6 Das Collections-Framework

In diesem Kapitel beschreibe ich das Collections-Framework, 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 6.1 beschreibt in der Praxis relevante Datenstrukturen und zeigt kurze Nutzungsbeispiele. Auf gebräuchliche Anwendungsfälle wie Suchen und Sortieren gehe ich in Abschnitt 6.2 ein. Diverse weitere nützliche Funktionalitäten werden durch die zwei Utility-Klassen `Collections` und `Arrays` bereitgestellt und in Abschnitt 6.3 beschrieben. Abschnitt 6.4 beschäftigt sich mit dem Zusammenspiel von Generics und Collections und zeigt, welche Dinge vor allem in Kombination mit Vererbung beachtet werden sollten. Darüber hinaus widmet sich Abschnitt 6.5 der Klasse `Optional<T>` zur Modellierung optionaler Werte. Abschließend werden in Abschnitt 6.6 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 9.6.1 ein. Letzteres wird in Kapitel 22 behandelt.

6.1 Datenstrukturen und Containerklasse

Im Collections-Framework werden Listen, Mengen und Schlüssel-Wert-Abbildungen durch sogenannte ***Containerklassen*** realisiert. Diese heißen so, weil sie Objekte anderer Klassen speichern und verwalten. Als Basis für die Containerklassen dienen die Interfaces `List<E>`, `Set<E>` bzw. `Map<K,V>` aus dem Package `java.util`.

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 speichern Objektreferenzen. Die Verwaltung primitiver Typen ist dort nur möglich, wenn diese in ein Wrapper-Objekt (wie `Byte`, `Integer` oder `Double`) umgewandelt werden. Durch das Auto-Boxing (vgl. Abschnitt 4.2.1) geschieht dies oft automatisch.

6.1.1 Wahl einer geeigneten Datenstruktur

Um Daten in eigenen Applikationen 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 disjunkte 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 Entscheidung zwischen Listen und Mengen mit dem Basisinterface `Collection<E>` sowie Schlüssel-Wert-Abbildungen mit dem Basisinterface `Map<K,V>` zu treffen. Anschließend gilt es, eine geeignete konkrete Realisierung zu finden. Dazu gebe ich nachfolgend 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` kann man eine Implementierung des Interface `Collection<E>` wählen 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 6-1 zeigt die Typhierarchie von Listen und Mengen, wobei aus Gründen der Übersichtlichkeit die generische Definition nicht dargestellt wird.

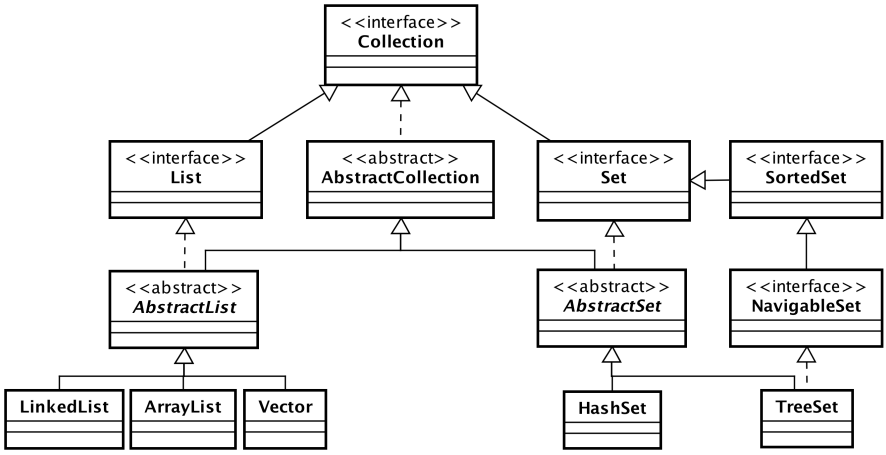


Abbildung 6-1 `Collection`*-Hierarchie*

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 einem Mapping 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 6-2 (auch hier wieder ohne generische Typinformation).

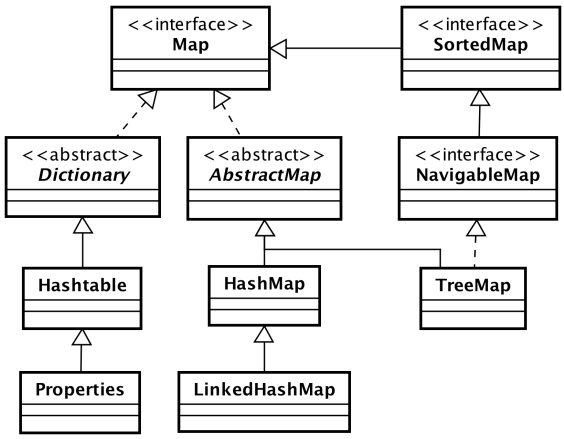


Abbildung 6-2 *`Map`-Hierarchie*

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. Das bedeutet, 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`.

6.1.2 Arrays

Arrays sind Datenstrukturen, die in einem zusammenhängenden Speicherbereich entweder Werte eines primitiven Datentyps oder Objektreferenzen verwalten können. Das ist 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];                    // Definition ohne Daten
final String[] names1 = new String[] { "Tim", "Mike" }; // Normalschreibweise
final String[] names2 = { "Tim", "Mike" };              // Kurzschreibweise
```

Ein Array stellt lediglich einen einfachen Datenbehälter bereit, dessen Größe durch die Initialisierung vorgegeben 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. Folgendes Beispiel des Einlesens von Personendaten aus einer Datenbank verdeutlicht das Beschriebene, wobei initial Platz für 1.000 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. Arrays eignen sich damit ganz speziell 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 eine fixe Größe festgelegt, die das Fassungsvermögen (auch ***Kapazität*** genannt) bestimmt – für eine Größenänderung muss ein neues Array erzeugt werden. 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 für den Fall, dass die Anzahl der zu speichernden Daten im Voraus schlecht schätzbar ist oder variiert, etwa bei Suchen.
- Anhand der Größe eines Arrays kann man 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 und Verwaltung von Eingabewerten nutzen und Daten über folgende Methode `storeValue(byte)` einlesen:

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

Werden viele Daten gespeichert, so kann eine anfangs gewählte Array-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 Methoden in der Utility-Klasse `Arrays` sind praktisch: Sie erlauben es, Aufgaben auf einer höheren Abstraktionsebene als mit `System.arraycopy()` zu beschreiben.

6.1.3 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), die der Collection alle übergebenen Elemente hinzufügt.

Im Interface `Collection<E>` nutzen einige Methoden den Typparameter `Object` oder den Typplatzhalter '`?`' 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 alle angegebenen 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 entweder `E` oder ein Subtyp davon 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] Ansonsten wäre dies für eine Enthaltensein-Prüfung eine zu starke Einschränkung gewesen.

Mengenoperationen auf Collections

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 leichteren Verständnis ist die Arbeitsweise in Abbildung 6-3 für die Collections A, B und C visualisiert.

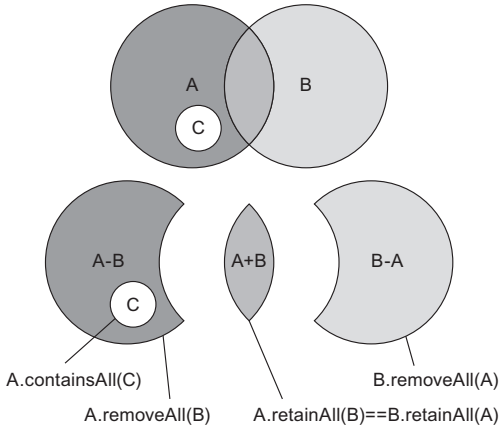


Abbildung 6-3 *Mengenoperationen auf Collections*

Erweiterung im Interface `Collection<E>` in JDK 8

Seit JDK 8 kann man mit der Methode `removeIf(Predicate<T>)` aus einer Collection diejenigen Elemente entfernen, die einer übergebenen Bedingung entsprechen. Das wollen wir am Beispiel einer Liste von Namen kennenlernen. Aus dieser sollen diejenigen Einträge herausgelöscht werden, die einen Leereintrag darstellen:

