow{\texttt{hashCode()}} Hashwert \xrightarrow{f(Hashwert)} Bucket\text{-}Nummer$$

Als Abbildungsfunktion $f(Hashwert)$ zur Bestimmung der Bucket-Nummer wird von den Hashcontainern des JDKs eine Modulo-Operation angewendet: $f(Hashwert) = Hashwert \ \% \ Kapazität$. Die vorgestellte Arbeitsweise hat gewisse Konsequenzen:

- Selbst wenn die für Objekte berechneten Hashwerte unterschiedlich sind, kann es aufgrund der Abbildungsfunktion f passieren, dass dieselbe Bucket-Nummer berechnet wird und es zu einer ***Kollision*** kommt: Verschiedene Objekte werden in dasselbe Bucket eingeordnet. Dort wird zur Speicherung eine Liste verwendet.

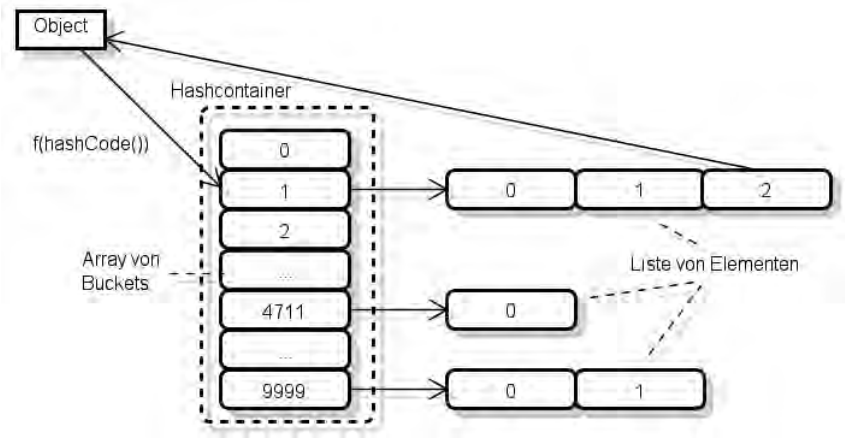


Abbildung 5-6 *Aufbau von hashbasierten Containern*

- Würden alle Objekte lediglich wenige unterschiedliche Bucket-Nummern (oder gar dieselbe) zurückliefern, so würde keine einigermaßen gleichmäßige Verteilung mehr erfolgen, sondern es käme zu einem Effekt, den man ***Clustering*** nennt. Damit bezeichnet man den Vorgang, dass in einigen Buckets sehr viele Elemente gespeichert werden und in anderen nahezu keine. Im Extremfall wird für alle Elemente die gleiche Bucket-Nummer berechnet. Der Hashcontainer würde dadurch auf eine einfache Liste reduziert werden – zusätzlich aber mit deutlichem Verwaltungsoverhead.

Ein Zugriff auf ein Element in einem Hashcontainer oder eine Suche danach erfordert durch den Hashcontainer ein zweistufiges Vorgehen:

1. Zunächst wird das Bucket bestimmt. Dazu wird die Methode `hashCode()` und die interne Abbildungsfunktion f des Hashcontainers benutzt.
2. Anschließend wird mit `equals(Object)` in der Liste des Buckets nach dem gewünschten Element gesucht.

Nach diesen grundsätzlichen Betrachtungen zur Arbeitsweise wollen wir uns konkret die Implikationen für Java-Klassen ansehen. Die Klasse `Object` stellt bekanntlich Defaultimplementierungen der Methoden `hashCode()` und `equals(Object)` bereit. Diese sind aber lediglich für sehr wenige Anwendungsfälle ausreichend. ***Darum sollten bei der Speicherung von Objekten eigener Klassen in hashbasierten Containern unbedingt immer deren Methoden `hashCode()` und `equals(Object)` konsistent zueinander überschrieben werden***.

Hinweis: Hashwerte in Mengen bzw. Schlüssel-Wert-Abbildungen

Der Hashwert wird mit der `hashCode()`-Methode entweder des Objekts selbst (bei Mengen) oder für Schlüssel-Wert-Abbildungen desjenigen Objekts, das als Schlüssel dient, berechnet. Damit ich beides im Anschluss nicht immer auseinanderhalten muss, beschreiben die folgenden Ausführungen zur einfacheren Darstellung den Ablauf für Mengen. Für Schlüssel-Wert-Abbildungen muss man sich abweichend davon nur gewahr sein, dass die `hashCode()`-Methode für Objekte des Typs des Schlüssels und zur späteren Suche im Bucket die `equals(Object)`-Methode derjenigen Klasse aufgerufen wird, die den Typ des Werts beschreibt.

Die Rolle von `hashCode()` beim Suchen

Konkretisieren wir die gerade gemachten Aussagen. Dazu wird das in Abschnitt 4.1.2 zur Demonstration der Methode `equals(Object)` verwendete Beispiel mit Objekten des Typs `Spielkarte` etwas abgewandelt: Statt einer Speicherung in einer `ArrayList<Spielkarte>` erfolgt diese nun in einem `HashSet<Spielkarte>`:

```
public static void main(final String[] args)
{
    final Collection<Spielkarte> spielkarten = new HashSet<>();
    spielkarten.add(new Spielkarte(Farbe.HERZ, 7));
    // PIK 8 einfügen
    spielkarten.add(new Spielkarte(Farbe.PIK, 8));
    spielkarten.add(new Spielkarte(Farbe.KARO, 9));

    // Finden wir eine PIK 8?
    final boolean gefunden = spielkarten.contains(new Spielkarte(Farbe.PIK, 8));
    System.out.println("gefunden = " + gefunden);
}
```

Listing 5.11 *Ausführbar als* **'SPIELKARTEINHASHSET'**

Wir erwarten, dass das Ergebnis einer Suche unabhängig davon ist, ob man Objekte in einem `HashSet<Spielkarte>` oder einer `ArrayList<Spielkarte>` speichert. Es stellt sich die Frage: Gefunden oder nicht gefunden? Prüfen wir den Wert der Variablen `gefunden`. Möglicherweise erleben wir dabei eine Überraschung. Die gesuchte »Pik 8« wird im Container nicht gefunden. Das ist merkwürdig, da die Methode `equals(Object)` der Klasse `Spielkarte` im oben genannten Abschnitt bereits korrekt implementiert wurde.

Ein kurzes Nachdenken bringt die Lösung: Beim Zugriff auf Hashcontainer wird zunächst durch Aufruf von `hashCode()` die Schublade berechnet, in der anschließend mit `equals(Object)` nach Objekten gesucht wird. Für die Klasse `Spielkarte` wurde die Methode `hashCode()` jedoch nicht überschrieben. Dadurch wird die Defaultimplementierung aus der Klasse `Object` ausgeführt, die typischerweise als `hashCode()` die Speicheradresse der Objektreferenz zurückliefert. Zur Suche wird aber ein neu erzeugtes Objekt verwendet, das zwar die gleiche Spielkarte darstellt, aber eine unterschied-

liche Referenz besitzt. Dadurch sind die für die beiden Spielkartenobjekte berechneten Hashwerte unterschiedlich und es wird in zwei unterschiedlichen Buckets gesucht.

Wir müssen also die `hashCode()`-Methode der Klasse `Spielkarte` korrigieren. Schauen wir dazu zunächst auf den `hashCode()`-Kontrakt.

Der `hashCode()`-Kontrakt

Die Methode `hashCode()` bildet den Objektzustand (besser: den möglichst unveränderlichen Teil davon) auf eine Zahl ab und wird in der Regel dazu benötigt, Objekte in hashbasierten Containern verarbeiten zu können. Die Methode `hashCode()` ist durch die JLS mit folgender Signatur definiert:

```
public int hashCode()
```

Eine Implementierung sollte folgende Eigenschaften erfüllen:[5]

- **Eindeutigkeit** – Während der Ausführung eines Programms sollte der Aufruf der Methode `hashCode()` für ein Objekt, sofern möglich (d. h. falls sich keine relevanten Attribute ändern), denselben Wert zurückliefern.
- **Verträglichkeit mit `equals()`** – Wenn die Methode `equals(Object)` für zwei Objekte `true` zurückgibt, dann muss die Methode `hashCode()` für beide Objekte denselben Wert liefern. Umgekehrt gilt dies nicht: Bei gleichem Hashwert können zwei Objekte per `equals(Object)` verschieden sein.

Daraus können wir folgende Hinweise zur Realisierung der Methode `hashCode()` herleiten: Zur Berechnung sollten diejenigen (möglichst ***unveränderlichen***) Attribute verwendet werden, die auch in `equals(Object)` zur Bestimmung der Gleichheit genutzt werden. Dadurch werden Änderungen des Hashwerts vermieden bzw. auf nur tatsächlich benötigte Fälle eingeschränkt. Die Verträglichkeit mit `equals(Object)` ist automatisch dadurch gegeben, dass nur diejenigen Attribute zur Berechnung genutzt werden (oder auch nur ein Teil davon), die in `equals(Object)` verglichen werden.

Fallstricke bei der Implementierung von `hashCode()` Ein typischer Fehler ist, dass die Methode `equals(Object)` überschrieben wird, die Methode `hashCode()` jedoch nicht. Dadurch wird in der Regel die Zusicherung verletzt, die besagt, dass für zwei laut `equals(Object)` gleiche Objekte auch der gleiche Wert durch `hashCode()` berechnet wird.

Auch sieht man Realisierungen von `hashCode()`, die veränderliche Attribute zur Berechnung verwenden. Das kann in einigen Fällen korrekt sein, aber manchmal Probleme bereiten.[6] Wir hatten bereits angesprochen, dass wir Objekte in Hashcontainern nicht mehr wiederfinden, wenn nach einer Änderung des Hashwerts im falschen Bucket

[5] Werden diese nicht eingehalten, sollte dies unbedingt in der Javadoc vermerkt werden.

[6] Die in Eclipse eingebaute Automatik aus dem Menü SOURCE –> GENERATE HASHCODE() AND EQUALS()… erzeugt eine `hashCode()`-Methode, die potenziell zu viele Attribute nutzt.

gesucht wird. Aufgrund dessen sollte man einen kritischen Blick auf die Zusammensetzung der zur `hashCode()`-Berechnung verwendeten Attribute werfen und versuchen, bevorzugt unveränderliche Attribute zu nutzen. Damit kann man vermeiden, dass sich der berechnete Hashwert bei jeder Modifikation von Attributen des Objekts ändert.

Realisierung von `hashCode()` für die Klasse `Spielkarte` Zur Korrektur der Klasse `Spielkarte` implementieren wir dort die Methode `hashCode()`. Dazu werfen wir einen Blick auf die Methode `equals(Object)`:

```
@Override
public boolean equals(Object other)
{
    if (other == null) // Null-Akzeptanz
        return false;
    if (this == other) // Reflexivität
        return true;
    if (this.getClass() != other.getClass()) // Typgleichheit
        return false;

    // int mit Wertevergleich, Enum mit equals()
    final Spielkarte karte = (Spielkarte) other;
    return this.wert == karte.wert && this.farbe.equals(karte.farbe);
}
```

Bei der Realisierung der Methode `hashCode()` dürfen in diesem Fall maximal die Attribute `wert` und `farbe` zur Berechnung verwendet werden. Die Verteilung auf die Buckets sollte möglichst gut gestreut werden. Das erreicht man, indem man mit Primzahlen als Multiplikatoren arbeitet: Jeder Spielkartenwert wird mit einem Primzahlfaktor multipliziert und dann der Hashwert der Farbe hinzu addiert, etwa wie folgt:

```
@Override
public int hashCode()
{
    final int PRIME = 37;
    return this.wert * PRIME + this.farbe.hashCode();
}
```

Man erreicht zwar so eine gleichmäßigen Verteilung, aber die Implementierung der Methode `hashCode()` wird doch schnell unübersichtlich und kompliziert. Als weitere Anforderung neben der gleichmäßigen Verteilung sollten die Hashfunktionen recht einfach und effizient zu berechnen sein, da sie unter Umständen sehr oft aufgerufen werden. Um sich darüber nicht allzu viele Gedanken machen zu müssen, ist der Einsatz einer passenden Utility-Klasse wünschenswert.

Entwurf einer Utility-Klasse zur Berechnung von `hashCode()`

Wir nutzen das in Abschnitt 4.2.4 gewonnene Wissen über Zahlen und Operationen und erstellen damit eine Utility-Klasse `HashUtils` mit überladenen Methoden `calcHashCode()` für primitive Datentypen und für Objektreferenzen folgendermaßen:

```
public final class HashUtils
{
    public static final int PRIME = 31;

    private HashUtils()
    {}

    public static final int calcHashCode(final int hash, final boolean input)
    {
        return PRIME * hash + (input ? 1 : 0);
    }

    public static final int calcHashCode(final int hash, final int input)
    {
        return PRIME * hash + input;
    }

    public static final int calcHashCode(final int hash, final long input)
    {
        return PRIME * hash + (int) (input ^ (input >>> 32));
    }

    public static final int calcHashCode(final int hash, final float input)
    {
        return calcHashCode(hash, Float.floatToIntBits(input));
    }

    public static final int calcHashCode(final int hash, final double input)
    {
        return calcHashCode(hash, Double.doubleToLongBits(input));
    }

    public static final int calcHashCode(final int hash, final Object input)
    {
        return (input == null) ? 0 : PRIME * hash + input.hashCode();
    }
}
```

Diese Realisierung orientiert sich an den von Joshua Bloch in seinem Buch »Effective Java« [8] entwickelten Ideen zur Implementierung der Methode `hashCode()`. Erwähnenswert ist, dass aufgrund des Rückgabetyps `int` bei der Berechnung verschiedene Konvertierungen und Wertebereichseinschränkungen durchgeführt werden müssen.

Einsatz einer Utility-Klasse Um für die Klasse `Spielkarte` die Berechnungen in `hashCode()` zu vereinfachen, nutzen wir die zuvor erstellte Utility-Klasse `HashUtils`. Man ruft einfach die entsprechenden `calcHashCode()`-Methoden für die zu verwendenden Attribute hintereinander auf:

```
@Override
public int hashCode()
{
    int hash = 17;
    hash = HashUtils.calcHashCode(hash, farbe);
    hash = HashUtils.calcHashCode(hash, wert);
    return hash;
}
```

Vereinfachung mit JDK 7 In der mit Java 7 eingeführten Klasse `Objects` enthält das JDK verschiedene nützliche Funktionalitäten. Hier bietet sich der Einsatz der Methode `hash()` an, um die Berechnung einfach und übersichtlich zu schreiben:

```
@Override
public int hashCode()
{
   return Objects.hash(this.wert, this.farbe);
}
```

Realisierung von `hashCode()` für die Klasse `Person` Mit dem bis hierher erlangten Wissen können wir nun auch die Klasse `Person` um eine passende, gut verständliche Realisierung von `hashCode()` erweitern:

```
@Override
public int hashCode()
{
    return Objects.hash(this.name, this.birthday, this.city);
}
```

Füllgrad (Load Factor)

Wir besitzen nun das Wissen, um `hashCode()` so zu implementieren, dass Kollisionen weitestgehend vermieden werden, indem eine möglichst gleichmäßige Verteilung der zu speichernden Elemente erfolgt. Das hängt einerseits von der gewählten Hashfunktion sowie andererseits von der Anzahl verfügbarer Buckets und gespeicherter Elemente ab: Werden fortlaufend immer mehr Elemente in einem hashbasierten Container gespeichert, so wächst die Wahrscheinlichkeit für und die Anzahl von Kollisionen. Ähnlich wie die Klasse `ArrayList<E>` führen auch Hashcontainer eine automatische Größenanpassung der Hashtabelle, d. h. eine Erweiterung um Buckets, durch, wenn die Anzahl gespeicherter Elemente einen Grenzwert übersteigt. Um zu bestimmen, ob eine Größenanpassung nötig ist, betrachtet man nicht den Inhalt aller Buckets im Einzelnen, sondern nutzt eine einfachere, aber effektive Variante: Dabei hilft als Kenngröße der sogenannte ***Füllgrad***, auch ***Load Factor*** genannt, der sich aus dem Quotienten der Anzahl gespeicherter Elemente und der Anzahl der Buckets ergibt. Dieser Wert beschreibt, wie voll der Hashcontainer werden darf, bis es zu einer Größenanpassung kommt. Bis zu einem Füllgrad von 75% sind Kollisionen erfahrungsgemäß relativ unwahrscheinlich (vgl. Javadoc-Dokumentation der Klassen `Hashtable<K,V>` und `HashMap<K,V>`).

