Collections-Framework – Weiterführende Themen

Hash-Tabellen und Bäume

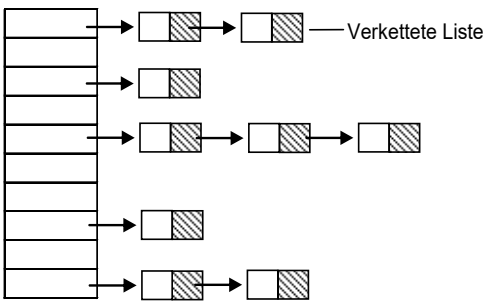
Bei größeren Datenmengen, die in Form von Arrays oder verketteten Listen gespeichert sind, ist die Suche nach einem Element, dessen Position nicht bekannt ist, oft zeitaufwendig. Die Suche wird sequenziell Element für Element durchgeführt, bis das betreffende Element gefunden wurde. Zur Lösung dieses Problems sind die beiden Datenstrukturen Hash-Tabelle und Baumstruktur entwickelt worden.

Hash-Tabellen

Eine Datenstruktur, die für schnelle Zugriffe konzipiert wurde, ist die Hash-Tabelle. Die Position, an der die Elemente gespeichert werden, wird mit einer mathematischen Formel berechnet. Diese Berechnung ist schnell und erlaubt somit auch einen schnellen Zugriff auf das Element. Dazu wird für ein Element ein Hashcode berechnet, der nur vom Zustand des Elements abhängt und nicht von anderen Objekten, wie Vorgänger oder Nachfolger. Der Nachteil von Hash-Tabellen ist, dass die Reihenfolge der Elemente nicht erhalten bleibt.

Eine Hash-Tabelle ist ein Array von verketteten Listen. Die Listen werden auch als **Bucket** bezeichnet. Die Anzahl der verketteten Listen ist von der Anzahl der zu speichernden Elemente abhängig. In eine verkettete Liste werden jeweils die Elemente mit dem gleichen Hashcode aufgenommen. Die genaue Position eines Elements berechnet sich aus dessen Hashcode modulo der Gesamtzahl der verbundenen Listen.

Eine Hash-Tabelle wird von zwei Merkmalen beeinflusst, der Kapazität und dem Ladefaktor. Die Kapazität entspricht der Anzahl der maximal zu erwartenden Elemente. Die Kapazität sollte aber um das 1,5-fache größer gewählt werden als die zu erwartende Elementanzahl. Sonst ist die Wahrscheinlichkeit zu groß, dass mehrere Elemente in einer verketteten Liste gespeichert werden und so die Effizienz abnimmt.

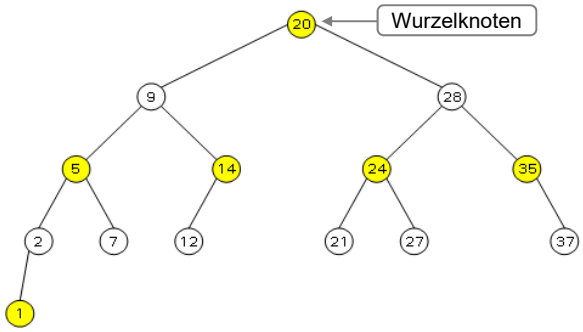


Aufbau einer Hash-Tabelle

Ist eine Hash-Tabelle voll, wird sie in eine neue Hash-Tabelle mit doppelt so vielen Buckets umgespeichert. Der Ladefaktor gibt an, bei welchem Füllstand dieses Umspeichern durchgeführt werden soll. Der Standardwert dafür beträgt in den implementierten Java-Klassen 75 %.

Baumstrukturen

Eine andere Möglichkeit, Daten schnell aufzufinden und sie dabei auch noch sortiert zu speichern, bieten Baumstrukturen. Die in der nebenstehenden Abbildung dargestellten Kreise werden als Knoten bezeichnet.
Der oberste Knoten (im Bsp. Nr. 20) ist der Wurzelknoten. Die untersten Knoten (im Bsp. 1, 7, 12 ...) sind die Blätter der Baumstruktur.
Das Beispiel zeigt einen **binären** Baum (B-Baum), d. h., jeder Knoten hat **zwei** Unterknoten bzw. jeder **Elternknoten** hat zwei **Kindknoten**.



Daten in einer Baumstruktur speichern (die Zahlen sind willkürlich gewählt)

Die Daten werden nach folgenden Regeln verteilt:

- Der Wert des jeweiligen linken Kindknotens ist kleiner als der des Elternknotens.
- Der Wert des jeweiligen rechten Kindknotens ist größer als der des Elternknotens.

Beispielsweise hat der Knoten 9 einen linken Kindknoten mit dem Wert 5 (kleiner), der rechte Kindknoten ist größer, er hat den Wert 14.

Daten suchen und Daten hinzufügen

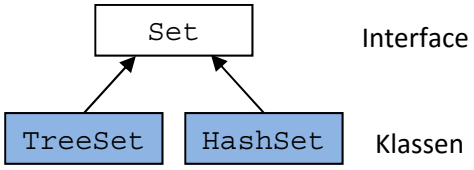
Mit nur wenigen Vergleichsoperationen lassen sich Knoten mit einem bestimmten Wert finden und neue Knoten einsortieren. Um in dem Beispiel einen Knoten mit dem Wert 8 einzufügen, sind vier Vergleiche beginnend beim Wurzelknoten erforderlich.

Vergleich		Erwartete Position
1.	8 ist kleiner als 20	Linker Unterknoten
2.	8 ist kleiner als 9	Linker Unterknoten
3.	8 ist größer als 5	Rechter Unterknoten
4.	8 ist größer als 7	Rechter Unterknoten

Sets – Collections vom Typ `Set`

Was sind Sets?