```
public static void main(final String[] args)
{
   final List<String> names = new ArrayList<>();
   names.add("Max");
   names.add("");                    // Leereintrag
   names.add("Andy");
   names.add("   ");                  // potenziell auch ein "Leereintrag"
   names.add("Stefan");

   names.removeIf(String::isEmpty) // Löschaktionen ausführen
   System.out.println(names);
}
```

Listing 6.1 *Ausführbar als* '**REMOVEIFEXAMPLE**'

Starten wir das Programm REMOVEIFEXAMPLE, so wird die beschriebene Löschoperation ausgeführt und es kommt zu folgender Ausgabe:

```
[Max, Andy,    , Stefan]
```

Wir sehen, dass der Whitespace-Eintrag in der Liste verblieben ist. Eine minimal komplexere Bedingung hilft, auch diesen Eintrag zu entfernen:

```
names.removeIf(str -> str.trim().isEmpty());
```

Die gezeigte Umsetzung birgt jedoch die Gefahr von `NullPointerException`s, wenn die Eingabewerte auch den Wert `null` enthalten können. Statt den Lambda etwas komplexer zu gestalten, wollen wir später in Abschnitt 6.1.5 als Alternative und zur Korrektur die mit JDK 8 im Interface `List<E>` neu eingeführte Methode `replaceAll(UnaryOperator<T>)` verwenden.

6.1.4 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 6.3.1) wandeln zu können. Das Ergebnis ist eine typisierte, aber insbesondere auch unmodifizierbare `List<E>`. Von dieser erhalten wir durch Aufruf von `iterator()` einen `Iterator<E>`. Mit dessen Methode `hasNext()` kann man ermitteln, ob noch weitere Elemente zum Durchlaufen vorhanden 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.println(it.next());
    }
}
```

Listing 6.2 *Ausführbar als* **'ITERATIONEXAMPLE'**

Das Programm ITERATIONEXAMPLE gibt alle Elemente nacheinander aus:

```
Durchlauf
mit
Iterator
```

Löschfunktionalität im Interface `Iterator<E>`

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. Dieses Verhalten wird seit JDK 8 in Form einer Defaultmethode vorgegeben.

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 = { "Andy", "Carsten", "Clemens", "Mike", "Merten" };

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

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

Listing 6.3 *Ausführbar als '***ITERATORCOLLECTIONREMOVEEXAMPLE***'*

Man würde erwarten, dass als Ergebnis die Namen "Andy", "Carsten", "Clemens" 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 nebenläufigen 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 ConcurrentModification-Exception 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<E>`

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.

6.1.5 Listen und das Interface `List`

Unter einer Liste versteht man eine über ihre Position geordnete Folge von Elementen – dabei können auch identische Elemente mehrfach vorkommen. 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]

Das Interface `List<E>`

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.

[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 `UnsupportedOperationExceptions`. Weitere Informationen finden Sie in Abschnitt 6.3.2.

- `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. Die Gleichheit zwischen dem Suchelement und den einzelnen Elementen der Liste 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 6.4 *Ausführbar als* **'FIRSTLISTEXAMPLE'**

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

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

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. Dadurch kann man beispielsweise innerhalb eines gewissen Bereichs

[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.

eine Suche durchführen oder diesen Bereich löschen. ***Dabei sollte man allerdings beachten, dass lediglich eine Sicht auf die ursprüngliche Liste geliefert wird***. Somit kommt es bei Veränderungen an der Teilliste auch zu Änderungen in der ursprünglichen Liste. Ändert man jedoch in der Originalliste, so kommt es zu einer `ConcurrentModificationException`. Folgendes Programm demonstriert dieses Verhalten:

```
public static void main(final String[] args)
{
    final List<String> original = new ArrayList<>(Arrays.asList(
                                        "ABC", "DEF", "GHI",
                                        "JKL", "MNO", "PQR"));

    final List<String> first3 = original.subList(0, 3);

    first3.remove(1);
    printLists(original, first3);

    original.add("XXX");  // Führt später zur Exception
    printLists(original, first3);
}

private static void printLists(final List<String> original, final List<String>
      first3)
{
    System.out.println("Original: " + original);
    System.out.println("SUblist:  " + first3);
}
```

Listing 6.5 *Ausführbar als* **'SUBLISTEXAMPLE'**

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

```
Original: [ABC, GHI, JKL, MNO, PQR]
SUblist:  [ABC, GHI]
Original: [ABC, GHI, JKL, MNO, PQR, XXX]
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.ArrayList$SubList.checkForComodification(ArrayList.java:1231)
    at java.util.ArrayList$SubList.listIterator(ArrayList.java:1091)
```

Das Interface `ListIterator<E>`

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 allerdings während einer Iteration manipulieren will, muss man diese modifizierenden Methoden im `ListIterator<E>` 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<E>` und `Vector<E>`

Die Klassen `ArrayList<E>` und `Vector<E>` verwenden zur Datenspeicherung intern 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 Größe 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 oftmals ein entscheidender Vorteil gegenüber dem Einsatz von Arrays darstellt, 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 6-4 skizziert, wobei die Texte `Obj 1`, `Obj 2` usw. nicht das Objekt selbst, sondern die Referenz darauf repräsentieren.