Die Hashtabelle wird erweitert, wenn folgende Bedingung zwischen Füllgrad, Anzahl gespeicherter Elemente und der Kapazität der Hashtabelle erfüllt ist:

$$\textit{maximaler Füllgrad} * \textit{Kapazität} \geq \textit{Anzahl Elemente}$$

Wenn man zu dem Zeitpunkt, an dem die Hashtabelle angelegt wird, die ungefähre Anzahl der später zu speichernden Elemente kennt, kann man nachträgliche Größenanpassungen oftmals vermeiden, indem man die initiale Kapazität als Quotient aus der

Anzahl der Elemente und dem maximal akzeptierten Füllgrad passend wählt:

$$\textit{initiale Kapazität} = \textit{Anzahl Elemente}/\textit{maximaler Füllgrad}$$

Für 1.000 Elemente ergibt sich bei einem maximalen Füllgrad von 75% die initiale Kapazität wie folgt: $\textit{initiale Kapazität} = 1.000/0,75 \approx 1.333$. Bei der Konstruktion eines Hashcontainers kann man den berechneten Wert der initialen Kapazität angeben, wobei dieser rund 1,3-mal größer als die geplante Anzahl der zu verwaltenden Elemente sein sollte. Bei einem angenommenen idealen Füllfaktor von 75% bedeutet dies, dass immer etwa 25% Kapazität unbelegt bleiben.

Der maximal erlaubte Füllgrad stellt damit eine Stellschraube von Hashcontainern dar, die sich auf Speicherplatz und Zugriffszeit auswirkt. Je geringer der Wert des maximal erlaubten Füllgrads, desto geringer ist die Wahrscheinlichkeit für Kollisionen. Damit ist der Zugriff schneller, als wenn es Kollisionen gibt. Allerdings geht dies zulasten des benötigten Speichers und erhöht den Anteil unbenutzter Buckets. Umgekehrt »verschwendet« ein maximal erlaubter Füllgrad größer als 75% zwar weniger Speicher, jedoch steigt die Wahrscheinlichkeit für Kollisionen. Dadurch verschlechtern sich die Zugriffszeiten auf die Elemente.

Auswirkungen von Größenanpassungen

Werden mehr Elemente in einem Hashcontainer gespeichert als zunächst erwartet, und übersteigt der momentane Füllgrad die durch den maximal erlaubten Füllgrad angegebene Schwelle, so wird die Hashtabelle automatisch vergrößert. Es stehen daraufhin mehr Buckets zur Verfügung. Im Gegensatz zu Array-basierten Listen, die neue Daten nach einem solchen Vergrößerungsschritt einfach am Ende anfügen können, ist der Sachverhalt für hashbasierte Container komplizierter. ***Nach einer erfolgten Größenanpassung muss die Hashtabelle vollständig neu organisiert werden***, da die Abbildungsfunktion für zuvor gespeicherte Werte nicht mehr korrekt arbeitet: *Die Modulo-Operation liefert nun in der Regel andere Werte als zuvor*. Für jedes Element in der Hashtabelle muss das entsprechende aufnehmende Bucket neu ermittelt werden. Diesen Umsortierungsvorgang nennt man ***Rehashing***. Da jedes gespeicherte Element betrachtet werden muss, ist dieser Vorgang relativ aufwendig. Als Optimierung wird zu jedem Element dessen zuvor über die Methode `hashCode()` berechnete Wert zwischengespeichert. Dieser ändert sich bei einem Rehashing nicht. Dadurch werden zusätzliche Performance-Einbußen durch die Neuberechnung der Hashwerte durch Aufrufe von `hashCode()` vermieden. Das neue, aufnehmende Bucket kann auf Basis des zwischengespeicherten Hashwerts bestimmt werden. Das Rehashing kostet Rechenzeit, macht spätere Zugriffe aber wieder performanter, weil dadurch weniger Kollisionen auftreten.

Neben dem Rehashing gibt es Folgendes zu bedenken: Falls in einem Hashcontainer irgendwann einmal sehr viele Elemente gespeichert wurden, so kam es höchstwahrscheinlich zu einigen Vergrößerungsschritten und Rehashing-Vorgängen. Werden später viele Elementen gelöscht, wird die Größe der Hashtabelle nicht automatisch verkleinert und der Speicher bleibt (unnütz) belegt. Schlimmer noch: ***Im Ge-***

gensatz zu Listen gibt es für die Hashcontainer kein Pendant zu `trimToSize()`, das es nach einer mittlerweile hinfälligen Expansion erlaubt, den Speicherverbrauch zu beschneiden. Als Abhilfe kann man einen neuen Hashcontainer mit passend gewählter Größe anlegen, der mit dem Inhalt des bisherigen gefüllt wird.

Um wieder das Beispiel einer Schrankwand zu bemühen: Analog zu den Vergrößerungen wird diese um einen Anbausatz und damit weiteren Stauraum ergänzt, wenn der Platz eng wird. Ein Umräumvorgang sorgt für eine bessere Verteilung der Sachen auch auf die neuen Schubladen und erleichtert eine spätere Suche, da wieder mehr Ordnung herrscht und in jeder Schublade weniger Dinge gelagert sind. Bezogen auf die Speicherverschwendung gilt in etwa folgende Analogie: Nach einem Frühjahrsputz sind beispielsweise mehr als die Hälfte aller Schubladen des Schranks leer. Der Anbausatz wird aber nicht abmontiert, sondern nimmt dann einfach nur noch Platz weg.

5.1.8 Grundlagen automatisch sortierender Container

Für einige Anwendungsfälle ist es praktisch, wenn die in einer Containerklasse verwalteten Daten sortiert vorliegen. Bekanntermaßen gibt es die Containerklassen `TreeSet<E>` bzw. `TreeMap<K,V>`, die automatisch die Sortierung von Elementen ohne weiteren Implementierungsaufwand im Applikationscode herstellen. Für Arrays und Listen gibt es so etwas im JDK nicht. Um diese sortiert zu halten, wird ein manueller Schritt notwendig. Hierbei unterstützen die Methoden `sort()` aus den Utility-Klassen `Arrays` und `Collections` aus dem Package `java.util` (vgl. Abschnitt 5.2.2).

Aber unabhängig von automatischer oder manueller Sortierung muss immer eine Ordnung festgelegt werden, um bei Vergleichen von Objekten »kleiner« bzw. »größer« oder »gleich« ausdrücken zu können. Das kann man mithilfe von Implementierungen der Interfaces `Comparable<T>` und `Comparator<T>` beschreiben:

- **Natürliche Ordnung und `Comparable`** – Sofern zu speichernde Objekte das Interface `Comparable<T>` erfüllen, können sie darüber ihre Ordnung, d. h. ihre Reihenfolge untereinander, beschreiben. Diese Reihenfolge wird auch als ***natürliche Ordnung*** bezeichnet, da sie durch die Objekte selbst bestimmt wird.[7]
- **Weitere Ordnungen und `Comparator`** – Teilweise benötigt man zusätzlich zur natürlichen Ordnung weitere oder alternative Sortierungen, etwa wenn man Personen nicht nach Nachname, sondern alternativ nach Vorname und Geburtsdatum ordnen möchte. Diese ergänzenden Sortierungen können mithilfe von Implementierungen des Interface `Comparator<T>` festgelegt werden. Dadurch lassen sich von der natürlichen Ordnung abweichende Sortierungen für Objekte einer Klasse realisieren und auch Klassen sortieren, für die keine natürliche Ordnung definiert ist, weil sie das Interface `Comparable<T>` nicht implementieren.

[7] Über das »natürlich« kann man sich trefflich streiten, weil es nur um die Vorgabe einer Reihenfolge durch die Implementierung geht und die Ordnung auch unintuitiv oder unerwartet sein kann und sich somit möglicherweise sogar eher unnatürlich anfühlt.

Sortierungen und das Interface `Comparable`

Oftmals besitzen Werte oder Objekte eine natürliche Ordnung: Das gilt etwa für Zahlen und Strings. Für komplexe Typen ist die Aussage »kleiner« bzw. »größer« nicht immer sofort ersichtlich, lässt sich aber selbst definieren.

Dazu erlaubt das Interface `Comparable<T>` typsichere Vergleiche und deklariert die Methode `compareTo(T)` folgendermaßen:

```
public interface Comparable<T>
{
    public int compareTo(T o);
}
```

Das Vorzeichen des Rückgabewerts bestimmt die Reihenfolge der Sortierung:

- $= 0$: Der Wert 0 bedeutet Gleichheit des aktuellen und des übergebenen Objekts.
- < 0: Das aktuelle Objekt ist kleiner als das übergebene Objekt.
- > 0: Das aktuelle Objekt ist größer als das übergebene Objekt.

Diverse Klassen im JDK (alle Wrapper-Klassen, `String`, `Date` usw.) implementieren das Interface `Comparable<T>` und sind damit automatisch sortierbar.

Implementieren von `compareTo()` in eigenen Klassen Wie man das Interface `Comparable<T>` für eigene Klassen implementiert, zeige ich nun für die folgende Klasse `Person`. Dort wird anstatt des Geburtsdatums als `Date`-Objekt das Alter bewusst als primitiver Typ gespeichert, um einige Varianten bei der Realisierung des Interface `Comparable<Person>` zu verdeutlichen:

```
public final class Person implements Comparable<Person>
{
    private final String name;
    private final String city;
    private final int    age;

    public Person(final String name, final String city, final int age)
    {
        this.name = Objects.requireNonNull(name, "name must not be null");
        this.city = Objects.requireNonNull(city, "city must not be null");
        this.age = age;
    }
```

Eine erste Implementierung von `compareTo(Person)` könnte die natürliche Ordnung für `Person`-Objekten (eher leicht unintuitiv) ausschließlich über deren Namen realisieren:

```
@Override
public int compareTo(final Person otherPerson)
{
    Objects.requireNonNull(otherPerson, "otherPerson must not be null");

    return getName().compareTo(otherPerson.getName());
}
```

Wir benötigen eine `null`-Prüfung lediglich für den Methodenparameter, nicht aber für die Attribute, da wir für diese im Konstruktor `null`-Werte ausgeschlossen haben. Zudem nutzen wir, dass die Klasse `String` das Interface `Comparable<String>` erfüllt.

Betrachten wir den Einsatz unserer Methode `compareTo(Person)` und nehmen dazu an, eine Kundenliste `customers` enthielte etwa folgende Einträge:

```
customers.add(new Person("Müller", "Bremen", 27));
customers.add(new Person("Müller", "Kiel", 37));
```

Nutzen wir die obige Umsetzung, so werden laut `compareTo(Person)` alle Objekte vom Typ `Person` mit gleichem Namen als gleich angesehen. Dass dies keine wirklich gelungene Realisierung einer natürlichen Ordnung für Personen darstellt, wird nach kurzer Überlegung offensichtlich: Herr Müller aus Kiel ist nicht Herr Müller aus Bremen. Wie geht es also besser? Einen guten Anhaltspunkt zur Verbesserung stellt oftmals die Methode `equals(Object)` bzw. die dort zum Vergleich verwendeten Attribute dar. Nachfolgend werden hier neben dem Namen zusätzlich die Attribute `city` und `age` zur Gleichheitsprüfung herangezogen:

```
public boolean equals(final Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;

    final Person other = (Person) obj;
    return name.equals(other.name) && city.equals(other.city) &&
           age == other.age;
}
```

Vorgehen zur Implementierung von `compareTo()` Im Allgemeinen kann man sich beim Implementieren von `compareTo(T)` an einer bestehenden Realisierung der Methode `equals(Object)` der jeweiligen Klasse orientieren. Alle dort verglichenen Attribute sind in der Regel auch für die Sortierung gemäß der natürlichen Ordnung relevant.[8] Diese Attribute werden wie folgt verglichen:

1. Referenztypen, die das Interface `Comparable<T>` implementieren, verwenden deren `compareTo(T)`-Methoden.
2. Für primitive Datentypen lassen sich die Vergleichsoperatoren '<', '=' und '>' einsetzen, nachfolgend für einen Vergleich des Attributs `age`:[9]

[8] Allerdings ist die Reihenfolge für den Vergleich unbedingt zu beachten. Während diese für `equals(Object)` keinen Einfluss auf das Ergebnis besitzt, macht es für `compareTo(T)` möglicherweise einen großen Unterschied: Es ist entscheidend, ob erst die Namen und dann das Alter oder erst das Alter und dann die Namen verglichen werden.

[9] Für die Typen `float` und `double` sind Rundungsfehler zu bedenken (vgl. Abschnitt 4.1.2).

```
int result = 0;
if (this.getAge() < otherPerson.getAge())
{
    result = -1;
}
if (this.getAge() > otherPerson.getAge())
{
    result = 1;
}
```

Statt die drei Fälle größer, kleiner und gleich selbst abzufragen und den passenden Rückgabewert bereitzustellen, ist es sinnvoller, die Methoden `compare()` der jeweiligen Wrapper-Klasse zu nutzen, weil diese einem Arbeit abnehmen und der Vergleich wie folgt kürzer und klarer notiert werden kann:

```
int result = Integer.compare(this.getAge(), otherPerson.getAge());
```

3. Für alle Attribute anderen Typs muss der Vergleich selbst implementiert werden. Wenn man die Klasse des Attributs im Zugriff hat, kann man diese derart erweitern, dass sie das Interface `Comparable<T>` erfüllt und in der Realisierung die gewünschten Attribute vergleicht. Hat man eine Klasse jedoch nicht im Zugriff oder soll/darf diese nicht verändert werden, so muss der Vergleich der relevanten Attribute dieser Klasse gemäß der Schritte 1 und 2 selbst programmiert werden.

Konsistenz von `compareTo()` und `equals()` Die Methoden `compareTo(T)` und `equals(Object)` sollten so implementiert werden, dass `x.compareTo(y)` genau dann den Wert 0 zurückgibt, wenn der Vergleich `x.equals(y)` den Wert `true` liefert. Wird gegen diese Regel verstoßen, so empfiehlt es sich, dies im Javadoc zu vermerken.

Für unser Beispiel ist die Forderung nicht eingehalten, weil `compareTo(Person)` schwächer prüft als `equals(Object)`. Um nicht für Verwirrung beim Einsatz zu sorgen, korrigieren wir die Implementierung dahingehend, dass `compareTo(Person)` auch die Attribute `city` und `age` beim Vergleich heranzieht:

```
@Override
public int compareTo(final Person otherPerson)
{
    Objects.requireNonNull(otherPerson, "otherPerson must not be null");

    int ret = getName().compareTo(otherPerson.getName());
    if (ret == 0)
    {
        ret = getCity().compareTo(otherPerson.getCity());
    }
    if (ret == 0)
    {
        ret = Integer.compare(getAge(), otherPerson.getAge()));
    }
    return ret;
}
```

Falls man in den Methoden `compareTo(T)` und `equals(Object)` dieselben Attribute nutzt, lässt sich Sourcecode-Duplikation vermeiden, indem man in `equals(Object)` die Methode `compareTo(T)` aufruft:

```
@Override
public boolean equals(final Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;

    final Person other = (Person) obj;
    return compareTo(other) == 0;       // Vergleich mittels compareTo(Person)
}
```

Diese Art der Realisierung vermeidet einerseits Konsistenzprobleme zwischen beiden Methoden und führt andererseits dazu, dass die Vergleichslogik nur einmal in der Methode `compareTo(T)` realisiert wird. Dies ist wiederum hilfreich, wenn Änderungen an der Klasse erfolgen, etwa Attribute hinzugefügt werden. Schnell wird in einem solchen Fall übersehen, beide Implementierungen anzupassen. Auch hier erkennt man den Vorteil, eine Funktionalität möglichst nur einmal zu realisieren. Andrew Hunt und David Thomas beschreiben dies in ihrem Buch »Der Pragmatische Programmierer« [45] als das sogenannte DRY-Prinzip (**D**on't **R**epeat **Y**ourself).

Entwicklung: `compareTo(T)` basierend auf `equals(Object)`

Häufig entwickelt man bei einer Neuimplementierung einer Klasse zunächst eine `equals(Object)`-Methode. Wird im Verlauf der Entwicklung eine natürliche Ordnung durch Erfüllen des Interface `Comparable<T>` erforderlich, so bietet es sich oftmals an, `equals(Object)` durch Aufruf von `compareTo(T)` zu realisieren und die Vergleichslogik in `compareTo(T)` zu verlagern.

Sortierungen und das Interface `Comparator`

Wir haben zur Beschreibung der natürlichen Ordnung das Interface `Comparable<T>` kennengelernt. Darüber lässt sich lediglich *eine* spezielle Sortierung beschreiben. In vielen Anwendungsfällen sind weitere Sortierungen wünschenswert, z. B. möchte man in Tabellen häufig nach jeder beliebigen Spalte sortieren können. Dies wird durch den Einsatz der im Folgenden beschriebenen ***Komparatoren*** möglich. Der Vorteil dieses Vorgehens ist, dass man Anwendungsklassen nicht mit Sortierfunktionalität überfrachtet, sondern diese in ***eigenständigen Vergleichsklassen*** definiert wird. Dazu müssen diese das Interface `Comparator<T>` erfüllen und die gewünschte Sortierung realisieren. Als Hinweis sei angemerkt, dass die dafür benötigten Attribute bzw. deren Zugriffsmethoden in ihrer Sichtbarkeit möglicherweise eingeschränkt und im `Comparator<T>` nicht zugreifbar sind. Mit `Comparable<T>` hat man immer Zugriff auf alle Attribute.