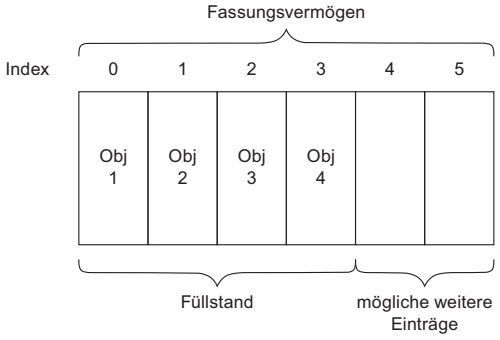
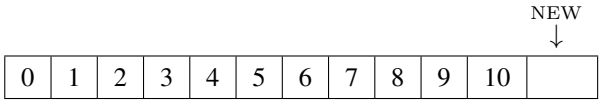


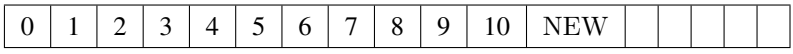
Abbildung 6-4 *Array der Klasse* `ArrayList`

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

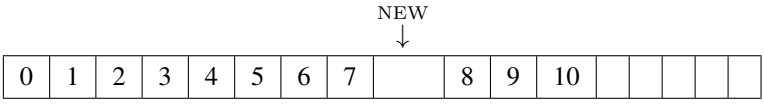
Element an letzter Position hinzufügen – `add(element)` Nehmen wir an, es ist (gerade) noch ausreichend Kapazität im Array vorhanden und es soll ein Element als letztes Element in die Datenstruktur eingefügt werden. In diesem Fall muss nur die Referenz in dem Array gespeichert werden:



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 – das ist aber oftmals kaum messbar. Allerdings muss der Garbage Collector (vgl. Abschnitt 10.4) die obsoleten Arrays wieder wegräumen: Das ist Arbeit, die sich häufig verringern oder vermeiden lässt, indem man ***die erwartete Maximalgröße***

als Konstruktorparameter übergibt. Jedoch ist die Wahl einer geeigneten Größe, wie bereits in Abschnitt 6.1.2 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 Daten speichernden 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.

`ArrayList<E>` oder `Vector<E>`? 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 9). Somit gibt es eher selten Anwendungsfälle für einen `Vector<E>` und in der Regel sollte man die ***`ArrayList<E>` bevorzugen***.

[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.

Arbeitsweise der Klasse `LinkedList<E>`

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 6-5, wobei `Obj 1`, `Obj 2` usw. Referenzen auf Objekte repräsentieren.

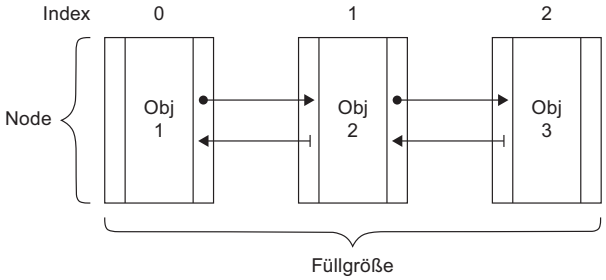


Abbildung 6-5 *Schematischer Aufbau eines Objekts der Klasse* `LinkedList`

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 beliebigen 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 noch 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.

Erweiterungen im Interface `List<E>` in JDK 8

Auch das Interface `List<E>` wurde mit JDK 8 erweitert. Nachfolgend betrachten wir die Methode `replaceAll(UnaryOperator<T>)`. Diese ermöglicht es, für alle Elemente einer `Collection<E>` eine Aktion auszuführen: Jedes Element wird durch den Rückgabewert der Implementierung der Methode `apply(T)` des funktionalen Interface `UnaryOperator<T>` ersetzt. Durch die Anweisungen der Realisierung wird auch die Entscheidung getroffen, welche Elemente wie bearbeitet werden sollen. Im Speziellen müssen nicht immer alle Elemente tatsächlich auch verändert werden. Das »`All`« im Namen bezieht sich also lediglich darauf, dass `apply(T)` für alle Elemente der Liste aufgerufen wird.

Nach diesen zuvor eher theoretischen Details kommen wir auf ein konkretes Anwendungsbeispiel: Nehmen wir an, wir würden von einer externen Datenquelle oder dem GUI eine Liste von Eingabewerten erhalten. Oftmals entsprechen solche Eingaben nicht den Erwartungen und verstoßen gegen Regeln, so sind Einträge beispielsweise leer, bestehen nur aus Leerzeichen oder enthalten diese am Anfang oder Ende. All dies erschwert die weitere Bearbeitung. Die Grundlagen für eine Korrekturfunktionalität, die derartige Werte umwandelt oder herausfiltert, haben wir bereits kennengelernt. Wir müssen das Ganze nur noch geeignet kombinieren:

```
public static void main(final String[] args)
{
    final List<String> names = createDemoNames();

    // Spezialbehandlung von null-Werten
    final UnaryOperator<String> mapNullToEmpty = str -> str == null ? "" : str;
    names.replaceAll(mapNullToEmpty);

    // Leerzeichen abschneiden
    names.replaceAll(String::trim);

    // Leereinträge herausfiltern
    names.removeIf(String::isEmpty);

    System.out.println(names);
}

private static List<String> createDemoNames()
{
   final List<String> names = new ArrayList<>();
   names.add("  Max");
   names.add("");             // Leereintrag
   names.add("  Andy ");
   names.add("   ");           // potenziell auch ein "Leereintrag"
   names.add("Stefan  ");
   return names;
}
```

Listing 6.6 *Ausführbar als* **'REPLACEALLEXAMPLE'**

Mit dieser Erweiterung werden nicht nur potenzielle `null`-Einträge entfernt, sondern auch diejenigen Einträge, die Leerstrings oder lediglich Leerzeichen enthalten. Außerdem werden Leerzeichen am Anfang und Ende von Einträgen gelöscht. Die Ausgabe des Programms REPLACEALLEXAMPLE verdeutlicht dies:

```
[Max, Andy, Stefan]
```

6.1.6 Mengen und das Interface `Set`

Zum Einstieg in das Collections-Framework haben wir uns zunächst ausführlich mit Listen beschäftigt, kommen wir nun zu Mengen und dem Interface `Set<E>`. Das mathematische Konzept der Mengen besagt, dass diese keine Duplikate enthalten. Das Interface `Set<E>` basiert auf dem Interface `Collection<E>`. 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 liegenden 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

Für einen ersten Zugang zum Thema Mengen nutzen wir mit `HashSet<E>` und `TreeSet<E>` zwei gebräuchliche Implementierungen des Interface `Set<E>` und füllen diese 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<>();
    addStringDemoData(hashSet);
    System.out.println(hashSet);

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

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

    // StringBuilder implementiert Comparable nicht
    final Set<StringBuilder> treeSetSurprise = new TreeSet<>();
    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 6.7 *Ausführbar als* '**FIRSTSETEXAMPLE**'