Das Interface `Comparator` Das Interface `Comparator<T>` beschreibt einen Baustein zum Vergleich von Objekten des Typs `T`. Hierfür wird die Methode `int compare(T, T)` angeboten, die dazu dient, zwei beliebige Objekte des Typs `T` miteinander zu vergleichen:

```
public interface Comparator<T>
{
    int compare(T o1, T o2);
}
```

Über den Rückgabewert wird die Reihenfolge der Sortierung bestimmt. Es gilt:

- $= 0$: Der Wert 0 bedeutet Gleichheit der beiden Objekte `o1` und `o2`.
- < 0: Das erste Objekt `o1` ist als kleiner als das zweite Objekt `o2` anzusehen.
- > 0: Bei positiven Zahlen ist das erste Objekt `o1` als größer anzusehen.

Grundgerüst eines einfachen Komparators Stellen wir uns vor, unsere Aufgabe bestünde darin, eine Liste mit `Person`-Objekten nach verschiedenen Kriterien zu sortieren, etwa nach Name, Wohnort oder Alter. Der grundsätzliche Aufbau einer Realisierung für Komparatoren für einen Typ `T` folgt immer einem gleichen Schema: In der `compare(T, T)`-Methode werden die benötigten Vergleiche durchgeführt. Einen Vergleich auf Namen realisiert man beispielsweise wie folgt:

```
public final class PersonNameComparator implements Comparator<Person>
{
    public int compare(final Person person1, final Person person2)
    {
        Objects.requireNonNull(person1, "person1 must not be null");
        Objects.requireNonNull(person2, "person2 must not be null");

        return person1.getName().compareTo(person2.getName());
    }
}
```

Diskussion: Konsistenz von `compare()` und `equals()`

Obwohl im `Comparator<T>` im Javadoc von `compare(T, T)` empfohlen wird, dass `(compare(x, y) == 0) == (x.equals(y))` gelten sollte, ist dies in der Praxis häufig nicht der Fall. Eine entsprechende Forderung wurde bereits bezüglich des Interface `Comparable<T>` aufgestellt. Dabei gibt es zwischen beiden Forderungen die folgenden, entscheidenden Unterschiede.

Unterschiede der Forderungen von `compareTo()` und `compare()`

Realisierungen des Interface `Comparable<T>` sind meistens bijektiv, d. h., es existiert eine »Genau-dann-wenn«-Beziehung: Aus einer Gleichheit bezüglich `compareTo(T)` folgt eine Gleichheit bezüglich `equals(Object)` und auch umgekehrt: Ergibt `equals(Object)` Gleichheit, so gilt dies auch für `compareTo(T)`.

Realisierungen des Interface `Comparator<T>` sind dagegen oftmals injektiv, d. h., es wird eine »Daraus-folgt«-Beziehung beschrieben: Aus einer Gleichheit gemäß `equals(Object)` kann man (in der Regel) auf eine Gleichheit bezüglich `compare(T, T)` schließen. Aus einer Gleichheit gemäß `compare(T, T)` folgt jedoch meist *keine* Gleichheit bezüglich `equals(Object)`. Anhand von Komparatoren für `Person`-Objekte, die die Attribute `name` bzw. `city` vergleichen, kann man sich dies verdeutlichen: Zwei gleichnamige oder in der gleichen Stadt wohnende Personen werden über die jeweilige Realisierung von `compare(Person, Person)` als gleich angesehen, für `equals(Object)` gilt das logischerweise nicht, da hier noch weitere Attribute wie z. B. Geburtstag oder Größe verglichen werden.

Hintergrundwissen: Arbeitsweise sortierender Datenstrukturen

Zum besseren Verständnis der Arbeitsweise der Containerklassen `TreeSet<E>` bzw. `TreeMap<K,V>` betrachten wir ein `TreeSet<Long>`, das initial die Werte 1, 2 und 3 speichert und in das anschließend die Werte 4, 5 und 6 eingefügt werden. Bevor ich auf Details beim Einfügen eingehe, erläutere ich kurz die zugrunde liegende Datenstruktur.

Es wird ein sogenannter ***binärer Baum*** genutzt, der sich dadurch auszeichnet, dass es einen speziellen Startknoten (***Wurzel*** genannt) gibt, der maximal einen direkten linken und einen direkten rechten Kindknoten besitzt. Diese Kindknoten können wiederum jeweils maximal zwei direkte Kindknoten haben, dies aber beliebig fortgesetzt, sodass ein Knoten beliebig viele Nachfahren besitzen kann. Die ***Tiefe des Baums*** ist als die maximale Anzahl der Knoten von der Wurzel bis zu einem Knoten ohne Nachfahren (auch ***Blatt*** genannt) definiert. Per Definition werden ausgehend von der Wurzel jeweils in den linken Kindknoten diejenigen Elemente eingefügt, die in der Wertebelegung ihrer Attribute als kleiner als der momentane Knoten anzusehen sind. Analog gilt dies für »größere« Elemente, die im rechten Teilbaum gespeichert werden.[10]

Für unser Beispiel des `TreeSet<Long>` ergibt sich mit diesem Wissen und den initialen Werten ein Baum, dessen Wurzelknoten den Wert 2 hat und einen linken sowie rechten Nachfolger mit den Werten 1 bzw. 3. Um die Arbeitsweise beim Einfügen von Elementen zu verdeutlichen, werden dann sukzessive die Elemente 4, 5 und 6 eingefügt. Bei Einfügeoperationen (und selbstverständlich auch bei den hier nicht gezeigten Löschoperationen) wird einerseits immer die gewünschte Sortierung hergestellt und andererseits durch ***Balancierung*** (Höhenausgleich der Teilbäume) für eine ausgeglichene Verteilung der innerhalb der Datenstruktur gespeicherten Elemente gesorgt. Die Auswirkungen verschiedener Aktionen auf den Baum zeigt Abbildung 5-7.

[10] Schnell stellt sich die Frage: Was ist mit gleichen Werten? In den baumbasierten Datenstrukturen `TreeSet<E>` und `TreeMap<K,V>` werden nicht mehrere gleiche Werte gespeichert, sondern es existiert jeweils nur ein derartiger Eintrag. Für Mengen ist dies per Definition so, für Maps gilt dies, da hier die Eindeutigkeit von Schlüsseln gefordert wird. Versucht man trotzdem einen gleichen Wert zu speichern, so wird der alte Eintrag ersetzt, also für `TreeMap<K,V>` ein neuer Wert für den Schlüssel eingetragen.

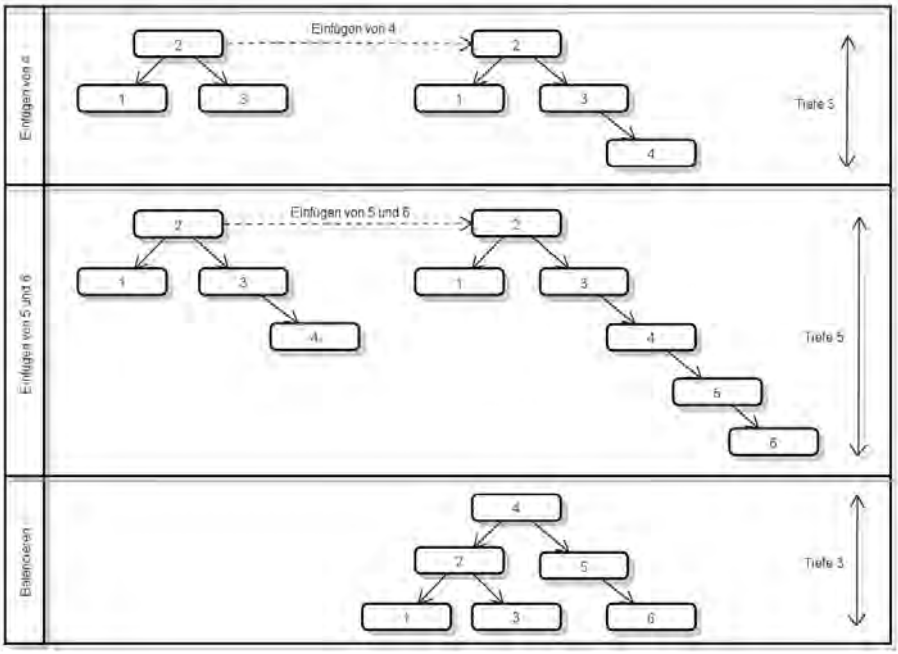


Abbildung 5-7 *Arbeitsweise eines balancierten Baums*

Da die Elemente 4, 5 und 6 größer als die Wurzel sind, werden sie zunächst im rechten Teilbaum einsortiert. Durch Einfügen des Werts 4 entsteht lediglich eine Höhendifferenz von 1 zwischen dem linken und dem rechten Teilbaum. Eine solche Differenz führt nicht zu einem Ausgleichsvorgang, weil sich eine derartige Dysbalance nicht in jedem Fall vermeiden lässt – beispielsweise ist bei zwei gespeicherten Elementen immer ein Teilbaum leer. Durch das Einfügen der Werte 5 und 6 im obigen Beispiel ergibt sich allerdings eine Unausgewogenheit in der Tiefe der Teilbäume, deren Differenz größer als eins ist. Wie in der Abbildung ersichtlich, erhält man nach einem Einfügen möglicherweise fast so etwas wie eine lineare Liste. Um performante Suchen zu gewährleisten, ist es das Ziel, die Tiefe minimal zu halten, den Baum also möglichst auszugleichen. Dies wird durch ein Rotieren der Knoten erreicht, wodurch auch eine Degeneration vermieden wird. Hier wird der Knoten mit dem Wert 4 zur neuen Wurzel.

Durch die beschriebenen Ausgleichsvorgänge wird die Tiefe des Baums nahezu identisch für den linken und den rechten Teilbaum gehalten – genauer: Die Höhendifferenz überschreitet nie den Wert eins. Damit bleibt die maximale Tiefe immer logarithmisch zur Anzahl der im Baum gespeicherten Elemente. Die Ausgeglichenheit sorgt dafür, die maximale Suchgeschwindigkeit auf logarithmische Komplexität zu begrenzen. Das ermöglicht sehr performante Suchvorgänge: Bei 1.000 Elementen beträgt die Tiefe 10 und definiert damit auch die maximale Anzahl an Suchschritten bis zum Auffinden des gesuchten Elements bzw. zum Erkennen, dass kein solches existiert. Selbst bei 1 Million gespeicherter Elemente sind dadurch maximal 20 Schritte notwendig.

5.1.9 Die Methoden `equals()`, `hashCode()` und `compareTo()` im Zusammenspiel

In C++ gibt es das sogenannte »***Law of the Big Three***«: Wenn für eine Klasse entweder ein Copy-Konstruktor, ein Zuweisungsoperator oder ein Destruktor benötigt wird, so sind in der Regel alle drei zu implementieren. Eine ähnliche Aussage gilt in Java für die drei Methoden `equals(Object)`, `hashCode()` und `compareTo(T)`. Hier ist es aber vor allem wichtig, diese Methoden konsistent zueinander zu implementieren, wobei `compareTo(T)` nicht in jedem Fall angeboten werden muss. Wenn es aber existiert, dann sollte es konsistent zu `equals(Object)` sein.[11] Beachtet man die Forderung nach Konsistenz nicht, kann es zu Fehlern kommen, die sich nur schwierig reproduzieren lassen und sich in merkwürdigem Programmverhalten äußern.

Betrachten wir dies anhand der Verwaltung einiger Objekte der folgenden Klasse `SimplePerson` mithilfe der Datenstrukturen `HashSet<SimplePerson>` und `TreeSet<SimplePerson>`. Die Klasse `SimplePerson` implementiert das Interface `Comparable<SimplePerson>` und ist wie folgt definiert:

```
private static class SimplePerson implements Comparable<SimplePerson>
{
    private final String name;

    SimplePerson(final String name)
    {
        this.name = name;
    }

    @Override
    public int compareTo(final SimplePerson other)
    {
        return name.compareTo(other.name);
    }
}
```

Das folgende Listing zeigt, wie zwei inhaltlich gleiche `SimplePerson`-Objekte erzeugt und per `add(SimplePerson)` in einem `HashSet<SimplePerson>` und einem `TreeSet<SimplePerson>` gespeichert werden. Anschließend ermitteln wir durch Aufruf der Methode `size()` die Anzahl der gespeicherten Elemente im jeweiligen Container:

```
public static void main(final String[] args)
{
    final Set<SimplePerson> hashSet = new HashSet<>();
    hashSet.add(new SimplePerson("Test"));
    hashSet.add(new SimplePerson("Test"));
    System.out.println("HashSet size = " + hashSet.size()); // Size = 2

    final Set<SimplePerson> treeSet = new TreeSet<>();
    treeSet.add(new SimplePerson("Test"));
    treeSet.add(new SimplePerson("Test"));
    System.out.println("TreeSet size = " + treeSet.size()); // Size = 1
}
```

Listing 5.12 *Ausführbar als* **'LAWOFBIG3EXAMPLE'**

[11] Ausnahmen davon sind entsprechend zu dokumentieren.

Zunächst ist überraschend, dass im `HashSet<SimplePerson>` zwei Elemente vorhanden sind und nicht wie im `TreeSet<SimplePerson>` nur eins. Wie ist das zu erklären? Das liegt daran, dass die Methode `equals(Object)`, die zur Bestimmung der Gleichheit von Einträgen innerhalb von Buckets verwendet wird, in der Klasse `SimplePerson` nicht implementiert ist. Somit findet ein Referenzvergleich statt, wenn zwei `SimplePerson`-Objekte verglichen werden. Für die Klasse `TreeSet<SimplePerson>` wird beim Hinzufügen zum Ausschluss doppelter Einträge und damit zum Erhalt der Integrität der Menge die Methode `compareTo(SimplePerson)` anstelle von `equals(Object)` verwendet. In diesem Beispiel besteht demnach das Problem, dass die Methode `compareTo(SimplePerson)`[12] nicht mit `equals(Object)` kompatibel ist, wie dies in Abschnitt 5.1.8 gefordert wurde. Es fehlt eine entsprechende Implementierung von `equals(Object)` in der Klasse `SimplePerson`:

```
@Override
public boolean equals(final Object other)
{
    if (other == null)                          // Null-Akzeptanz
        return false;
    if (this == other)                          // Reflexivität
        return true;
    if (this.getClass() != other.getClass())    // Typgleichheit
        return false;

    final SimplePerson otherPerson = (SimplePerson) other;
    return compareTo(otherPerson) == 0;         // Vergleich mit compareTo()
}
```

Listing 5.13 *Ausführbar als* **'LAWOFBIG3EXAMPLE2'**

Ein erneuter Testlauf liefert immer noch zwei Elemente im `HashSet<SimplePerson>` und eins im `TreeSet<SimplePerson>`. Wie kann das sein, nachdem wir auch die `equals(Object)`-Methode korrigiert haben? Überlegen wir kurz.

Die Erklärung ist einfach: Zwar werden nun zwei `SimplePerson`-Objekte als gleich angesehen, wenn sie denselben Inhalt besitzen, aber zu diesem Vergleich kommt es erst gar nicht. Wie bereits in Abschnitt 5.1.7 angedeutet, berechnet die Methode `hashCode()` zunächst das Bucket zur Speicherung der Objekte. Da keine eigene Implementierung der Methode `hashCode()` existiert, werden die von `equals(Object)` als gleich angesehenen `SimplePerson`-Objekte in unterschiedlichen Buckets gespeichert. Dies widerspricht dem `hashCode()`-Kontrakt und führt dazu, dass das gleiche `SimplePerson`-Objekt zweimal in das `HashSet<SimplePerson>` eingefügt wird. Als Korrektur realisieren wir die `hashCode()`-Methode wie folgt:

```
@Override
public int hashCode()
{
    return name.hashCode();
}
```

Listing 5.14 *Ausführbar als* **'LAWOFBIG3EXAMPLE3'**

[12] Das gilt ebenso, wenn ein `Comparator<SimplePerson>` genutzt wird.

Ein erneuter Testlauf bestätigt, dass als Folge dieser Korrektur nun sowohl das `HashSet<SimplePerson>` als auch das `TreeSet<SimplePerson>` nur noch einen Eintrag enthalten.

Fazit

Dieses einfache Beispiel verdeutlicht das Zusammenspiel der drei Methoden und die Notwendigkeit, die Forderungen der jeweiligen Methodenkontrakte einzuhalten, um Überraschungen oder Merkwürdigkeiten zu vermeiden.

5.1.10 Schlüssel-Wert-Abbildungen und das Interface `Map`

Nachdem wir bisher die konkreten Realisierungen des Interface `Collection<E>` besprochen haben, wenden wir uns nun den Implementierungen des Interface `Map<K,V>` zu. Sie realisieren, wie bereits erwähnt, Abbildungen von Schlüsseln auf Werte. Häufig werden Maps deshalb auch als ***Dictionary*** oder ***Look-up-Tabelle*** bezeichnet.

Die zugrunde liegende Idee ist, jedem gespeicherten Wert einen eindeutigen Schlüssel zuzuordnen. Ein intuitiv verständliches Beispiel sind Telefonbücher: Hier werden Namen auf Telefonnummern abgebildet. Eine Suche über einen Namen (Schlüssel) liefert meistens recht schnell eine Telefonnummer (Wert). Da jedoch keine Rückabbildung von Telefonnummer auf Name existiert, wird das Ermitteln eines Namens zu einer Telefonnummer recht aufwendig.

Das Interface `Map`

Maps speichern Schlüssel-Wert-Paare. Jeder Eintrag in einer Map wird durch das innere Interface `Map.Entry<K,V>` repräsentiert, das die Abbildung zwischen Schlüsseln (Typparameter `K`) und Werten (Typparameter `V`) realisiert. Die Methoden im Interface `Map<K,V>` sind daher auf diese spezielle Form der Speicherung von Schlüssel-Wert-Abbildungen ausgelegt, ähneln aber denen des Interface `Collection<E>`. Das Interface `Map<K,V>` bietet unter anderem folgende Methoden:

- `V put(K key, V value)` – Fügt dieser Map eine Abbildung (Schlüssel auf Wert) als Eintrag hinzu. Falls zu dem übergebenen Schlüssel bereits ein Wert gespeichert ist, so wird dieser mit dem neuen Wert überschrieben. Die Methode gibt den zuvor mit diesem Schlüssel verbundenen Wert zurück, sofern es einen derartigen Eintrag gab, ansonsten wird `null` zurückgegeben.
- `void putAll(Map<? extends K, ? extends V> map)` – Fügt alle Einträge aus der übergebenen Map in diese Map ein. Werte bereits existierender Einträge werden, analog zur Arbeitsweise der Methode `put(K,V)`, überschrieben.
- `V remove(Object key)` – Löscht einen Eintrag (Schlüssel und zugehörigen Wert) aus der Map. Als Rückgabe erhält man den zum Schlüssel `key` gehörenden Wert oder `null`, wenn zu diesem Schlüssel kein Eintrag gespeichert war.

- `V get(Object key)` – Ermittelt zu einem Schlüssel `key` den assoziierten Wert. Existiert kein Eintrag zu dem Schlüssel, so wird `null` zurückgegeben.
- `boolean containsKey(Object key)` – Prüft, ob der Schlüssel `key` in der Map gespeichert ist und liefert genau dann `true`, wenn dies der Fall ist.
- `boolean containsValue(Object value)` – Prüft, ob der Wert `value` in der Map gespeichert ist und liefert genau dann `true`, wenn dies der Fall ist.
- `void clear()` – Löscht alle Einträge der Map.
- `int size()` – Ermittelt die Anzahl der in der Map gespeicherten Einträge.
- `boolean isEmpty()` – Prüft, ob die Map leer ist.

Folgende Methoden bieten Zugriff auf gespeicherte Schlüssel, Werte und Einträge:

- `Set<K> keySet()` – Liefert eine Menge mit allen Schlüsseln.
- `Collection<V> values()` – Liefert die Werte in Form einer Collection.
- `Set<Map.Entry<K,V» entrySet()` – Liefert die Menge aller Einträge. Dadurch hat man sowohl Zugriff auf die Schlüssel als auch auf die Werte.

Diese drei Methoden liefern jeweils Sichten auf die Daten. Erfolgen Veränderungen in der zugrunde liegenden Map, so werden diese in den Sichten widergespiegelt. ***Beachten Sie bitte, dass Änderungen in der Sicht ebenfalls in die Map übertragen werden***. Ähnliches haben wir für Listen und Sets kennengelernt.

Tipp: Der Wert `null` als Schlüssel und als Wert

Liefert die Methode `get(Object)` den Wert `null`, so wird dies vielfach als Nichtvorhandensein eines Eintrags in der Map gedeutet. Diese Schlussfolgerung ist allerdings nicht immer korrekt: In einigen Realisierungen des Interface `Map<K,V>` ist `null` als Wert und sogar als Schlüssel erlaubt. Für `null`-Werte kann man dadurch die Fälle »kein Wert« und »Speicherung des Werts `null`« anhand der Rückgabe von `get()` nicht voneinander unterscheiden. Für diesen Zweck gibt es die Methode `containsKey(Object)`.

Beispiel: Maps im Einsatz

Bevor wir uns die konkreten Realisierungen des Interface `Map<K,V>` anschauen, wollen wir durch ein kleines Beispiel ein wenig vertrauter mit Maps werden. Wir bauen eine Art Telefonbuch nach bzw. realisieren eine Abbildung von `String` auf `Integer`:

```
public static void main(final String[] args)
{
   final Map<String, Integer> nameToNumber = new TreeMap<>();
   nameToNumber.put("Micha", 4711);
   nameToNumber.put("Tim", 0714);
   nameToNumber.put("Jens", 1234);
   nameToNumber.put("Tim", 1508);    // Zweites put() für "Tim"
   nameToNumber.put("Ralph", 2208);
```

```
    // Verschiedene Aktionen ausführen
    System.out.println(nameToNumber);
    System.out.println(nameToNumber.containsKey("Tim")); // Prüfe Schlüssel
    System.out.println(nameToNumber.get("Jens"));        // Zugriff per Schlüssel
    System.out.println(nameToNumber.size());             // Anzahl der Einträge
    System.out.println(nameToNumber.keySet());           // Alle Schlüssel
    System.out.println(nameToNumber.values());           // Alle Werte
}
```

Listing 5.15 *Ausführbar als* **'FIRSTMAPEXAMPLE'**

Starten wir das Programm FIRSTMAPEXAMPLE, so kommt es zu folgenden Ausgaben, die uns schon ein paar Dinge über Maps verraten, nämlich etwa, dass Werte überschrieben werden, wenn mehrmals Daten zum gleichen Schlüssel eingefügt werden:

```
{Jens=1234, Micha=4711, Ralph=2208, Tim=1508}
true
1234
4
[Jens, Micha, Ralph, Tim]
[1234, 4711, 2208, 1508]
```

Die Klasse `HashMap`

Die Klasse `HashMap<K,V>` ist eine Realisierung der abstrakten Klasse `AbstractMap<K,V>`, die das Interface `Map<K,V>` implementiert. Die Datenhaltung geschieht in einer Hashtabelle und ermöglicht dadurch eine effiziente Ausführung gebräuchlicher Operationen wie `get(Object)`, `put(K,V)`, `containsKey(Object)` und `size()`. Die Reihenfolge der Elemente bei einer Iteration wirkt zufällig. Tatsächlich wird sie durch den jeweiligen Hashwert sowie die Verteilung auf die Buckets bestimmt, wie dies bereits für das `HashSet<E>` besprochen wurde (vgl. Abschnitt 5.1.7).

Beispiel Zur Demonstration der Klasse `HashMap<K,V>` wollen wir einen in der Praxis häufig anzutreffenden Anwendungsfall betrachten, bei dem eine Menge von Eingabewerten auf eine Menge von Ausgabewerten abgebildet werden soll. Dazu sieht man häufig `if`- oder `switch`-Anweisungen wie die folgende:

```
private static Color mapToColor(final String colorName)
{
    switch (colorName)
    {
    case "BLACK":
        return Color.BLACK;
    case "RED":
        return Color.RED;
    case "GREEN":
        return Color.GREEN;
    // ... viele mehr ...

    default:
        throw new IllegalArgumentException("No color for: '" + colorName + "'");
    }
}
```

Sind nur ein paar wenige Fälle abzudecken, kann diese Realisierung durchaus akzeptabel sein, je mehr Fälle jedoch aufeinander abgebildet werden sollen, desto umfangreicher und schwieriger wartbar werden solche Konstrukte. Als Abhilfe kann man sich eine Abbildungstabelle in Form einer `HashMap<K,V>` definieren und die Abbildung wird durch einen Zugriff mit dem entsprechenden Schlüssel realisiert:

```
private static final Map<String, Color> nameToColor = new HashMap<>();
static
{
    initMapping(nameToColor);
}

public static void main(final String[] args)
{
    System.out.println(mapToColor("RED"));     // java.awt.Color[r=255,g=0,b=0]
    System.out.println(mapToColor("GREEN"));   // java.awt.Color[r=0,g=255,b=0]
    System.out.println(mapToColor("UNKNOWN")); // => Exception
}

private static Color mapToColor(final String colorName)
{
    if (nameToColor.containsKey(colorName))
    {
        return nameToColor.get(colorName);
    }
    throw new IllegalArgumentException("No color for: '" + colorName + "'");
}

private static void initMapping(final Map<String, Color> nameToColor)
{
    nameToColor.put("BLACK", Color.BLACK);
    nameToColor.put("RED", Color.RED);
    nameToColor.put("GREEN", Color.GREEN);
    // ... viele mehr ...
}
```

Listing 5.16 *Ausführbar als* **'HASHMAPEXAMPLE'**

Diese Art der Realisierung hält die Funktionalität der Abbildung in der Applikation selbst kurz, einfach und übersichtlich. Hier im Beispiel wird aus Gründen der Einfachheit eine statische Definition und Initialisierung genutzt. Sofern benötigt kann die Initialisierung auch ausgelagert werden und mithilfe einer externen Datenquelle, etwa einer Datei, erfolgen. Dadurch erzielt man eine größere Flexibilität.

Die Klasse `LinkedHashMap`

Die Klasse `LinkedHashMap<K,V>` bietet die Funktionalität einer `HashMap<K,V>` und erweitert diese um die Möglichkeit, Elemente in einer definierten Reihenfolge (wahlweise Einfüge- bzw. Zugriffsreihenfolge) zu speichern und abrufen zu können.

Zum einen kann dies nützlich sein, wenn man eine feste Reihenfolge bei der Iteration benötigt – für `HashMap<K,V>` ist die Ausgabe recht willkürlich. Zum anderen und für die Praxis relevanter ist es, dass man mithilfe der Klasse `LinkedHashMap<K,V>` auf einfache Weise Zwischenspeicher, auch ***Cache*** genannt, realisieren kann. Diese sind immer dann nützlich, wenn man beispielsweise wiederholt auf ähnliche Daten aus dem

Dateisystem oder einer Datenbank zugreift. Diese Zugriffe sind teuer, d. h., sie sind aufwendig und führen durch Latenzzeiten auch zu Verzögerungen in der Abarbeitung des Programms. Als Optimierung kann man Caches für die relevantesten Daten im Speicher halten, um auf diese direkt zugreifen zu können. Häufig sind das die zuletzt zugegriffenen Daten.

Im Folgenden betrachten wir zunächst die Realisierung einer Größenbeschränkung, wobei hier das älteste Element anhand der Reihenfolge des Einfügens bestimmt wird. Neuere Daten verdrängen so früher eingefügte.

Steuerung durch Callback-Methode Die Klasse `LinkedHashMap<K,V>` bietet die folgende Callback-Methode, die beim Einfügen von Elementen aufgerufen wird:

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest)
```

Der Rückgabewert bestimmt, ob das jeweils älteste Element aus der Map entfernt werden soll. Die Defaultimplementierung dieser Methode liefert den Wert `false` und sorgt damit dafür, dass beim Hinzufügen von Elementen kein Element gelöscht wird. Soll dieses Verhalten geändert werden, so muss die Methode überschrieben und für zu löschende Elemente den Wert `true` zurückgegeben werden. Das Löschen geschieht dann automatisch durch die Implementierung der Map selbst.

Hinweis: Aussagekräftige Methodennamen im API

Da die Methode `removeEldestEntry(Map.Entry<K,V> eldest)` kein Element löscht, sondern lediglich bestimmt, ob dies geschehen soll, hätte man sie besser `shouldRemoveEldestEntry(Map.Entry<K,V> eldest)` genannt.

Beispiel: Realisierung einer Größenbeschränkung Mithilfe der gerade vorgestellten Callback-Methode kann man leicht eine in ihrer Größe beschränkte Map implementieren, die ältere Elemente entfernt, wenn eine gewisse Größe überschritten ist und dann Elemente eingefügt werden. Die folgende Klasse `FixedSizeLinkedHashMap<K,V>` zeigt, wie einfach eine derartige Größenbeschränkung zu realisieren ist:

```
public final class FixedSizeLinkedHashMap<K, V> extends LinkedHashMap<K, V>
{
    private final int maxEntryCount;

    public FixedSizeLinkedHashMap(final int maxEntryCount)
    {
        this.maxEntryCount = maxEntryCount;
    }

    @Override
    protected boolean removeEldestEntry(final Map.Entry<K, V> customer)
    {
        return size() > maxEntryCount;
    }
}
```

Für die Größenbeschränkung überschreibt man lediglich die Methode `removeEldest-Entry(Map.Entry<K,V>)` und prüft in der Implementierung, ob die Anzahl der gespeicherten Elemente eine zuvor festgelegte Größe übersteigt.

Da hier keine anderweitige Parametrisierung der zugrunde liegenden `Linked-HashMap<K,V>` erfolgt, wird das älteste Element anhand der Reihenfolge des Einfügens bestimmt. Bei Überschreiten der angegebenen Größe wird das laut Einfügereihenfolge älteste, d. h. das zuerst eingefügte Element gelöscht und damit die Größenbeschränkung erhalten.

Im nachfolgenden Beispiel wird die Größe des realisierten Containers zu Demonstrationszwecken auf den Wert 3 festgelegt. Anschließend werden fünf Abbildungen von Namen auf `Customer`-Objekte in der Map abgelegt.

```
public static void main(final String[] args)
{
    // Größenbeschränkung auf drei Elemente
    final int MAX_ELEMENT_COUNT = 3;
    final FixedSizeLinkedHashMap<String, Customer> fixedSizeMap =
            new FixedSizeLinkedHashMap<String, Customer>(MAX_ELEMENT_COUNT);

    // Initial befüllen
    fixedSizeMap.put("Erster", new Customer("Erster", "Stuhr", 11));
    fixedSizeMap.put("Zweiter", new Customer("Zweiter", "Hamburg", 22));
    fixedSizeMap.put("M. Inden", new Customer("Inden", "Aachen", 39));
    printCustomerList("Initial", fixedSizeMap.values());

    // Änderungen durchführen und ausgeben
    fixedSizeMap.put("New1", new Customer("New_1", "London", 44));
    printCustomerList("After insertion of 'New_1'", fixedSizeMap.values());

    fixedSizeMap.put("New2", new Customer("New_2", "San Francisco", 55));
    printCustomerList("After insertion of 'New_2'", fixedSizeMap.values());
}

private static void printCustomerList(final String title,
                                      final Collection<Customer> customers)
{
    System.out.println(title);
    for (final Customer customer : customers)
        System.out.println(customer);
}
```

Listing 5.17 *Ausführbar als* **'FIXEDSIZELINKEDHASHMAPEXAMPLE'**

Führt man das Programm FIXEDSIZELINKEDHASHMAPEXAMPLE aus, wird die Ersetzungsstrategie deutlich: Die beiden zuerst eingefügten Elemente "Erster" und "Zweiter" werden durch die neu hinzugefügten Elemente "New1" und "New2" verdrängt:

```
[...]
After insertion of 'New_1'
Customer [name=Zweiter, city=Hamburg, age=22]
Customer [name=Inden, city=Aachen, age=39]
Customer [name=New_1, city=London, age=44]
After insertion of 'New_2'
Customer [name=Inden, city=Aachen, age=39]
Customer [name=New_1, city=London, age=44]
Customer [name=New_2, city=San Francisco, age=55]
```

Beispiel: Realisierung eines LRU-Caches Statt der Reihenfolge des Einfügens als Verbleibkriterium zu nutzen, ist es oftmals sinnvoller, zu betrachten, welche Elemente zuletzt verwendet wurden.[13] Man realisiert dazu einen sogenannten ***LRU-Cache*** (Least-Recently-Used), der die zuletzt benutzten Objekte zwischenspeichert, indem er die am längsten nicht mehr zugegriffenen Elemente im Cache austauscht:

```
public final class LruLinkedHashMap<K, V>  extends LinkedHashMap<K, V>
{
    // Kopie der Package-privaten Definitionen aus der Klasse HashMap
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    private static final float DEFAULT_LOAD_FACTOR = 0.75f;
    private static final boolean USE_ACCESS_ORDER = true;

    private final int maxEntryCount;

    public LruLinkedHashMap(final int maxEntryCount)
    {
        // Unschön: Um die Eigenschaft accessOrder anzugeben, müssen wir Werte
        // an den Konstruktor übergeben, die wir nicht spezifizieren wollen
        super(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR, USE_ACCESS_ORDER);
        this.maxEntryCount = maxEntryCount;
    }

    @Override
    protected boolean removeEldestEntry(final Map.Entry<K, V> customer)
    {
        return size() > maxEntryCount;
    }
}
```

Zur Verdeutlichung der Arbeitsweise der Klasse `LruLinkedHashMap` speichern wir dort wieder einige `Customer`-Objekte. Danach wird dann nur auf drei der vier zuvor gespeicherten Einträge zugegriffen. Durch das Einfügen eines weiteren Eintrags wird der am längsten nicht verwendete Eintrag ersetzt. Im folgenden Beispiel wird der Eintrag »`M. Inden`« durch den Eintrag »`D. Dummy`« ersetzt.