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

```
[Hallo, Welt]
[Welt, Welt, Hallo]
[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: Bei Strings funktioniert alles wie erwartet, für den Typ `StringBuilder` werden Duplikate nicht erkannt und für das `TreeSet<StringBuilder>` wird 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>` dient die Methode `hashCode()` zum Klassifizieren von Elementen in Form eines `int`-Zahlenwerts und die Methode `equals(Object)` zum Auffinden. Die Klasse `TreeSet<E>` nutzt dazu die Methoden `compareTo(T)` bzw. `compare(T, T)` aus den Interfaces `Comparable<T>` bzw. `Comparator<T>`. Abschnitt 6.1.9 geht auf das Zusammenspiel der relevanten Methoden im Detail ein. In den Abschnitten 6.1.7 und 6.1.8 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 einfache Beispiele für `HashSet<E>` und `TreeSet<E>` betrachten, um ein Gefühl für die Arbeit mit Mengen zu erhalten.

Fallstrick: Fehlende Angabe eines Sortierkriteriums

Zur Kompilierzeit wird für ein `TreeSet<E>` nicht geprüft, ob dort 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<E>`

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 6.1.10 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 6.8 *Ausführbar als* **'HASHSETSTORAGEEXAMPLE'**

Bei einem Blick auf die Ausgabe des Programms HASHSETSTORAGEEXAMPLE scheint ein `HashSet<Integer>` die eingefügten Werte zu sortieren:

```
Initial: [1, 2, 3]
```

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 6.9 *Ausführbar als* **'HASHSETSTORAGEEXAMPLE2'**

Die Ausgabe des Programms HASHSETSTORAGEEXAMPLE2 wirkt zufällig, ist aber durch die Verteilung im Container verursacht:

```
Initial: [1, 2, 3]
Add: [1, 33, 2, 3, 22, 11]
```

Die Klasse `TreeSet<E>`

Mitunter benötigt man eine Ordnung der Elemente. Dann bietet sich der Einsatz der Klasse `TreeSet<E>` an, die das Interface `SortedSet<E>` implementiert. Die Sortierung der Elemente wird entweder durch das Interface `Comparable<T>` oder einen explizit im Konstruktor von `TreeSet<E>` übergebenen `Comparator<T>` festgelegt. Wir schauen uns ein ähnliches Beispiel an 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 6.10 *Ausführbar als* **'TREESETSTORAGEEXAMPLE'**

Weil im Konstruktor kein `Comparator<T>` übergeben wurde, basiert die Sortierung auf `Comparable<T>`, das von der zu speichernden Klasse `Integer` erfüllt wird und eine sogenannte natürliche Ordnung (realisiert über eine Vergleichsfunktion durch die Methode `compareTo(T))` bereitstellt. Das lässt sich durch einen Start des Programms TREESETSTORAGEEXAMPLE nachvollziehen:

```
Initial: [1, 2, 3]
Add: [1, 2, 3, 11, 22, 33]
```

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 6.11 *Ausführbar als* **'TREESETSTORAGEEXAMPLE2'**

Die Ausgabe des Programms TREESETSTORAGEEXAMPLE2 demonstriert die Arbeitsweise der obigen Methoden:

```
Initial: [1, 2, 3, 11, 22, 33]
first: 1
last: 33
headSet:   [1, 2, 3]
tailSet:   [11, 22, 33]
subSet:    [11, 22]
headSet:   [1, 2, 6]
numberSet: [1, 2, 6, 11, 22, 33]
```

6.1.7 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()`, die einen `int` zurückliefert:

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 `int`-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. Eine gleichmäßige Verteilung von Objekten auf Buckets erreicht man, wenn die `hashCode()`-Methode für verschiedene Objekte möglichst verschiedene Werte zurückgibt. Idealerweise bildet man die Attribute selbst wieder auf Zahlen ab und multipliziert diese mit Primzahlen, 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 Praxishinweis).

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 Hashcontainer 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 gewöhnlich eine Liste, in der Elemente abgelegt werden.[5]

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

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

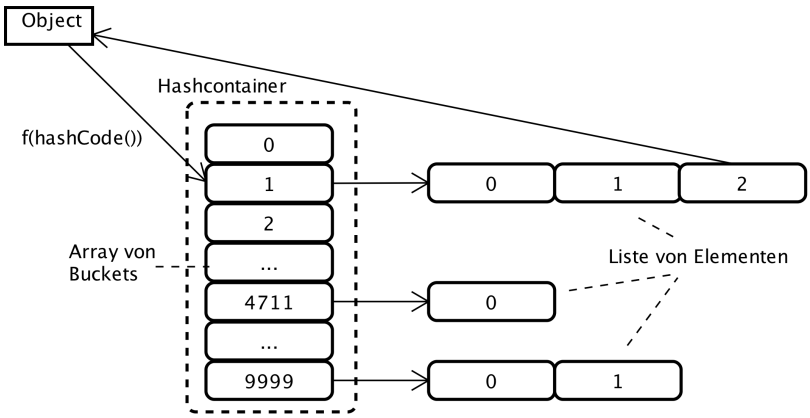


Abbildung 6-6 *Aufbau von hashbasierten Containern*

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.
- Würden alle Objekte lediglich wenige unterschiedliche Bucket-Nummern (oder gar dieselbe) zurückliefern, so würde keine einigermaßen gleichmäßige Verteilung

[5] Zur Speicherung wird zwar in der Regel eine Liste verwendet – als Optimierung wird seit JDK 8 ein Baum genutzt, sofern die zu speichernden Typen das Interface `Comparable<T>` erfüllen.

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 bzw. idealerweise auf einen Baum reduziert werden – zusätzlich aber mit deutlichem Verwaltungsoverhead. Trotzdem würde das Programm noch funktionieren, nur recht inperformant.

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 Collection 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.[a] 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.

[a] Für die Elemente von Listen kann man zwar auch `hashCode()` implementieren, hier hat es jedoch keinen Einfluss auf die Speicherung innerhalb der Liste.

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 bereits in Abschnitt 6.1.6 vorgestellten `HashSet<E>`, nachfolgend also `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 6.12 *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 unterschiedliche 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. Betrachten wir dazu zunächst 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 (Java Language Specification) mit folgender Signatur definiert:

```
public int hashCode()
```

Eine Implementierung von `hashCode()` sollte folgende Eigenschaften erfüllen:[6]

- **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.[7] Wir hatten bereits angesprochen, dass wir Objekte in Hashcontainern nicht mehr wiederfinden, wenn nach einer Änderung des Hashwerts im falschen Bucket 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, um zu 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)`:

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

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