```
public static void main(final String[] args)
{
   // Größenbeschränkung auf vier Elemente
   final int MAX_ELEMENT_COUNT = 4;
   final LruLinkedHashMap<String, Customer> lruMap =
                                    new LruLinkedHashMap<>(MAX_ELEMENT_COUNT);

   lruMap.put("A. Mustermann", new Customer("A. Mustermann", "Stuhr", 16));
   lruMap.put("B. Mustermann", new Customer("B. Mustermann", "Hamburg", 32));
   lruMap.put("C. Mustermann", new Customer("C. Mustermann", "Zürich", 64));
   lruMap.put("M. Inden", new Customer("M. Inden", "Kiel", 32));

   printCustomerList("Initial", lruMap.values());

   // Zugriff auf alle bis auf M. Inden
   lruMap.get("A. Mustermann");
   lruMap.get("B. Mustermann");
   lruMap.get("C. Mustermann");
```

[13] Um die Eigenschaft der Zugriffsreihenfolge überhaupt setzen zu können, ist man durch die Konstruktoren der `LinkedHashMap<K,V>` dazu gezwungen, die Werte für Initialkapazität und Füllgrad (Load Factor) anzugeben.

```
    // Neuer Eintrag sollte M. Inden ersetzen
    lruMap.put("Dummy", new Customer("D. Dummy", "Oldenburg", 128));

    printCustomerList("Nach Zugriffen", lruMap.values());
}
```

Listing 5.18 *Ausführbar als* **'LRULINKEDHASHMAPEXAMPLE'**

Ein Start des Programms LRULINKEDHASHMAPEXAMPLE zeigt dies und produziert die hier gekürzte Ausgabe:

```
[...]
Nach Zugriffen
Customer [name=A. Mustermann, city=Stuhr, age=16]
Customer [name=B. Mustermann, city=Hamburg, age=32]
Customer [name=C. Mustermann, city=Zürich, age=64]
Customer [name=D. Dummy, city=Oldenburg, age=128]
```

Die Klasse `TreeMap`

Die Klasse `TreeMap<K,V>` ist eine Erweiterung der abstrakten Klasse `AbstractMap<K,V>` und implementiert das Interface `SortedMap<K,V>`. Eine `TreeMap<K,V>` stellt automatisch die Ordnung der gespeicherten Schlüssel her, und nutzt dazu entweder das Interface `Comparable<T>` oder einem im Konstruktor übergebenen `Comparator<T>`. Außerdem implementiert die Klasse `TreeMap<K,V>` das Interface `NavigableMap<K,V>`, das einige nützliche Methoden definiert: Durch Aufruf der Methode `ceilingKey(K)` erhält man einen passenden Schlüssel, der größer oder gleich dem übergebenen Schlüssel ist. Korrespondierende Methoden `floorKey(K)`, `lowerKey(K)` und `higherKey(K)` liefern Schlüssel, die kleiner oder gleich, kleiner und größer als der angegebene Schlüssel sind. Weiterhin kann man dazugehörende Einträge der Map über korrespondierende `xyzEntry(K)`-Methoden ermitteln, wobei `xyz` für `lower`, `higher` usw. steht.

Beispiel In folgendem Beispiel nutzen wir die genannten Methoden, um eine Abbildung von Namen auf das Alter zu erreichen und passende Schlüssel bzw. Einträge zu einem übergebenen Namenskürzel zu ermitteln:

```
public static void main(final String[] args)
{
    final NavigableMap<String, Integer> nameToAgeMap = new TreeMap<>();
    nameToAgeMap.put("Max", 47);
    nameToAgeMap.put("Moritz", 39);
    nameToAgeMap.put("Micha", 43);

    System.out.println("floor   Ma: " + nameToAgeMap.floorKey("Ma"));
    System.out.println("higher  Ma: " + nameToAgeMap.higherKey("Ma"));
    System.out.println("lower   Mz: " + nameToAgeMap.lowerKey("Mz"));
    System.out.println("ceiling Mc: " + nameToAgeMap.ceilingEntry("Mc"));
}
```

Listing 5.19 *Ausführbar als* **'TREEMAPEXAMPLE'**

Führt man das Programm TREEMAPEXAMPLE aus, so erhält man diese Ausgabe:

```
floor   Ma: null
higher  Ma: Max
lower   Mz: Moritz
ceiling Mc: Micha=43
```

Die Werte verdeutlichen die vorangegangenen Beschreibungen: Der Aufruf von `floorKey("Ma")` liefert den Vorgängereintrag von "Ma". Weil dieser nicht existiert, wird `null` zurückgeliefert. Ein Aufruf von `higherKey("Ma")` liefert den Nachfolgereintrag von "Ma" und dies ist der Schlüssel "Max". Analog arbeiten die anderen Zugriffe.

5.1.11 Erweiterungen am Beispiel der Klasse `HashMap`

Manchmal möchte man die bestehenden Containerklassen um etwas Funktionalität erweitern. Nachfolgend wollen wir das exemplarisch für die Klasse `HashMap<K,V>` tun. Dort soll die Normierung von Schlüsseln gezeigt werden. Das dient z. B. dazu, Benutzereingaben dezent zu korrigieren, etwa führende oder abschließende Leerzeichen zu entfernen, damit nach einheitlichen Schlüsseln gesucht werden kann.

Anwendungskontext

Verwenden wir die Klasse `HashMap<K,V>`, um Bilder von Typ `java.awt.Image` über einen symbolischen Namen statt über ihren Dateipfad zu referenzieren. Für eine erste, typsichere Umsetzung kann man im einfachsten Fall direkt eine `HashMap<String, Image>` nutzen. Ein Textfeld erlaubt die Eingabe von beliebigen Namen mit dem Ziel, ein Bild darzustellen, wenn zu dem eingegebenen Namen ein entsprechender Eintrag in der Map vorhanden ist.

Das folgende Listing zeigt das initiale Befüllen der Abbildungstabelle und einige Zugriffe darauf, insbesondere den Umgang mit `null` als Schlüssel und als Wert sowie den Spezialfall eines Schlüssels mit Leerzeichen. Die Kommentare im Listing sowie die Ausgaben auf der Konsole helfen, ein erstes Verständnis aufzubauen:

```
public static void main(final String[] args) throws IOException
{
    // Typsichere Definition
    final Map<String, Image> nameToImageMap = new HashMap<>();

    // Speicherung einiger Mappings Name -> Bild
    nameToImageMap.put("Fußball", readImageFile("tile_gras_1.jpg"));
    nameToImageMap.put("Wasserball", readImageFile("tile_water.jpg"));
    nameToImageMap.put("Klettern", readImageFile("tile_rock_2.jpg"));
    // Zugriff liefert BufferedImage
    System.out.println("'Fußball' "  + nameToImageMap.get("Fußball"));

    // Spezialfall: Schlüssel mit Leerzeichen und null als Wert
    nameToImageMap.put("Skaten ", null);
    // Zugriff liefert null, obwohl Schlüssel vorhanden
    System.out.println("'Skaten ' " + nameToImageMap.get("Skaten "));
    // containsKey() wertet dies korrekt mit true aus
    System.out.println("'Skaten ' " + nameToImageMap.containsKey("Skaten "));
```

```
    // Füge eine Abbildung von null auf Bild hinzu
    nameToImageMap.put(null, readImageFile("tile_gras_2.jpg"));

    // Ausgabe aller Schlüssel und Werte
    System.out.println("Keys = " + nameToImageMap.keySet());
    System.out.println("Values = " + nameToImageMap.values());

    final JFrame frame = new AppFrame("NameToImageMap-Demo", nameToImageMap);
    frame.setVisible(true);
}
```

Listing 5.20 *Ausführbar als* **'NAMETOIMAGEMAPEXAMPLE'**

Das Programm NAMETOIMAGEMAPEXAMPLE produziert folgende Ausgaben:

```
'Fußball' BufferedImage@4fca772d: type = 5 ColorModel: #pixelBits = 24...
'Skaten ' null
'Skaten ' true
Keys = [Fußball, null, Wasserball, Skaten , Klettern]
Values = [BufferedImage@4fca772d: type = 5 ColorModel: #pixelBits = 24...
```

Darüber hinaus wird durch das Programm ein einfaches GUI bereitgestellt. Dort stehen in einer Combobox alle vorhandenen Schlüssel-Wert-Abbildungen zur Auswahl. Das zeigt Abbildung 5-8.

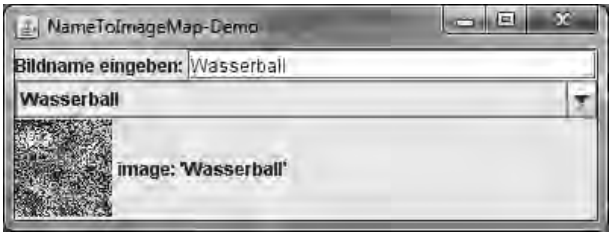


Abbildung 5-8 *Beispielapplikation* NAMETOIMAGEMAPEXAMPLE

Hinweis: Benutzereingaben als Schlüssel

Beim Experimentieren mit eigenen Eingaben wird ein mögliches Problem bei der Verarbeitung von Benutzereingaben sichtbar: Werden diese ungeprüft, unbearbeitet und ohne Korrekturen direkt zur Abfrage als Schlüssel einer Map verwendet, so ist es nicht möglich, gespeicherte Werte zuverlässig wiederzufinden. Fehler treten z. B. dann auf, wenn man ein Wort oder einen Buchstaben kleinschreibt oder die Eingabe versehentlich ein führendes oder nachfolgendes Leerzeichen enthält. Sollen textuelle Werte als Referenz auf Schlüssel einer Map dienen, ist es daher wichtig, eine konsistente Umwandlung oder Normalisierung (z. B. Abschneiden von Leerzeichen) der eingegebenen Texte in eine festgelegte Darstellungsform (beispielsweise eine Darstellung komplett in Groß- oder Kleinbuchstaben) zu definieren. Im Folgenden gehe ich auf derartige Erweiterungen ein.

Lösungsvarianten

Weil die Schlüssel aus Benutzereingaben stammen können, besteht die Gefahr von Inkonsistenzen. Daher soll eine konsistente Normalisierung von Schlüsseln in eine festgelegte Darstellungsform erfolgen. Weiterhin ist es wünschenswert, dies in der zu erstellenden Containerklasse einmal zentral zu realisieren. Zudem soll die Namensabbildung für nutzende Applikationen unsichtbar und ohne Aufwand einsetzbar sein. Um die gewünschten Erweiterungen umzusetzen, existieren folgende zwei Alternativen:

1. ***Aggregation*** einer Containerklasse und ***Delegation*** an deren Methoden
2. ***Ableitung*** von einer Containerklasse und ***Überschreiben*** von Methoden

Aggregation und Delegation Verwendet man Delegation, so muss die benötigte Funktionalität über Methodenaufrufe an die aggregierte Containerklasse selbst programmiert werden. Eine Realisierung könnte wie folgt aussehen:

```
public final class NameToImageMapUsingDelegation
{
    private final Map<String, Image> nameToImage = new HashMap<>();

    public void put(final String name, final Image image)
    {
        final String key = name != null ? name.toUpperCase().trim() : null;
        nameToImage.put(key, image);
    }

    public Image get(final String name)
    {
        final String key = name != null ? name.toUpperCase().trim() : null;
        return nameToImage.get(key);
    }

    public void clear()
    {
        nameToImage.clear();
    }
}
```

Diese Art der Realisierung besitzt folgende Auswirkungen:

- Es wird häufig – wie im Beispiel auch – zunächst nur diejenige Funktionalität bereitgestellt, die man initial benötigt. Ist später mehr Containerfunktionalität erforderlich, so muss diese passend realisiert werden. Im Speziellen kann man aber auch bewusst gewisse Methoden in der Schnittstelle *nicht* anbieten.[14]
- Ein derart realisierte Klasse erschwert die Handhabung, weil sie nicht gut mit dem Collections-Framework harmoniert: Da weder ein Basisinterface erfüllt noch eine Basisklasse erweitert wird, etwa `Map<K,V>`, lässt sich die von der Utility-Klasse `Collections` angebotene Funktionalität nicht oder nur eingeschränkt nutzen.

[14] Bei einer Realisierung als Subklasse kann man dies nur durch Überschreiben und Auslösen von z. B. einer `UnsupportedOperationException` in der Implementierung ausdrücken.

- Ein weiteres Problem ist, dass Methodensignaturen umdefiniert werden können. Dies führt aber zu weiteren Inkompatibilitäten mit dem Collections-Framework.

Ableitung und Überschreiben Seit JDK 5 kann man typsichere Containerklassen auf elegantere Art und Weise als bisher gezeigt realisieren. Man nutzt dazu eine Kombination von Ableitung und Einsatz von Generics. Mithilfe kovarianter Rückgabewerte werden sogar typsichere Rückgabewerte möglich (vgl. Abschnitt 3.6.2). Außerdem bleibt man durch diese Art der Realisierung kompatibel zu allen Funktionalitäten, die durch die Utility-Klasse `Collections` zur Verfügung gestellt werden.

Folgendes Beispiel zeigt den ersten Versuch, die geforderte Funktionalität auf Basis einer typsicheren `HashMap<String, Image>` umzusetzen:

```
// ACHTUNG: Fehlerhafter erster Versuch!
public final class NameToImageMap extends HashMap<String, Image>
{
    @Override
    public Image put(final String name, final Image image)
    {
        return super.put(name.toUpperCase().trim(), image);
    }

    // @Override => nicht möglich da Signatur get(Object)
    public Image get(final String name)
    {
        return super.get(name.toUpperCase().trim());
    }
    // ...
}
```

Auf den ersten Blick ist kein Fehler zu erkennen. Tatsächlich enthält diese Art der Umsetzung jedoch folgende Probleme:

1. Die Realisierung unterstützt keine `null`-Werte als Schlüssel, sondern führt stattdessen sogar zu einer `NullPointerException`. ***Damit verstößt diese Umsetzung gegen die Methodenkontrakte und verhält sich nicht korrekt wie eine Subklasse***. Das Problem ist relativ einfach dadurch zu lösen, dass man eine Hilfsmethode `normalizeKey(String)` wie folgt implementiert:

   ```
   private String normalizeKey(final String key)
   {
       if (key == null)
           return null;

       return key.toUpperCase().trim();
   }
   ```

2. Die oben im Listing gezeigte Methode `put(String, Image)` ist korrekt überschrieben, die Methode `get(Object)` jedoch nicht! Hier findet vielmehr ein versehentliches Überladen von `get(Object)` statt. Ohne Nutzung der Annotation `@Override` kann das leicht passieren, da im Collections-Framework leider einige Methoden mit Parametern vom Typ `Object` statt des Typs `K` des Schlüssels

definiert sind. Diese Besonderheit gilt auch für `get()`-Methoden und erfordert Vorsicht, um Typfehler zu vermeiden. Um Fehler beim Überschreiben durch den Compiler aufdecken zu können, bietet es sich an, ***alle Methoden, die man überschreiben möchte, mit der Annotation `@Override` zu kennzeichnen***.

3. Damit sich die Klasse korrekt als Spezialisierung verhält, müssen alle Methoden angepasst werden, die einen Schlüssel als Parameter erwarten. Geschieht dies nicht, kann man ansonsten zwar problemlos Elemente speichern, eine Abfrage über `containsKey(Object)` oder ein Löschen über `remove(Object)` würde jedoch nicht richtig arbeiten. Ohne Anpassungen in einer Überschreibung werden lediglich die Methoden der Oberklasse aufgerufen. ***Es wird zuvor eine Umwandlung der Schlüssel benötigt, um garantiert mit passenden Schlüsseln zu suchen***.

Man kann die Klasse derart verallgemeinern, dass statt des Typs `Image` beliebige Typen gespeichert werden können, indem man eine generische Definition mit dem Typkürzel `V` nutzt. Zudem werden alle Methoden, die auf Schlüssel zugreifen, entsprechend angepasst, wodurch sich die Klasse wie eine Spezialisierung einer `HashMap<K,V>` verhält, die als Besonderheit jedoch die Schlüssel normalisiert. Folgende Klasse `UpperCaseNormalizedHashMap<V>` behebt die angesprochenen Mängel:

```
public final class UpperCaseNormalizedHashMap<V> extends HashMap<String, V>
{
    @Override
    public V put(final String key, final V value)
    {
        return super.put(normalizeKey(key), value);
    }

    @Override
    public V get(final Object key)
    {
        return super.get(normalizeKey((String) key));
    }

    @Override
    public boolean containsKey(final Object key)
    {
        return super.containsKey(normalizeKey((String) key));
    }

    // ...

    private String normalizeKey(final String key)
    {
        if (key == null)
            return null;

        return key.toUpperCase().trim();
    }
}
```

Listing 5.21 *Ausführbar als* **'UPPERCASENORMALIZEDHASHMAP'**

Diese Klasse erfüllt die Anforderungen, passt sich ins Collections-Framework ein und stellt somit eine gelungenere Realisierung als diejenige durch Aggregation dar.

5.1.12 Entscheidungshilfe zur Wahl von Datenstrukturen

Wir haben mittlerweile eine Vielzahl von Containerklassen und dabei auch Details zu deren Arbeitsweise kennengelernt. Nachfolgend möchte ich daraus eine Entscheidungshilfe ableiten, weil die adäquate Wahl der für ein Problem geeigneten Datenstruktur große Auswirkungen sowohl auf die Lesbarkeit und Verständlichkeit als auch auf die Performance haben kann: Nehmen wir an, man würde statt einer Liste ein Array verwenden, obwohl die nutzenden Programmteile Listenfunktionalität benötigen. Diese Funktionalität müsste dann jeweils an der einsetzenden Stelle vom Entwickler selbst programmiert werden. Durch diese für das Problem unpassend gewählte Datenstruktur kommt es zu mehr Sourcecode und mehr Komplexität. Zudem steigt die Wahrscheinlichkeit für Fehler, weil Eigenimplementierungen weniger ausgereift und gut getestet sind als die Containerklassen des JDKs.

Abbildung 5-9 bietet eine Entscheidungshilfe zur Auswahl einer geeigneten Datenstruktur aus dem Collections-Framework. Als Faustregel gilt, dass für Listen die `ArrayList<E>` und für Maps die `HashMap<K,V>` vielfach eine gute Wahl sind, unter anderem auch weil sie in der Regel die beste Performance (vgl. Abschnitt 22.2.1) liefern. Arrays nutzt man z. B. zur Verwaltung primitiver Typen. Insbesondere bei Mehrdimensionalität stellt ein Array-Zugriff vielfach die natürlichste Zugriffsvariante dar.

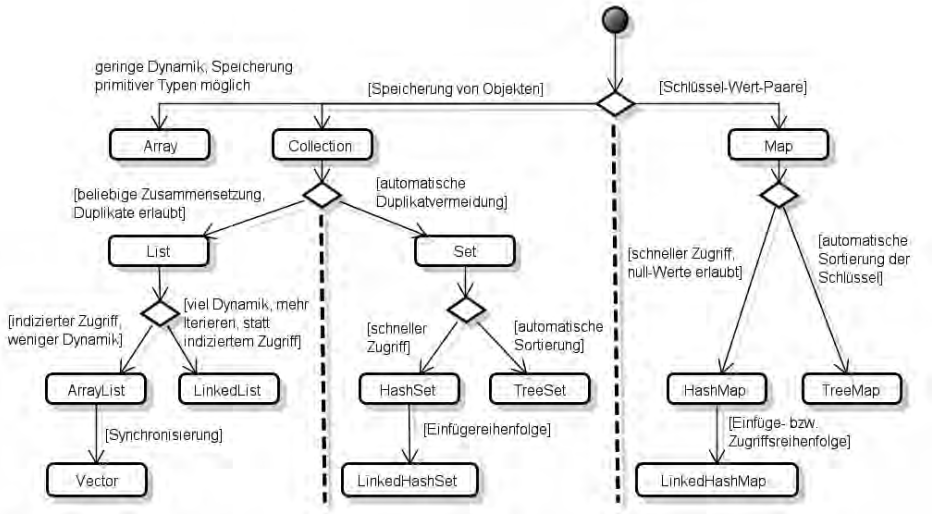


Abbildung 5-9 *Entscheidungshilfe zur Wahl von Datenstrukturen*

Die Grafik lehnt sich an Ideen aus dem Buch »Java 2 – Designmuster und Zertifizierungswissen« [20] von Friedrich Esser an und beschränkt sich aus Gründen der Übersichtlichkeit nur auf die zuvor besprochenen Klassen.

5.2 Suchen, Sortieren und Filtern

Nachdem wir bisher hauptsächlich das Einfügen und Löschen von Daten in Containern betrachtet haben, wollen wir uns im Folgenden mit den Themen Suchen und Sortieren beschäftigen. Das sind zwei elementare Themen der Informatik im Bereich der Algorithmen und Datenstrukturen. Das Collections-Framework setzt beide um und nimmt einem dadurch viel Arbeit ab. Allerdings ist eine wichtige und in der Praxis häufig benötigte Funktionalität nicht enthalten: das Filtern.

Zunächst betrachten wir das Suchen in Abschnitt 5.2.1. Danach behandeln die Abschnitte 5.2.2 sowie 5.2.3 das Sortieren von Arrays und Listen sowie mithilfe von Komparatoren. Zum Schluss geht Abschnitt 5.2.4 auf das Filtern von Collections nach verschiedenen Kriterien ein.[15]

5.2.1 Suchen

Praktischerweise besitzen alle Containerklassen Methoden, mit denen man nach Elementen suchen kann und auch um zu prüfen, ob Elemente enthalten sind.

Suchen mit `contains()`

Wenn Containerklassen über den allgemeinen Typ `Collection<E>` angesprochen werden, so kann mithilfe der Methode `contains(Object)` lediglich ermittelt werden, ob gewünschte Elemente enthalten sind. Darüber hinaus kann mit `containsAll(Collection<?>)` geprüft werden, ob eine Menge von Elementen enthalten ist. Dabei wird über die gespeicherten Elemente iteriert, die mithilfe von `equals(Object)` auf Gleichheit mit dem übergebenen Element bzw. den übergebenen Elementen geprüft werden. Für Maps existieren – wie bereits erwähnt – korrespondierende Methoden `containsKey(Object)` und `containsValue(Object)`.

Suchen mit `indexOf()` und `lastIndexOf()`

Für Listen gibt es ergänzend zu der Prüfung auf Existenz mit `contains(Object)` die Methoden `indexOf(Object)` und `lastIndexOf(Object)`, um die Position eines gesuchten Elements zu ermitteln. Die erste Methode beginnt die Suche am Anfang einer Liste und die zweite beginnt an deren Ende. Auf diese Weise kann, sofern vorhanden, entweder das erste bzw. letzte Vorkommen ermittelt werden. Gleichheit wird wiederum durch `equals(Object)` überprüft.

Suchen mit `binarySearch()`

Neben den gerade genannten Suchmethoden, die iterativ so lange alle Elemente der Datenstruktur betrachten, bis sie fündig geworden sind, wird für die Datenstrukturen

[15] Das findet erst in JDK 8 mit dem Filter-Map-Reduce-Framework und den Stream-Klassen Einzug in Java und wird ausführlicher in Kapitel 12 besprochen.

Array und `List<E>` außerdem eine extrem effiziente Suche, die sogenannte ***Binärsuche***, angeboten. ***Das setzt allerdings zwingend eine sortierte Datenstruktur voraus.*** Den Vorteil der Binärsuche gegenüber einer linearen Suche erkennt man bei größeren Datenvolumina: Die Binärsuche ist extrem performant und benötigt zum Suchen eines Elements eine mit der Anzahl der gespeicherten Elemente logarithmisch wachsende Zeit. Das liegt daran, dass der Algorithmus der Suche die jeweils zu untersuchenden Suchabschnitte halbiert und danach im passenden Teilstück weitersucht. Die beschriebene Binärsuche wird im JDK durch die überladene Methode `binarySearch()` in den Utility-Klassen `Arrays` bzw. `Collections` realisiert. Abbildung 5-10 stellt den prinzipiellen Ablauf dar, wobei aussortierte Teilstücke grau markiert sind.

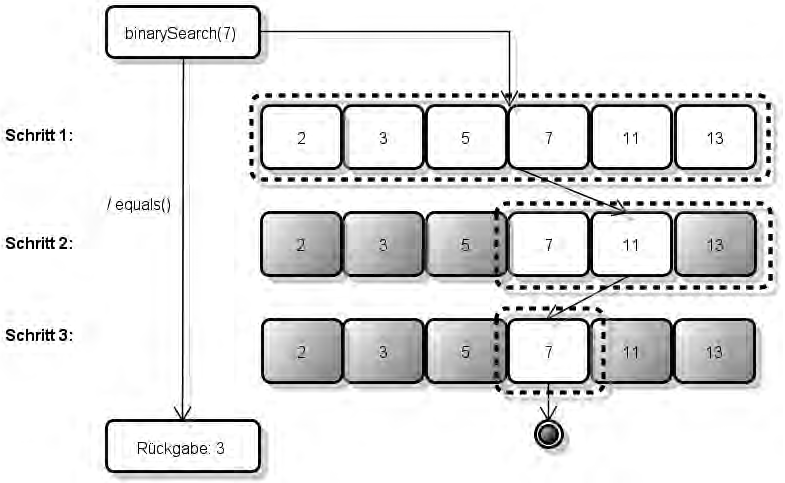


Abbildung 5-10 *Schematischer Ablauf bei der Binärsuche*

Analogie: Binärsuche im realen Leben

Für einige eher abstrakte Algorithmen lassen sich schöne Analogien aus dem realen Leben finden. Dies gilt auch für die Binärsuche. Betrachten wir dazu die Suche nach einer Telefonnummer in einem Telefonbuch, etwa für Aachen. Wollten wir dort die Telefonnummer von Michael Inden finden, so würden wir das Telefonbuch etwa in der Mitte aufschlagen, da wir wissen, dass die Einträge alphabetisch nach Nachnamen geordnet sind und demnach Inden ziemlich mittig zu finden sein sollte – jedenfalls wenn man in etwa eine Gleichverteilung der Nachnamen und deren Anfangsbuchstaben annimmt. Möglicherweise haben wir nicht exakt getroffen und sind bei Max Mustermann gelandet. Dann gehen wir etwas nach vorne und schlagen dort auf. Wir landen etwa bei Nachnamen mit dem Anfangsbuchstaben H. Dann gehen wir wieder ein Stück nach hinten und landen etwa beim Buchstaben J. Das setzen wir fort, bis wir den Autor gefunden haben. Nach diesem Prinzip arbeitet die Binärsuche und ist damit deutlich effizienter, als die Suche beim Buchstaben A zu starten und sich sukzessive bis I vorzuarbeiten.

Binärsuche in Arrays und Listen Betrachten wir als Beispiel eine Suche in einer `List<Integer>`, die einige Primzahlen in aufsteigender Ordnung speichert. Dort suchen wir nach den Zahlen 7 und 14. Im Anschluss daran kehren wir die Reihenfolge der Elemente um und wiederholen die Suche. Das Ganze wird folgendermaßen realisiert:

```
public static void main(final String[] args)
{
    final List<Integer> primes = Arrays.asList(2, 3, 5, 7, 11, 13);

    System.out.println(primes); // [2, 3, 5, 7, 11, 13]

    // Suche nach 7 und 14
    System.out.println(Collections.binarySearch(primes, 7));  // 3
    System.out.println(Collections.binarySearch(primes, 14)); // -7

    // Anderes Sortierkriterium anwenden, um Versagen der Suche zu provozieren
    Collections.sort(primes, Collections.reverseOrder());
    System.out.println(primes); // [13, 11, 7, 5, 3, 2]

    // Suche nach 7 und 14
    System.out.println(Collections.binarySearch(primes, 7));  // 2
    System.out.println(Collections.binarySearch(primes, 14)); // -7
}
```

Listing 5.22 *Ausführbar als* **'BINARYSEARCHEXAMPLE'**

Führt man das Programm BINARYSEARCHEXAMPLE aus, so wird der Wert 7 wie erwartet an Position drei (0-basiert) gefunden. Der gesuchte Wert 14 ist jedoch nicht gespeichert. Die Methode `binarySearch()` liefert als Ergebnis den Wert -7. Sind die Rückgabewerte der Methode `binarySearch()` größer oder gleich 0, so entspricht dies dem 0-basierten Index, an dem der gesuchte Wert gefunden wurde. Ein negativer Wert bedeutet, dass das gesuchte Element nicht in der Datenstruktur gespeichert ist. Über den Rückgabewert wird die Position beschrieben, an der der gesuchte Wert in den Container eingefügt werden kann, ohne die Sortierung zu zerstören. Demnach codiert der Rückgabewert die Position wie folgt: $Position = -Rückgabewert - 1$. Die Zahl 14 könnte demnach an Position 6 eingefügt werden, also hinter der 13.

Bis jetzt scheint alles recht einfach. Um auf Fallstricke hinzuweisen, wird für das Beispiel anschließend eine absteigende Sortierung durch einen Aufruf der statischen Methode `Collections.sort(List<T>, Comparator<? super T>)` hergestellt. Dazu nutzen wird einen Komparator, den man durch Aufruf der statischen Methode `reverseOrder()` erzeugt. Nun wird die Suche auf der absteigend sortierten Liste wiederholt. Der Wert 7 wird dann an Position 2 gefunden, was jedoch ein Zufallstreffer ist, der durch die Teilung der Eingabewerte und die Mittenposition des Werts 7 entsteht.[16] Auch der Rückgabewert -7 für die Suche nach dem Wert 14 verwundert, denn in der umsortierten Liste wäre der Wert -1 die richtige Rückgabe für Position 0 gewesen, an der 14 eingefügt werden müsste (siehe nachfolgenden Kasten »Fallstrick«).

[16] Erweitert man die Eingabemenge der Primzahlen um die Werte 17 und 19, so wird als Folge die Zahl 7 nicht mehr gefunden und die Binärsuche liefert die Rückgabe -1.

Fallstrick: Unterschiedliche Reihenfolgen bei Sortierung und Suche

Wie im Beispiel gesehen, ist es elementar wichtig, dass ***sowohl die zur Sortierung genutzte als auch die bei der Suche verwendete Reihenfolge identisch sind***.[a] Ansonsten ist das Ergebnis einer solchen Suche nicht definiert, liefert jedoch häufig den Wert -1. Da die Binärsuche ohne Angabe eines Sortierkriteriums von einer Sortierung gemäß der natürlichen Ordnung »ausgeht«, kommt es im obigen Beispiel durch die abweichende Sortierung aber auch zu der falschen Angabe von -7 für eine Suche der Zahl 14 in der absteigend sortierten Liste von Primzahlen.

[a] Dies ist eine »beliebte« Fangfrage bei OCPJP/SCJP-Prüfungen.

Binärsuche in Sets und Maps? Intuitiv fragt man sich: Wie sucht man effizient in Sets und Maps? Mit dem bisher erlangten Wissen können wir diese Frage leicht beantworten: Die hashbasierten Container bieten aufgrund der Speicherung in einer Hashtabelle bereits einen extrem performanten Zugriff (sofern die Hashfunktion gut und nicht zu aufwendig zu berechnen ist), sodass eine zusätzliche *eigenständige* Realisierung einer derartigen Suche nicht sinnvoll ist.[17] Das Gleiche gilt für die sortierten Container `TreeSet<E>` und `TreeMap<K,V>`. Wie in Abschnitt 5.1.8 beschrieben, verwenden diese Datenstrukturen einen balancierten Baum, in dem die Suche nach dem gleichen Prinzip wie bei der Binärsuche erfolgt.

5.2.2 Sortieren von Arrays und Listen

Zum Sortieren kann man im Collections-Framework neben verschiedenen Realisierungen der automatisch sortierenden Container `SortedSet<E>` und `SortedMap<K,V>` auch Arrays und Listen durch einen Aufruf der jeweiligen Methode `sort()` aus den Utility-Klassen `Arrays` bzw. `Collections` bei Bedarf sortieren.

Bevor wir im nächsten Abschnitt Komparatoren, die die Reihenfolge der Elemente steuern, genauer betrachten, wollen wir anhand der manuellen Sortierung von Listen den Einsatz von Komparatoren rekapitulieren und danach auch einen Blick auf die Sortierung von Arrays werfen.

Warnhinweis: `null`-Prüfungen in Komparatoren

Um die nachfolgenden Ausführungen zu den Komparatoren nicht mit `null`-Prüfungen unübersichtlich zu machen, werde ich darauf (weitestgehend) verzichten. In eigenen Applikationen ist es jedoch sinnvoll, diese Prüfungen vorzunehmen. Noch besser ist es, die Eingabedaten `null`-frei zu halten, sofern dies möglich ist.

[17] Zudem ist eine Realisierung aufgrund der unsortierten Speicherung in einer Hashtabelle nicht möglich.

Beispiel: Sortieren einer Liste

Zur Sortierung von Listen nutzen wir die Methode `sort()` sowie die Interfaces `Comparable<T>` und `Comparator<T>`, die die Reihenfolgen von Objekten festlegen:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Andy", "Michael", "Tim", "Stefan");

   // Comparable-basierte Sortierung
   Collections.sort(names);
   System.out.println(names);

   final Comparator<String> byLength = new Comparator<String>()
   {
      @Override
      public int compare(final String str1, final String str2)
      {
         return Integer.compare(str1.length(), str2.length());
      }
   };

   // Comparator-bestimmte Sortierung
   Collections.sort(names, byLength);
   System.out.println(names);
}
```