```
@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);
}
```

In der Realisierung der Methode `hashCode()` können wir in diesem Fall die Attribute `wert` und `farbe` zur Berechnung verwenden. Um die Verteilung auf die Buckets möglichst gut zu streuen, kann man mit Primzahlen als Multiplikatoren arbeiten: 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 gleichmäßige Verteilung – für komplexere Klassen wird die Implementierung der Methode `hashCode()` aber 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.

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

In den vorherigen Auflagen dieses Buchs habe ich basierend auf dem in Abschnitt 4.2.4 gewonnenen Wissen über Zahlen und Operationen eine Utility-Klasse `HashUtils` entwickelt. Weil aber seit Java 7 im JDK selbst eine adäquate und einfach zu nutzende Realisierung existiert und man zudem Standards eigenen Entwicklungen vorziehen sollte, zeige ich hier die Methode `hash()` aus der Utility-Klasse `java.util.Objects`.

Einsatz der Utility-Klasse Für die Klasse `Spielkarte` nutzen wir die Klasse `Objects`, um die Berechnung in `hashCode()` 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 etwa 75 % sind Kollisionen erfahrungsgemäß eher 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 so groß wie 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 kleiner 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. ***Nachdem eine Größenanpassung erfolgt ist, 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 vom Hashcontainer zu jedem Element dessen zuvor über die Methode `hashCode()` berechneter 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 Detail zu bedenken: Falls in einem Hashcontainer irgendwann einmal sehr viele Elemente gespeichert wurden, kam es als Folge höchstwahrscheinlich zu einigen Vergrößerungsschritten und damit auch Rehashing-Vorgängen. Werden später Elemente gelöscht, so wird die Größe der Hashtabelle nicht automatisch verkleinert und der Speicher bleibt (unnütz) belegt. Schlimmer noch: ***Im Gegensatz 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.

6.1.8 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 6.2.2).

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<T>`** – 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.[8]
- **Weitere Ordnungen und `Comparator<T>`** – 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 Objekte von Klassen sortieren, für die keine natürliche Ordnung definiert ist, weil das Interface `Comparable<T>` nicht implementiert wird.

Sortierungen und das Interface `Comparable<T>`

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:

[8] 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.

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

Das Vorzeichen des Rückgabewerts bestimmt die Reihenfolge der Elemente:

- $= 0$: 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 für die folgende Klasse `Person`. Dort wird anstatt des Geburtsdatums als 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 (leicht unintuitiv) ausschließlich über deren Namen realisieren:

```
@Override
public int compareTo(final Person otherPerson)
{
    return getName().compareTo(otherPerson.getName());
}
```

Hier ist hilfreich, dass die für den Namen genutzte Klasse `String` das Interface `Comparable<String>` erfüllt. Zudem benötigen wir keine `null`-Prüfung, weil laut Kontrakt eine `NullPointerException` ausgelöst werden soll, sofern ein Aufruf von `compareTo(null)` erfolgt.

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 ein wenig Überlegen klar: Herr Müller aus Kiel ist nicht Herr Müller aus Bremen. Wie geht es also besser? Einen guten Anhaltspunkt stellt oft die Methode `equals(Object)` und 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.[9] Für die Attribute kann man wie folgt vorgehen:

1. Referenztypen, die das Interface `Comparable<T>` implementieren, verwenden deren `compareTo(T)`-Methoden.
2. Für primitive Datentypen lassen sich die Vergleichsoperatoren '<', '=' und '>' einsetzen, etwa für einen Vergleich des Attributs `age`.[10] Statt jedoch 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 den Sourcecode der Klasse des Attributs im Zugriff hat, kann man diese

[9] 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.

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

derart erweitern, dass sie das Interface `Comparable<T>` erfüllt und dort 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)` für beliebige `x` und `y` gleichen Typs 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 zum einen Konsistenzprobleme zwischen beiden Methoden und führt zum anderen dazu, dass die Vergleichslogik nur einmal in der Me-

thode `compareTo(T)` realisiert wird. Dies ist wiederum hilfreich, wenn Änderungen an der Klasse erfolgen, etwa Attribute hinzugefügt werden. Schnell wird dabei 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« [38] 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<T>`

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<T>` 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`.
- > 0: Bei positiven Rückgabewerten ist das erste Objekt `o1` größer als das zweite Objekt `o2`.

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());
    }
}
```

Vereinfachungen mit JDK 8 In Java 8 wurde das Interface `Comparator<T>` stark erweitert und bietet nun diverse Möglichkeiten zur Erzeugung von Komparatoren. Zunächst einmal kann man nun einen Lambda wie folgt nutzen:

```
final Comparator<Person> nameComparator = (person1, person2) ->
{
    return person1.getName().compareTo(person2.getName());
};
```

Noch prägnanter geht es mit den neuen Konstruktionsmethoden, etwa `comparing()`, wie folgt:

```
final Comparator<Person> nameComparator = Comparator.comparing(Person::getName);
```

Die Thematik schauen wir uns detailliert in Abschnitt 6.2.4 an.

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 auf dem Weg von der Wurzel bis zu einem Knoten ohne Nachfahren (auch ***Blatt*** genannt) definiert. Per Definition werden 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.[11]

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 6-7.

[11] 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.

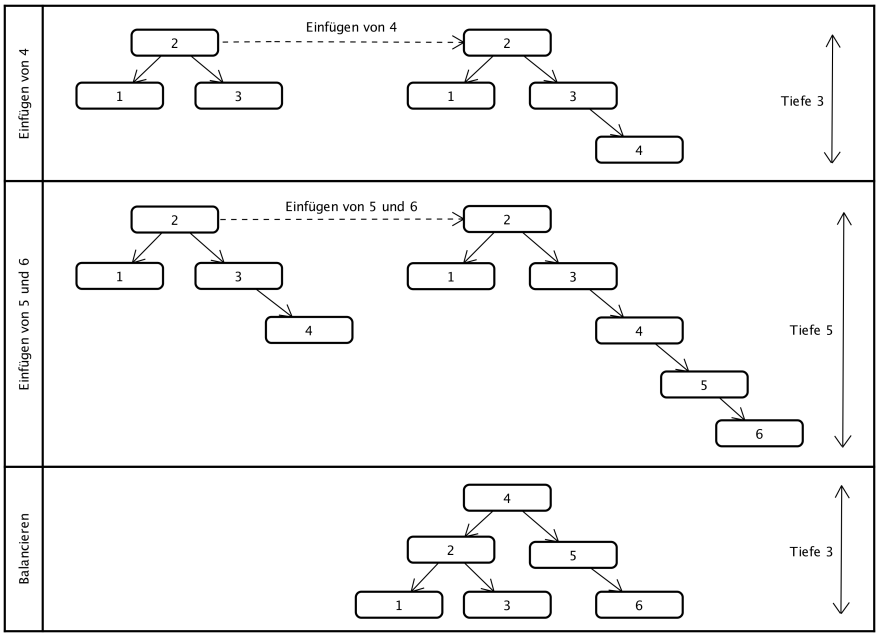


Abbildung 6-7 *Arbeitsweise eines balancierten Baums*

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.