Listing 5.23 *Ausführbar als* **'LISTSORTEXAMPLE'**

Führen wir das Programm LISTSORTEXAMPLE aus, so zeigt die Ausgabe zunächst eine alphabetische Sortierung und dann eine nach der Länge der Strings:

```
[Andy, Michael,  Stefan, Tim]
[Tim, Andy, Stefan, Michael]
```

Vereinfachungen mit JDK 8 JDK 8 bietet verschiedenste Neuerungen, die bei Sortierungen nützlich sind. Das sind unter anderem Lambdas, Methodenreferenzen und Defaultmethoden (vgl. Kapitel 11) – aber auch Streams können gewinnbringend eingesetzt werden. Zudem wurde auch das Interface `Comparator` erweitert, sodass sich nun mit sehr wenig Aufwand eigene Komparatoren erstellen lassen. Details finden Sie in Abschnitt 15.1. Nachfolgend ist dazu nur ein kurzes Beispiel angegeben:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Andy", "Michael", "Tim", "Stefan");

   // Comparable-basierte Sortierung
   names.sort(Comparator.naturalOrder());
   System.out.println(names);

   // Comparator-bestimmte Sortierung mit Lambda
   final Comparator<String> byLength =
         (str1, str2) -> Integer.compare(str1.length(), str2.length());
   names.sort(byLength);
   System.out.println(names);
}
```

Beispiel: Sortieren eines Arrays

Analog zu dem zuvor gezeigten Sortieren von Listen kann man Arrays sortieren. Dafür nutzt man die Methode `Arrays.sort()` – allerdings existiert hier nur eine `Comparable<T>`-basierte Sortierung:

```
public static void main(final String[] args)
{
    final String[] names = { "Andy", "Michael", "Tim", "Stefan" };

    // Comparable-basierte Sortierung
    Arrays.sort(names);
    System.out.println(Arrays.toString(names));
}
```

Vereinfachungen mit JDK 8 Auch für Arrays wurde das Sortieren mit JDK 8 erweitert. Hier wird sogar eine parallele Ausführung der Sortierung möglich. Details finden Sie in Abschnitt 15.3. Nachfolgend ist lediglich die prägnante Schreibweise dargestellt, damit Sie schon einen Eindruck der Vorzüge von Java 8 bekommen:

```
public static void main(final String[] args)
{
    final String[] manyNames = { "Andy", "Michael", "Tim", /* ... */ "Stefan" };

    // Comparable-basierte, parallele Sortierung
    Arrays.parallelSort(manyNames);
    System.out.println(Arrays.toString(manyNames));

    // Comparator-bestimmte, parallele Sortierung mit Lambda
    final Comparator<String> byLength =
          (str1, str2) -> Integer.compare(str1.length(), str2.length());
    Arrays.parallelSort(manyNames, byLength);
    System.out.println(Arrays.toString(manyNames));
}
```

Listing 5.24 *Ausführbar als '***ARRAYSSORTJDK8EXAMPLE***'*

Führen wir das Programm ARRAYSSORTJDK8EXAMPLE aus, so kommt es erwartungsgemäß zu folgender Ausgabe, die derjenigen der Listensortierung entspricht:

```
[Andy, Michael, Stefan, Tim]
[Tim, Andy, Stefan, Michael]
```

5.2.3 Sortieren mit Komparatoren

Im Folgenden wird der Einsatz von Komparatoren etwas genauer betrachtet, um verschiedene Sortierungen und Kombinationen davon zu realisieren. Stellen wir uns vor, wir hätten eine Liste mit `Person`-Objekten, die nach verschiedenen Kriterien sortiert werden sollen, etwa nach Name, Wohnort oder Alter. ***Um die Beispiele einfach halten zu können, gehe ich davon aus, dass die zu sortierenden Collections keine `null`-Werte enthalten und man daher auf `null`-Prüfungen verzichten kann***.

Definition eines Komparators (nach Name)

Zunächst wollen wir Personen nach Namen sortieren. Dazu implementieren wir eine Klasse `PersonNameComparator`, die zur Realisierung die Methode `compareTo(String)` der Klasse `String` verwendet:

```
public final class PersonNameComparator implements Comparator<Person>
{
    public int compare(final Person person1, final Person person2)
    {
        return person1.getName().compareTo(person2.getName());
    }
}
```

Definition eines Komparators (nach Alter)

Schnell kommt der Wunsch auf, Personen auch nach deren Alter sortieren zu können. Für primitive Datentypen existieren keine `compareTo()`-Methoden. Häufig sieht man dann folgende Art der Realisierung des Vergleichs von Hand:

```
public final class PersonAgeComparator implements Comparator<Person>
{
    public int compare(final Person person1, final Person person2)
    {
        if (person1.getAge() < person2.getAge())
        {
            return -1;
        }
        if (person1.getAge() > person2.getAge())
        {
            return 1;
        }
        return 0;
    }
}
```

Diese Implementierung löst das Problem, erfordert allerdings einige Zeilen Sourcecode. Zum Teil sieht man eine Subtraktion der zu vergleichenden Werte:

```
return person1.getAge() - person2.getAge();
```

Diese Form der Berechnung scheint zunächst kürzer und einfacher zu sein. ***Sie ist jedoch zu vermeiden, da sie nicht das zu lösende Problem widerspiegelt***. Darüber hinaus kann diese Art des Vergleichs zu Problemen durch Wertebereichsverletzungen führen, wenn die Differenz zweier Werte aus dem Wertebereich des verwendeten Datentyps `int` herausfällt. Für Altersangaben kann man dies wohl ausschließen.

Nutzt man die Methode `Integer.compare(int, int)`, die einen Rückgabewert kompatibel zu dem von Komparatoren liefert, erhält man als Vorteil eine kurze Realisierung mit einer guten Lesbarkeit und Verständlichkeit:

```
return Integer.compare(person1.getAge(), person2.getAge());
```

Kombination von Sortierkriterien

Sind Sortierungen nach unterschiedlichen Kriterien möglich, so ist der Wunsch nach einer Kombination verschiedener Sortierungen eine naheliegende Folge, etwa nach einer Kombination der Sortierungen erst nach Alter und dann nach Name.

Beim Sortieren anhand verschiedener Kriterien gibt es vorrangige und nachrangige Sortierkriterien. Nur wenn das erste (vorrangige) Kriterium Gleichheit ergibt, wird anhand des nächsten (nachrangigen) Kriteriums weiter sortiert. Im Falle der Sortierung zuerst nach Alter und dann nach Name heißt dies, dass der Name nur dann zum Tragen kommt, wenn das Alter gleich ist. Verallgemeinert gilt, dass bei Gleichheit eines Sortierkriteriums so lange das jeweils folgende Kriterium betrachtet werden muss, bis entweder alle Sortierkriterien ausgewertet sind oder vorher eine Ungleichheit festgestellt wird. Schauen wir zur Verdeutlichung auf eine Realisierung der Kombination der Sortierungen nach Alter und Name:

```
public final class PersonAgeAndNameComparator implements Comparator<Person>
{
    public int compare(final Person person1, final Person person2)
    {
        int ret = Integer.compare(person1.getAge(), person2.getAge());

        if (ret == 0)
        {
            ret = person1.getName().compareTo(person2.getName());
        }

        return ret;
    }
}
```

Häufig sind Anforderungen nicht präzise formuliert. Nehmen wir an, es bestände zusätzlich der Wunsch, auch in der anderen Kombination, d. h. zuerst nach Name und dann nach Alter, sortieren zu können. Die Realisierung dieser Funktionalität ist einfach:

```
public final class PersonNameAndAgeComparator implements Comparator<Person>
{
    public int compare(final Person person1, final Person person2)
    {
        int ret = person1.getName().compareTo(person2.getName());

        if (ret == 0)
        {
            ret = Integer.compare(person1.getAge(), person2.getAge());
        }

        return ret;
    }
}
```

Schnell erkennt man, dass die hier eingesetzten Vergleichsoperationen denen der beiden eingangs vorgestellten Komparatoren sehr ähneln. Diese Duplikation ist hier zwar nicht ausgeprägt, aber trotzdem nicht optimal, wenn man qualitativ hochwertigen Sourcecode schreiben möchte. Was kann man also verbessern?

Zusammengesetzte Komparatoren

Die Klassen `PersonAgeComparator` und `PersonNameComparator` sollten ohne Copy-Paste-Duplikation der entsprechenden Zeilen zum Entwurf von neuen Komparatoren wiederzuverwenden sein. Es sollte eine geschicktere Umsetzung geben, als dies für die zwei bisher vorgestellten Sortierkombinationen der Fall war. Man könnte die bereits definierten Komparatoren wie folgt kombinieren:

```
public final class PersonNameAndAgeComparatorV2 implements Comparator<Person>
{
   private final Comparator<Person> ageComparator  = new PersonAgeComparator();
   private final Comparator<Person> nameComparator = new PersonNameComparator();

   public int compare(final Person person1, final Person person2)
   {
       int ret = nameComparator.compare(person1, person2);

       if (ret == 0)
       {
          ret = ageComparator.compare(person1, person2);
       }

       return ret;
   }
}
```

Diese Technik des Zusammenfassens von einzelnen Komparatoren zu neuen speziellen Komparatoren kann man für beliebige Kombinationen von Sortierkriterien nutzen. Allerdings wird dann für jede Kombination von Sortierkriterien eine eigene Klasse benötigt. Man stößt schnell an Grenzen für sinnvolle Namen für die Klassen. Insgesamt entsteht ein wenig flexibles Design, das sich nur aufwendig und umständlich erweitern lässt. Das erinnert an die kombinatorische Explosion von Klassen durch die Realisierung von Eigenschaften und Verhalten mittels Vererbung (vgl. Abschnitt 3.3.2).

Hintereinanderschaltung von Komparatoren

Der eben beschriebene Ansatz lässt sich praktischerweise leicht in einen flexiblen, erweiterbaren und generischen Ansatz umgestalten – ganz nebenbei werden die besprochenen Nachteile der Namenskonflikte und der explodierenden Klassenhierarchien behoben: Wir entwerfen eine Klasse `PersonUniversalComparator`. Diese verwaltet eine Liste von beliebigen `Comparator<Person>`-Objekten, die hintereinander ausgeführt werden. Die Klasse bietet dazu einen Konstruktor, der eine Liste mit gewünschten `Comparator<Person>`-Objekten entgegennimmt. Weil oftmals genau zwei Komparatoren den Vergleich festlegen, bieten wir einen dafür passenden speziellen Konstruktor an, um die Arbeit für Klienten zu erleichtern. Alternativ können nutzende Klassen beliebige Kombinationen von Komparatoren individuell zusammenstellen. Die Implementierung der Klasse `PersonUniversalComparator` zeigt folgendes Listing:

```
public final class PersonUniversalComparator implements Comparator<Person>
{
    private final List<Comparator<Person>> comparators = new ArrayList<>();

    public PersonUniversalComparator(final List<Comparator<Person>> comparators)
    {
        comparators.addAll(comparators);
    }

    // Convenience
    public PersonUniversalComparator(final Comparator<Person> comparator1,
                                     final Comparator<Person> comparator2)
    {
        comparators.add(comparator1);
        comparators.add(comparator2);
    }

    public int compare(final Person person1, final Person person2)
    {
        int ret = 0;

        for (final Comparator<Person> comparator : comparators)
        {
            ret = comparator.compare(person1, person2);
            if (ret != 0)
                break;
        }

        return ret;
    }
}
```

Diese Realisierung macht den Einsatz für Klienten einfach und flexibel. Betrachten wir dies anhand einer kurzen Kundenliste und der Definition zweier zusammengesetzter Komparatoren. Im folgenden Listing ist lediglich die Definition der Komparatoren gezeigt. Die Arbeitsweise wird deutlich, wenn man das Programm PERSONUNIVERSALCOMPARATOREXAMPLE ausführt.

```
public static void main(final String[] args)
{
    // Bereitstellen der Daten
    final List<Person> customers = new ArrayList<>();
    customers.add(new Person("Werner", "Stuhr", 43));
    customers.add(new Person("Tim", "Kiel", 33));
    customers.add(new Person("Tim", "Aachen", 43));
    customers.add(new Person("Reinhard", "Osterrönfeld", 50));
    customers.add(new Person("Peter", "Oldenburg", 33));

    // Definition von Einzelkomparatoren
    final Comparator<Person> ageComparator = new PersonAgeComparator();
    final Comparator<Person> nameComparator = new PersonNameComparator();

    // Definition zusammengesetzter Komparatoren
    final Comparator<Person> ageAndNameComparator =
                new PersonUniversalComparator(ageComparator, nameComparator);
    final Comparator<Person> nameAndAgeComparator =
                new PersonUniversalComparator(nameComparator, ageComparator);

    // Einsatz der Komparatoren
    final List<Person> ageAndNameSortedList = new ArrayList<>(customers);
```

```
    Collections.sort(ageAndNameSortedList, ageAndNameComparator);
    printCustomerList("Sorted by Age And Name:", ageAndNameSortedList);

    System.out.println("-----------------------------------------------");

    final List<Person> nameAndAgeSortedList = new ArrayList<>(customers);
    Collections.sort(nameAndAgeSortedList, nameAndAgeComparator);
    printCustomerList("Sorted by Name And Age:", nameAndAgeSortedList);
}
```

Listing 5.25 *Ausführbar als* **'PERSONUNIVERSALCOMPARATOREXAMPLE'**

Die Ausgabe des Programms PERSONUNIVERSALCOMPARATOREXAMPLE lässt die Arbeitsweise der zusammengesetzten Komparatoren gut erkennen:

```
Sorted by Age And Name:
Person: Name='Peter' City='Oldenburg' Age='33'
Person: Name='Tim' City='Kiel' Age='33'
Person: Name='Tim' City='Aachen' Age='43'
Person: Name='Werner' City='Stuhr' Age='43'
Person: Name='Reinhard' City='Osterrönfeld' Age='50'
-----------------------------------------------
Sorted by Name And Age:
Person: Name='Peter' City='Oldenburg' Age='33'
Person: Name='Reinhard' City='Osterrönfeld' Age='50'
Person: Name='Tim' City='Kiel' Age='33'
Person: Name='Tim' City='Aachen' Age='43'
Person: Name='Werner' City='Stuhr' Age='43'
```

Umkehren der Sortierreihenfolge per Komparator

Für Tabellen ist es ein üblicher Anwendungsfall, nach beliebigen Kriterien (Spalten) sortieren zu können. Häufig möchte man dabei zudem die Sortierreihenfolge zwischen aufsteigend und absteigend wechseln können.

Eine Realisierungsidee für den Wechsel der Sortierreihenfolge ist, den Rückgabewert eines Komparators zu negieren. Das funktioniert in der Regel.[18] Einfacher und semantisch klarer als durch Negation lässt sich die Umkehrung der Reihenfolge durch die Bibliotheksklasse `ReverseComparator` und die zwei überladenen Hilfsmethoden `reverseOrder()` ausdrücken. Eine davon basiert auf dem Interface `Comparable<T>` und erlaubt, die natürliche Sortierung umzukehren. Die zweite Version ist für das Interface `Comparator<T>` ausgelegt. Damit können Sortierungen invertiert werden.

Wir greifen hier das vorherige Beispiel auf und definieren eine absteigende Sortierung nach Alter und Name wie folgt:

```
// Definition des ReverseComparators
final Comparator<Person> reverseAgeAndNameComparator =
                         Collections.reverseOrder(ageAndNameComparator);
Collections.sort(ageAndNameSortedList, reverseAgeAndNameComparator);
```

Listing 5.26 *Ausführbar als* **'PERSONREVERSECOMPARATOREXAMPLE'**

[18] Die Ausnahme bildet der Wert `Integer.MIN_VALUE`. Für diesen gilt die Anomalie `-Integer.MIN_VALUE == Integer.MIN_VALUE` (vgl. Abschnitt 4.2).

Hiermit beschließe zunächst ich die Ausführungen zu Komparatoren. Mit JDK 8 hat sich in dem Bereich viel getan: Insbesondere in Kombination mit Lambda-Ausdrücken und Methodenreferenzen lassen sich auf einfache Weise selbst komplexere Komparatoren implementieren. Für Details verweise ich auf Abschnitt 15.1.

5.2.4 Filtern von Collections

Das Collections-Framework bietet zwar diverse Algorithmen out-of-the-Box, jedoch fehlt bis JDK 8 das gezielte Filtern von Elementen nach beliebigen Kriterien. Dieser Abschnitt stellt das Thema vor. Damit sind Sie für den Fall gerüstet, mit JDK 7 eine Filterung realisieren zu müssen. Neben Erleichterungen für Komparatoren existieren in JDK 8 mächtige Möglichkeiten zur Filterung mit Streams und dem Filter-Map-Reduce-Framework (vgl. Abschnitt 12.4).

Grundlagen

Der hier beschriebene Ansatz zum Filtern von Collections basiert auf der Definition eines generischen Interface `Filterable<T>`, das eine Methode `accept(T)` enthält. Diese entscheidet, ob ein übergebenes Element akzeptiert wird, und muss für jedes Element der Collection aufgerufen werden.