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

Nachdem wir nun ein gutes Verständnis zu `HashSet<E>` und `TreeSet<E>` aufgebaut haben, wollen wir nochmal auf Besonderheiten der drei Methoden `equals(Object)`, `hashCode()` und `compareTo(T)` eingehen. Dabei ist es 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.[12] 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 6.13 *Ausführbar als* **'LAWOFBIG3EXAMPLE'**

[12] 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)`[13] nicht mit `equals(Object)` kompatibel ist, wie dies in Abschnitt 6.1.8 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 6.14 *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 6.1.7 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 6.15 *Ausführbar als* **'LAWOFBIG3EXAMPLE3'**

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

Sowohl das `HashSet<SimplePerson>` als auch das `TreeSet<SimplePerson>` enthalten nach dieser Korrektur nur noch einen Eintrag.

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.

Hier nochmal zur Erinnerung die steuernden Methoden:

- `HashSet<E>` – `hashCode()` und danach `equals(Object)`
- `TreeSet<E>` – `compareTo(T)` bzw. `compare(T,T)`

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

Nachdem wir bisher die konkreten Realisierungen des Interface `Collection<E>` besprochen haben, wenden wir uns nun den davon unabhängigen 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 ***Lookup-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 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>`. Allerdings bildet `Collection<E>` nicht die Basis von `Map<K,V>`, das vielmehr einen davon unabhängigen Basistyp darstellt.

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.

Zum Umgang mit dem Wert `null` als Schlüssel oder Wert beachten Sie bitte den nachfolgenden Praxistipp.

Ergänzend zu den gerade vorgestellten Methoden gibt es folgende Methoden, die Zugriff auf gespeicherte Schlüssel, Werte und Einträge bieten:

- `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 jeweiligen 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 6.16 *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<K,V>`

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 6.1.7).

Beispiel: Lookup-Map 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<>();

public static void main(final String[] args)
{
    initMapping(nameToColor);

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

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

    return nameToColor.get(colorName);
}
```

Listing 6.17 *Ausführbar als '*HASHMAPLOOKUPEXAMPLE*'*

Diese Art der Realisierung hält die Funktionalität der Abbildung in der Applikation selbst kurz, lesbar 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<K,V>`

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 einen Zwischenspeicher, auch ***Cache*** genannt, realisieren kann. Ein solcher ist immer dann nützlich, wenn man beispielsweise wiederholt auf 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 der 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 `removeEldestEntry(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 `LinkedHashMap<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<>(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);
    customers.forEach(System.out::println); // Java-8-Defaultmethode
}
```

Listing 6.18 *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 die Reihenfolge des Einfügens als Verbleibkriterium zu nutzen, ist es oftmals sinnvoller, zu betrachten, welche Elemente zuletzt verwendet wurden.[14] 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 ersetzt:

```
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.

[14] 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.

```
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");

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

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

Listing 6.19 *Ausführbar als* **'LRULINKEDHASHMAPEXAMPLE'**

Ein Start des Programms LRULINKEDHASHMAPEXAMPLE 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<K,V>`

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 einen 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 6.20 *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 Ausgaben verdeutlichen die vorangegangenen Beschreibungen: Der Aufruf von `floorKey("Ma")` liefert den Vorgängerschlüssel von "`Ma`". Weil dieser nicht existiert, wird `null` zurückgeliefert. Ein Aufruf von `higherKey("Ma")` liefert den Nachfolgerschlüssel von "`Ma`" und dies ist der Schlüssel "`Max`". Mit `lowerKey("Mz")` wird der Vorgänger von "`Mz`" ermittelt, was hier "`Moritz`" ist. Schließlich liefert `ceilingEntry("Mc")` den Nachfolger von "`Mc`".

6.1.11 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.

Stellen Sie sich als Beispiel vor, man würde Bilder basierend auf Eingaben aus einem GUI referenzieren wollen. Werden Benutzereingaben 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 (etwa komplett in Großbuchstaben) zu definieren.

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[15]
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:

- Vielfach wird – 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,[16] etwa Löschoperationen.

[15] Aggregation bedeutet, dass eine Klasse eine andere Klasse als Attribut besitzt, und Delegation meint, dass Methodenaufrufe an die aggregrierte Klasse weitergeleitet werden.

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

- 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.
- Weil kein Basisinterface aus dem Collections-Framework genutzt wird, besteht ein Handicap darin, dass eben nicht die bekannten Methoden angeboten werden oder Methodensignaturen (auch stärker) abweichen. Dies führt aber zu Inkompatibilitäten mit dem Collections-Framework.

Ableitung und Überschreiben Seit JDK 5 kann man typsichere Containerklassen eleganter durch eine Kombination von Ableitung und Einsatz von Generics realisieren und bleibt 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 zu einer `NullPointerException`. ***Damit verstößt diese Umsetzung gegen die Methodenkontrakte und verhält sich nicht korrekt wie eine Subklasse***. Damit verletzt sie das in Abschnitt 3.5.3 vorgestellte LISKOV SUBSTITUTION PRINCIPLE (LSP). In diesem Fall lässt sich das Problem dadurch lösen, dass man eine Hilfsmethode `normalizeKey(String)` erstellt:

```
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 Subklasse werden lediglich die Methoden der Oberklasse aufgerufen. ***Es wird zuvor eine Umwandlung der Schlüssel benötigt, um garantiert mit passenden Schlüsseln zu suchen***.

Verallgemeinert nutzt man statt des Typs `Image` beliebige Typen mit dem Typkürzel `V`. 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();
    }
}
```

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

6.1.12 Erweiterungen im Interface `Map` mit JDK 8

Das Interface `Map<K,V>` wurde in JDK 8 erweitert, beispielsweise in Form der Methoden `getOrDefault(K,V)`, `putIfAbsent(K,V)`. Anhand eines Beispiels wollen wir nachvollziehen, wie wir die neuen Methoden im Interface `Map<K,V>` gewinnbringend einsetzen können. Nehmen wir an, wir müssten für eine Liste von Wörtern deren Häufigkeiten bestimmen. Weil uns die dazu benötigten Testdaten in einigen Listings begleiten werden, zeige ich zunächst einmalig die Methode, die diese Werte bereitstellt:

```
private static List<String> createTestData()
{
    final List<String> wordList = Arrays.asList("Dies", "ist", "eine", "Liste",
                      "Eine", "Liste", "kann", "Worte", "enthalten",
                      "Dies", "ist", "das", "Ende", "der", "Liste");
    return wordList;
}
```

Realisierung mit JDK 7 oder früher

Als Beispiel sollen die Häufigkeiten von Wörtern in einem Text ermittelt und eine Art Histogramm erstellt werden. Zunächst zeige ich, wie man dies herkömmlich mithilfe einer `Map<K,V>` ausprogrammieren könnte:

```
final List<String> wordList = createTestData();

final Map<String, Integer> wordCounts = new TreeMap<>();
for (final String word : wordList)
{
    // Wortvorkommen hoch zählen bzw. anlegen, wenn zuvor nicht existent
    if (wordCounts.containsKey(word))
    {
        final Integer oldValue = wordCounts.get(word);
        wordCounts.put(word, oldValue + 1);
    }
    else
    {
        wordCounts.put(word, 1);
    }
}

System.out.println(wordCounts);
```

Wir sehen die Behandlung verschiedener Sonderfälle, beispielsweise wenn kein Wert vorhanden ist sowie das Auslesen des alten und Setzen des neuen Werts. Das wirkt bereits ein wenig unelegant. Insbesondere problematisch sind zwei Dinge: Erstens muss man diese Funktionalität für andere, ähnliche Anwendungsfälle immer wieder erneut ausprogrammieren, im Speziellen auch dann, wenn lediglich andere Typen für Schlüssel oder Wert genutzt werden. Zweitens ist eine derartige Verarbeitung kritisch, falls

durch andere Threads Änderungen während der Verarbeitung erfolgen, sodass im Anschluss ein anderer Zustand existiert als vor der Unterbrechung und z. B. bei der Prüfung. Natürlich kann man durch Synchronisierung für einen kritischen Bereich und die exklusive Ausführung sorgen, dies geschieht jedoch auf Kosten der Möglichkeit zur parallelen Abarbeitung. Eine weitere Alternative wären Locks.

Zusammenfassend lässt sich feststellen, dass man diese Methoden zwar relativ einfach in ihrer Funktionalität nachbauen kann, es jedoch eher schwierig ist, dies Threadsicher und bei konkurrierenden Zugriffen korrekt hinzubekommen. Umso angenehmer ist es, dass diese Methoden von der für Multithreading und konkurrierende Zugriffe ausgelegten Klasse `ConcurrentHashMap<K,V>` angeboten werden. Nachfolgend wollen wir uns vor allem um Lesbarkeit und Verständlichkeit und weniger um Multithreading kümmern. Lernen wir also einige neue Methoden im Interface `Map<K,V>` kennen.

Die Methode `getOrDefault()`

Oftmals wünscht man sich beim Zugriff auf eine Map, dass ein Defaultwert zurückgeliefert werden kann, falls kein Eintrag zu einem gewünschten Schlüssel existiert. Diese Funktionalität wird in JDK 8 durch die Methode `getOrDefault(Object,V)` realisiert. Man vermeidet dadurch ansonsten notwendige Spezialbehandlungen. Eine Methode, die analog arbeitet, könnte man wie folgt realisieren:

```
// Achtung: Nur für Singlethreading korrekt
public Object getOrDefaultSimplified(final Object key,
                                     final V defaultValue)
{
    if (!map.containsKey(key))
        return defaultValue;

    final Object value = map.get(key);
    return value;
}
```

Die Methoden `putIfAbsent()` und `replace()`

Im Interface `Map<K,V>` konnte man bis JDK 8 durch Aufrufe von `put(K,V)` Werte zu einem Schlüssel sowohl in die Map einfügen als auch einen bereits existierenden Wert überschreiben. In JDK 8 werden nun mit `putIfAbsent(K,V)` und `replace(K,V)` zwei neue, speziellere Funktionen geboten. Wie bereits am Namen zu vermuten ist, fügt `putIfAbsent(K,V)` nur dann einen Wert ein, wenn zuvor noch keiner existierte. Für `replace(K,V)` gilt es andersherum: Mit der Methode `replace(K,V)` werden lediglich schon vorhandene Einträge ersetzt. Existiert kein Wert, passiert nichts.

Beispiel Wörterzählen Die drei Methoden `getOrDefault(Object,V)`, `putIfAbsent(K,V)` und `replace(K,V)` kombinieren wir für das Wörterzählen wie folgt zwar kürzer, jedoch möglicherweise nicht intuitiv verständlich. Insbesondere muss beim ersten Hochzählen ein wenig getrickst werden:

```
public static void main(final String[] args)
{
    final List<String> wordList = createTestData();

    final Map<String, Integer> wordCounts = new TreeMap<>();
    for (final String word : wordList)
    {
        // Initialen Wert vorgeben, Achtung 0, weil später Inkrement erfolgt
        wordCounts.putIfAbsent(word, 0);
        // Wert ermitteln, wenn vorhanden
        final Integer value = wordCounts.getOrDefault(word, 0);
        // Wert ersetzen
        wordCounts.replace(word, value + 1);
    }

    System.out.println(wordCounts);
}
```

Listing 6.21 *Ausführbar als* **'WORDCOUNTPUTIFABSENTREPLACEEXAMPLE'**

Die Methoden `computeIfAbsent()` und `computeIfPresent()`

Manchmal soll nicht nur ein ganz bestimmter Wert mit `putIfAbsent(K,V)` in eine Map eingefügt werden, sondern stattdessen eine Berechnung ausgeführt werden. Das kann man unter anderem dazu nutzen, um eine sogenannte Multi Map zu realisieren, bei der für einen Schlüssel mehrere Werte gespeichert werden können. Existiert noch kein Wert für einen Schlüssel, so muss zunächst eine Collection angelegt werden. Das implementieren wir wie folgt:

```
// Achtung: Nur für Singlethreading korrekt
if (!map.containsKey(key))
{
    map.put(new ArrayList<>());
}
```

Die obige Realisierung ist nur für Singlethreading korrekt, bei Multithreading könnten mehrere Threads zeitgleich die Prüfung vornehmen und danach unterbrochen werden. Nachfolgendes Hinzufügen von Elementen wird dann möglicherweise durch frisch initialisierte `ArrayList<E>`-Instanzen wieder zunichtegemacht. Die Standardimplementierung als Defaultmethode `computeIfAbsent(K, Function<? super K, ? extends V>)` im Interface `Map<K,V>` löst dieses Problem zwar nicht, verbessert aber die Lesbarkeit, insbesondere in Kombination mit einem Lambda (vgl. Kapitel 5).

```
map.computeIfAbsent(key, it -> new ArrayList<>());
```

Beispiel Wörterzählen Für das Beispiel des Wörterzählens kann man die Verarbeitung mithilfe von Lambdas deutlich klarer wie folgt schreiben:

```
public static void main(final String[] args)
{
    final List<String> wordList = createTestData();

    final Map<String, Integer> wordCounts = new TreeMap<>();
    for (final String word : wordList)
    {
        wordCounts.computeIfPresent(word, (str, val) -> val + 1);
        wordCounts.computeIfAbsent(word, (val) -> 1);
        // Alternativ: wordCounts.putIfAbsent(word, 1);
    }

    System.out.println(wordCounts);
}
```

Listing 6.22 *Ausführbar als* **'WORDCOUNTCOMPUTEIFEXAMPLE'**

Zum Erhöhen des Zählers nutzen wir hier im Aufruf von `computeIfPresent()` einen Lambda, der als Eingabe sowohl den Wert des Schlüssels als auch des Werts erhält und Letzteren um eins erhöht zurückgibt. Falls es noch keinen Eintrag für ein Wort gibt, kann man entweder einen Aufruf von `computeIfAbsent()` oder die Methode `putIfAbsent(K,V)` nutzen.