```
public interface Filterable<T>
{
    boolean accept(final T value);
}
```

Ein Filter, der durch Aufruf der Methode `equals(Object)` auf Gleichheit prüft, kann wie folgt realisiert werden:

```
public class EqualsFilter<T> implements Filterable<T>
{
    private final T acceptedValue;

    public EqualsFilter(final T acceptedValue)
    {
        this.acceptedValue = Objects.requireNonNull(acceptedValue,
                                    "acceptedValue must not be null");
    }

    @Override
    public boolean accept(final T value)
    {
        return acceptedValue.equals(value);
    }
}
```

Um gültige Werte sicherzustellen, setzen wir hier und in den weiteren Beispielen die statische Hilfsmethode `requireNonNull(T, String)` aus der Utility-Klasse `Objects` ein. Die genannte Methode prüft, ob ein übergebener Parameter ungleich `null` ist. Ist dies nicht der Fall, wird eine Exception ausgelöst.

In einer Utility-Klasse `FilterUtils` erstellen wir folgende Methode `applyFilter(List<T>, Filterable<T>)`:

```
public final class FilterUtils
{
    public static <T> List<T> applyfilter(final List<T> values,
                                          final Filterable<T> filter)
    {
        Objects.requireNonNull(values, "values must not be null");
        Objects.requireNonNull(filter, "filter must not be null");

        final List<T> filteredValues = new ArrayList<>();
        for (final T current : values)
        {
            if (filter.accept(current))
                filteredValues.add(current);
        }

        return filteredValues;
    }
}
```

Die Implementierung folgt dem STRATEGIE-Muster (vgl. Abschnitt 18.3.4) und erlaubt dadurch ohne Anpassungen des generellen Ablaufs, verschiedene Filterungen zu realisieren. Dazu wird die übergebene Liste durchlaufen und für jedes Element die durch das übergebene `Filterable<T>`-Objekt beschriebene Filterbedingung ausgewertet. Dann wird gegebenenfalls das entsprechende Element in die Ergebnisliste `filteredValues` aufgenommen.

Folgende `main()`-Methode nutzt die bisher vorgestellte Basisfunktionalität, um aus einer Liste von `Integer`-Objekten dasjenige mit dem Wert `2` herauszufiltern:

```
public static void main(final String[] args)
{
    final List<Integer> intValues = Arrays.asList(1, 2, 3, 4, 5, 6, 7);

    // int-Zahlenfilter auf den Wert 2
    final Filterable<Integer> numberFilter = new EqualsFilter<>(2);
    final List<Integer> filteredValues = FilterUtils.applyFilter(intValues,
                                                                  numberFilter);
    System.out.println(filteredValues);
}
```

Listing 5.27 *Ausführbar als* **'SIMPLEFILTEREXAMPLE'**

Vergleiche

Häufig soll nicht nach einem exakten Wert gefiltert werden, sondern nach Bedingungen oder Wertebereichen. Man möchte etwa alle Werte erhalten, die größer bzw. kleiner als ein bestimmter Wert sind. Für derartige Vergleiche haben wir bereits das Interface `Comparable<T>` kennengelernt.

Zur Realisierung typsicherer Vergleiche verwenden wir eine generische Definition von Filtern, die basierend auf dem Interface `Comparable<T>` eine Typeinschränkung `<T extends Object & Comparable<T>>` (vgl. Abschnitt 3.7.1) festlegen. Dadurch

lassen sich sehr einfach die Vergleiche »größer« bzw. »kleiner« realisieren und in die Bibliothek der angebotenen Filterbedingungen integrieren. Das folgende Listing zeigt exemplarisch die Klasse `Greater` als Implementierung eines »Größer als«-Filters:

```
public class Greater<T extends Object & Comparable<T>> implements Filterable<T>
{
    private final T lowerBound;

    public Greater(final T lowerBound)
    {
        this.lowerBound = Objects.requireNonNull(lowerBound,
                                                 "lowerBound must not be null");
    }

    @Override
    public boolean accept(final T value)
    {
        return lowerBound.compareTo(value) < 0;
    }
}
```

Neben Vergleichen mit einem Grenzwert sind häufig auch Wertebereiche von Interesse. Auch deren Definition wird einfach über folgende Filterklasse `Between` möglich:

```
public class Between<T extends Object & Comparable<T>> implements Filterable<T>
{
   private final T lowerBound;
   private final T upperBound;

   public Between(final T lowerBound, final T upperBound)
   {
      this.lowerBound = Objects.requireNonNull(lowerBound,
                                               "lowerBound must not be null");
      this.upperBound = Objects.requireNonNull(upperBound,
                                               "upperBound must not be null");

      if (!(lowerBound.compareTo(upperBound) <= 0))
          throw new IllegalArgumentException("lowerBound " + lowerBound +
                              " must be <= upperBound " + upperBound);
   }

   @Override
   public boolean accept(final T value)
   {
      return lowerBound.compareTo(value) <= 0 &&
             upperBound.compareTo(value) >= 0;
   }
}
```

Logische Verknüpfungen

Häufig sollen Bedingungen miteinander verknüpft werden. Bei Personen könnte man etwa nach solchen mit dem Namen »Meyer« und einem Alter zwischen 20 und 40 Jahren filtern wollen.

Die logischen Verknüpfungen AND, OR und NOT lassen sich auf einfache Weise als Filter realisieren, wie dies folgendes Listing für die Implementierung von OR zeigt:

```
public class Or<T> implements Filterable<T>
{
    private final Filterable<T> filter1;
    private final Filterable<T> filter2;

    public Or(final Filterable<T> filter1, final Filterable<T> filter2)
    {
        this.filter1 = Objects.requireNonNull(filter1,
                                              "filter1 must not be null");

        this.filter2 = Objects.requireNonNull(filter2,
                                              "filter2 must not be null");
    }

    @Override
    public boolean accept(final T value)
    {
        return (filter1.accept(value) || filter2.accept(value));
    }
}
```

Die Implementierung von AND verwendet lediglich eine andere Verknüpfung (Operator '`&&`') in der `accept(T)`-Methode. Die Filterklasse für NOT benötigt nur einen Eingabeparameter und ist wie folgt realisiert:

```
public class Not<T> implements Filterable<T>
{
    private final Filterable<T> filter;

    public Not(final Filterable<T> filter)
    {
        this.filter = Objects.requireNonNull(filter,
                                             "filter must not be null");
    }

    @Override
    public boolean accept(final T value)
    {
        return !(filter.accept(value));
    }
}
```

In den Implementierungen von AND und OR nutzen wir das KOMPOSITUM-Muster. Zur Realisierung von NOT kommt das DEKORIERER-Muster zum Einsatz. Detaillierte Informationen zu diesen Mustern liefern die Abschnitte 18.2.3 und 18.2.4.

Spezialisierungen für verschiedene Datentypen

Neben diesen allgemeingültigen Filtern ist es für einige Datentypen wünschenswert, spezielle Filter zu definieren. Für Strings kann etwa ein Enthaltensein-Filter wie folgt realisiert werden:

```
public class StringContains implements Filterable<String>
{
    private final String necessarySubstring;

    public StringContains(final String necessarySubstring)
    {
        this.necessarySubstring = Objects.requireNonNull(necessarySubstring,
                                    "necessarySubstring must not be null");
    }

    @Override
    public boolean accept(final String value)
    {
        if (value == null)
            return false;

        return value.contains(necessarySubstring);
    }
}
```

Die Filter im Einsatz

Nachfolgend nutzen wir viele der bisher erstellten Filter und realisieren damit zwei komplexere Filterbedingungen: Durch die Hintereinanderschaltung mehrerer Filterbedingungen kann man Wertebereiche invertieren oder Filter kombinieren, z. B. die Filter »Wertebereich 3 – 7« sowie »Werte größer als 12«:

```
public static void main(final String[] args)
{
    final List<Integer> intValues = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                                                  11, 12, 13, 14, 15);

    // Ermittle alle Werte, die "NICHT im Bereich 4-11" liegen
    final Filterable<Integer> notRange4_11 = new Not<>(new Between<>(4,11));
    final List<Integer> filteredValues1 =
                        FilterUtils.applyFilter(intValues, notRange4_11);
    System.out.println(filteredValues1);

    // Ermittle alle Werte, die "im Bereich 3-7 liegen oder größer als 12" sind
    final Filterable<Integer> range2_7OrGreater12 = new Or<>(
                                                    new Between<>(3,7),
                                                    new Greater<>(12));
    final List<Integer> filteredValues2 =
                        FilterUtils.applyFilter(intValues, range2_7OrGreater12);
    System.out.println(filteredValues2);
```

Listing 5.28 *Ausführbar als* **'SIMPLEFILTEREXAMPLE2'**

Erwartungsgemäß kommt es zu folgenden Ausgaben für die Ausgangsdaten 1 – 15:

```
[1, 2, 3, 12, 13, 14, 15]
[3, 4, 5, 6, 7, 13, 14, 15]
```

Erweiterung

In der Regel sind in Collections nicht, wie bisher dargestellt, Objekte von Klassen gespeichert, die sich derart einfach filtern lassen. Zum einen implementieren die Klassen möglicherweise das Interface `Comparable<T>` nicht. Zum anderen möchte man vielfach nicht auf Klassenbasis, sondern feingranular auf Basis der Wertebelegungen verschiedener Attribute filtern. In beiden Fällen benötigt man dafür in den Filtern Zugriffe auf Attribute.

Betrachten wir folgende Klasse `SimplePerson`, um das Filtern nach verschiedenen Kriterien nachzuvollziehen:

```
public final class SimplePerson
{
    private final String name;
    private final int age;

    public SimplePerson(final String name, final int age)
    {
        this.name = Objects.requireNonNull(name, "name must not be null");
        this.age = age;
    }

    @Override
    public String toString()
    {
        return "SimplePerson [age=" + age + ", name=" + name + "]";
    }

    public int getAge()
    {
        return age;
    }
    // ...
```

Um die bisher definierten Filter wiederverwenden zu können, benötigen wir für jedes Attribut einen speziellen ***Attributermittlungsfilter***. Im folgenden Listing ist ein solcher für das Attribut `age` gezeigt – eine Realisierung für das Namensattribut erfolgt analog.

```
public class SimplePersonAgeFilter implements Filterable<SimplePerson>
{
    private final Filterable<Integer> intFilter;

    public SimplePersonAgeFilter(final Filterable<Integer> intFilter)
    {
        this.intFilter = Objects.requireNonNull(intFilter,
                                                "intFilter must not be null");
    }

    @Override
    public boolean accept(final SimplePerson person)
    {
        return intFilter.accept(person.getAge());
    }
}
```

Durch den Einsatz dieser Attributermittlungsfilter kann man auf einfache Weise entsprechende `SimplePerson`-Objekte mit dem Namen »Meyer« und einem Alter zwischen 20 und 40 Jahren ermitteln:

```
public static void main(final String[] args)
{
   final SimplePerson demo1 = new SimplePerson("Meyer1", 11);
   final SimplePerson demo2 = new SimplePerson("Meyer2", 22);
   final SimplePerson demo3 = new SimplePerson("Meyer3", 33);
   final SimplePerson demo4 = new SimplePerson("Meyer4", 44);
   final SimplePerson demo5 = new SimplePerson("Müller", 34);

   final List<SimplePerson> demoDataValues = Arrays.asList(demo1, demo2, demo3,
                                                           demo4, demo5);

   // Filter für "Alter 20 - 40"
   final Filterable<SimplePerson> ageRangeFilter =
                           new SimplePersonAgeFilter(new Between<>(20, 40));

   // Filter für "Name enthält Meyer"
   final Filterable<SimplePerson> nameFilter =
                     new SimplePersonNameFilter(new StringContains("Meyer"));

   // Kombination der Filter
   final Filterable<SimplePerson> ageAndNamefilter = new And<>(nameFilter,
                                                              ageRangeFilter);

   final List<SimplePerson> filteredValues =
                           FilterUtils.applyFilter(demoDataValues,
                                                   ageAndNamefilter);
   System.out.println(filteredValues);
}
```

Listing 5.29 *Ausführbar als '***FILTEREXAMPLE***'*

Erwartungsgemäß kommt es zu folgender Ausgabe:

```
[SimplePerson [age=22, name=Meyer2], SimplePerson [age=33, name=Meyer3]]
```

Fazit

Dieser Abschnitt hat einen kurzen Einstieg in das Thema Filtern von Datenstrukturen gegeben. Dabei haben wir erfahren, wie durch den Einsatz von problemangepassten Entwurfsmustern kurze, verständliche und leicht erweiter- und kombinierbare Filterklassen realisiert werden können. Bitte bedenken Sie aber, dass mit JDK 8 vielfältige Filtermöglichkeiten bereitgestellt werden (vgl. Abschnitt 12.4).

5.3 Utility-Klassen und Hilfsmethoden

Das Collections-Framework bietet mit den Klassen `Collections` und `Arrays` zwei mächtige Utility-Klassen an, die diverse Algorithmen und Erweiterungen bereitstellen, von denen einige wichtige im Folgenden kurz vorgestellt werden.

5.3.1 Nützliche Hilfsmethoden

In diesem Abschnitt wollen wir einen Blick auf verschiedene Methoden rund um die Erzeugung von Collections werfen.

Listen befüllen und `Arrays.asList()`

Manchmal soll eine Liste aus einer Menge an vordefinierten Werten erstellt werden. Eine zur Initialisierung gebräuchliche Realisierung ist folgende:

```
final List<String> names = new ArrayList<>();
names.add("Max");
names.add("Michael");
names.add("Carsten");
```

Diese Schreibweise ist recht geschwätzig, da der Name der Collection-Variable immer wieder beim Hinzufügen angegeben werden muss. Etwas eleganter kann man das Ganze schreiben, wenn man zwei Java-Techniken kombiniert, nämlich die Definition einer anonymen Klasse und einen Initializer-Block:

```
final List<String> names = new ArrayList<String>()
{{
    add("Max");
    add("Michael");
    add("Carsten");
}};
```

Damit erspart man sich, den Namen der Collection ständig wiederholen zu müssen. Noch besser lesbar kann man diese Befüllung mit Daten durch Aufruf der statischen Methode `Arrays.asList(T...)` folgendermaßen schreiben:

```
final List<String> names = Arrays.asList("Max", "Michael", "Carsten");
```

Hier nutzt man, dass Java seit Version 5 das Sprachfeature Varargs bietet, wodurch die kommaseparierte Angabe beliebig vieler Werte als Parameter möglich wird. Allerdings muss man zwei Besonderheiten beachten, auf die ich nun eingehe.

Hinweis: Syntaktische Besonderheiten im Beispiel

Zur Konstruktion einer Liste mit einer Menge an vordefinierten Werten haben wir eben verschiedene Varianten gesehen. Auf Variante zwei möchte ich noch einmal explizit eingehen:

```
// Drei Besonderheiten
final List<String> names = new ArrayList<String>() // #1
{{ // #2
    add("Max");      // statt names.add("Max"); #3
    add("Michael");
    add("Carsten");
}}; // #2
```