Die Methode `merge()`

Die abschließend vorgestellte Methode `merge(K, V, BiFunction<? super V, ? super V, ? extends V>)` realisiert eine Funktionalität, ähnlich zu `computeIfAbsent()`, die einen existierenden Eintrag mit einer übergebenen Funktion verknüpft: Der neue Wert wird aus dem alten Wert und einer binären Operation ermittelt. Für den Fall, dass es den Wert noch nicht gibt, wird der an die Methode `merge()` übergebene Startwert vom Typ `V` in der Map gespeichert. Zur Verdeutlichung möchte ich dies für das Beispiel des Wörterzählens wie folgt nutzen:

```
public static void main(final String[] args)
{
    final List<String> wordList = createTestData();

    final Map<String, Integer> wordCounts = new TreeMap<>();
    for (final String word : wordList)
    {
        wordCounts.merge(word, 1, Integer::sum); // JDK 8: Methodenreferenz
    }

    System.out.println(wordCounts);
}
```

Listing 6.23 *Ausführbar als* **'WORDCOUNTMERGEEXAMPLE'**

Im Listing sehen wir die Methodenreferenz `Integer::sum` (vgl. Abschnitt 5.3), die auf die Methode `sum()` der Klasse `Integer` verweist und eine Addition wie folgender Lambda ausführt: `(int x, int y) -> x + y`.

Fazit

An der schrittweisen Weiterentwicklung des Beispiels erkennen wir sehr schön, dass man sich von der imperativen Programmierung hin zu einem deklarativen Programmierstil bewegt. Die erste Variante hat den Algorithmus mit 15 Zeilen umgesetzt. Zum Schluss benötigt man nur noch fünf Zeilen. Neben der Kürze kommuniziert die obige Lösung vor allem die gewünschte Funktionalität viel besser.

Die zuvor vorgestellten Methoden sind für viele Anwendungsfälle praktisch und erleichtern die tägliche Arbeit. Man kann sich dadurch wieder mehr auf das zu lösende Problem als auf die Details der Zugriffe auf die Map konzentrieren. Insgesamt sinkt durch den deklarativen Ansatz und durch die im Framework implementierten Algorithmen die Wahrscheinlichkeit für Flüchtigkeitsfehler – erschwerend dazu kann beim imperativen Ansatz ein kleiner Fehler viel schneller zu unerwarteten Resultaten führen.

6.1.13 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. Abbildung 6-8 bietet eine Entscheidungshilfe zur Auswahl einer geeigneten Datenstruktur aus dem Collections-Framework und greift Ideen aus dem Buch »Java 2 – Designmuster und Zertifizierungswissen« [17] von Friedrich Esser auf, beschränkt sich dabei aber aus Gründen der Übersichtlichkeit auf die zuvor besprochenen Klassen.

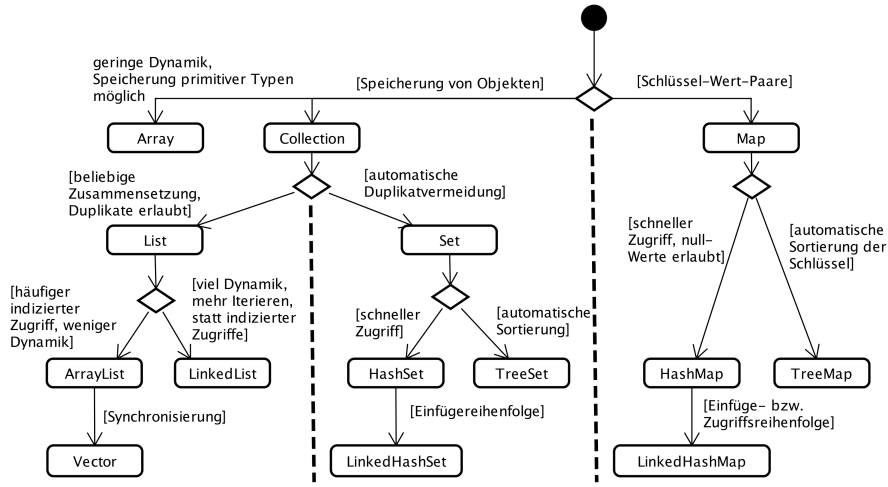


Abbildung 6-8 *Entscheidungshilfe zur Wahl von Datenstrukturen*

Als Faustregel gilt, dass die `ArrayList<E>` und 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 dienen z. B. zur Verwaltung primitiver Typen. Insbesondere bei Mehrdimensionalität stellt ein Array-Zugriff vielfach die natürlichste Zugriffsvariante dar.

Nutzt man jedoch ein Array, obwohl eigentlich Listenfunktionalität benötigt wird, dann ist dies ungünstig: Durch diese für das Problem unpassend gewählte Datenstruktur kommt es zu mehr Sourcecode und mehr Komplexität, weil die Listenfunktionalität dann jeweils an der einsetzenden Stelle vom Entwickler selbst programmiert werden muss. Dadurch steigt die Wahrscheinlichkeit für Fehler, weil Eigenimplementierungen weniger ausgereift und gut getestet sind als die Containerklassen des JDKs.

6.2 Suchen und Sortieren

Nachdem wir bisher hauptsächlich die Verwaltung 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 bis JDK 8 eine wichtige und in der Praxis häufig benötigte Funktionalität nicht enthalten: das Filtern. Das ändert sich mit Java 8. Dessen mächtige Möglichkeiten zur Filterung mit dem Filter-Map-Reduce-Framework werden in Abschnitt 7.2 beschrieben.

Zunächst betrachten wir das Suchen in Abschnitt 6.2.1. Danach behandeln die Abschnitte 6.2.2 sowie 6.2.3 das Sortieren von Arrays und Listen sowie das Sortieren mit Komparatoren. In Abschnitt 6.2.4 gehe ich dann auf Erweiterungen im Interface `Comparator<T>` mit JDK 8 ein.

6.2.1 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 durch Aufruf der Methode `contains(Object)` 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, und jedes einzelne wird basierend auf `equals(Object)` auf Gleichheit mit dem übergebenen Element bzw. den übergebenen Elementen geprüft. Für Maps existieren – wie bereits erwähnt – korrespondierende Methoden `containsKey(Object)` und `containsValue(Object)`.