Das Interface `Set` ist vom Interface `Collection` abgeleitet. So besitzen Sets wie auch Listen die gleichnamigen geerbten Methoden, die sich aber nicht immer gleich verhalten. Weitere Methoden wurden dem Interface `Set` nicht hinzugefügt.



Klassen zur Arbeit mit Sets

Das Interface `Set` ist in der abstrakten Klasse `AbstractSet` implementiert, von der z. B. die Klassen `HashSet` und `TreeSet` abgeleitet sind.

Unterschiede zu Listen

- Sets dürfen keine doppelten Elemente enthalten. Aus diesem Grund sind Sets im mathematischen Sinne **Mengen**.
- Vor dem Anfügen eines Elements mit der Methode `add` sollten Sie prüfen, ob ein gleiches Objekt bereits vorhanden ist. Zwei Objekte sind gleich, wenn der Test `obj1.equals(obj2)` den Wert `true` ergibt. Versuchen Sie, ein gleiches Objekt anzufügen, liefert die Methode `add` den Wert `false` zurück und das Objekt wird nicht angefügt.
- Für Sets lässt sich kein ListIterator erzeugen, sondern nur ein einfacher Iterator.
- Die Objekte besitzen in Sets keine festgelegte Reihenfolge. Erzeugen Sie ein `Iterator`-Objekt aus einem `Set`, kann dessen Reihenfolge anders sein als die des eigentlichen Sets.

Für schnellen Zugriff `HashSet` erstellen

HashSets werden dann verwendet, wenn kurze Zugriffszeiten erforderlich sind.

```
HashSet()
HashSet(int initialCapacity)
HashSet(int initialCapacity, float loadFactor)
```

HashSets sind auf der Basis von Hash-Tabellen aufgebaut. In dieser Klasse stehen unter anderem die obenstehenden Konstruktoren zur Verfügung:

- Die Kapazität (`initialCapacity`) hängt von der Anzahl der zu speichernden Elemente ab. Es wird empfohlen, für diesen Wert aufgrund der damit durchgeführten Berechnungen eine Primzahl zu verwenden. Um beispielsweise 100 Elemente zu verwalten, multiplizieren Sie diese Anzahl mit 1,5 und wählen Sie dann die nächstgrößere Primzahl. Als Kapazität würden Sie den Wert 151 wählen.
- Der Ladefaktor (`loadFactor`) legt fest, bei welchem Füllstand die Tabelle umgespeichert werden soll. Der Faktor wird als Prozentzahl dezimal mit einem Wert zwischen 0,0 und 1,0 angegeben. Bedenken Sie, dass mit steigendem Füllstand die Zugriffsgeschwindigkeit abnimmt.

Es steht auch ein Standard-Konstruktor zur Verfügung, der eine Kapazität von 101 verketteten Listen und einen Ladefaktor von 75 % vorsieht.

Zum Anfügen eines Elements können Sie wie gewohnt die `add`-Methode verwenden. Um zu prüfen, ob ein Element bereits im HashSet vorhanden ist, steht die Methode `contains` zur Verfügung. Für HashSets wurde sie so angepasst, dass sie nur in der entsprechenden verketteten Liste sucht und so sehr schnell ein Ergebnis liefert.

Hashcode zur Speicherung im HashSet berechnen

Mit der Methode `hashCode` berechnen Sie den Hashcode eines Elements. Diese Methode ist in der Klasse `Object` implementiert, sodass alle Klassen diese Methode besitzen.

In eigenen Klassen müssen Sie die Methode `hashCode` überschreiben, wenn deren Objekte in einem `HashSet` gespeichert werden sollen. Auch die Methode `equals` sollte für eigene Klassen neu definiert werden, da sonst nur auf die Gleichheit der Objekte geprüft wird, nicht aber auf inhaltliche Gleichheit.

Beispiel für die Verwendung eines HashSets: *com\herdt\java9\kap13\HashSetDemo.java*

Im folgenden Beispiel wird die Klasse `HashSetDemo` verwendet, mit deren `main`-Methode Objekte einer Klasse `Person` in einem HashSet verwaltet werden.

Zuerst wird die Klasse **`Person`** vorgestellt: *com\herdt\java9\kap13\Person.java*

```
①  public class Person
   {
②    private String lastname;
     private String prename;
     private int personalNr;
     ...                              //Getter- und Setter-Methoden
③    public Person()                  //Standardkonstruktor
     {
       this("", "", 0);
     }
④    public Person(String lastname,
       String prename, int personalNr) //Konstruktor
     {
       setLastname(lastname);
       setPrename(prename);
       setPersonalNr(personalNr);
     }
⑤    public int hashCode()
     {
⑥      return getLastname().hashCode() + getPrename().hashCode()
         + getPersonalNr();
     }
⑦    public boolean equals(Object o)
     {
⑧      if ((o == null) || (o.getClass() != this.getClass()))
         return false;
       else
       {
⑨        Person obj = (Person)o;
⑩        return ((obj.getLastname().equals(getLastname()))
                 && (obj.getPrename().equals(getPrename()))
                 && (obj.getPersonalNr() == getPersonalNr()));
       }
     }
   }
```

① Die Klasse `Person` wird definiert.

② Die Klasse besitzt die Attribute `lastname`, `prename` und `personalNr`. Deren Werte werden zum Vergleich von Objekten der Klasse herangezogen.

③ Die Klasse enthält einen parameterlosen Konstruktor, der die Standardwerte für die Attribute festlegt.

④ In einem weiteren Konstruktor werden die übergebenen Parameterwerte als Werte für die Attribute des neuen Objekts verwendet.

⑤ Die Methode `hashCode` zur Ermittlung des Hashcodes aufgrund der Eigenschaften des Objekts sollte in eigenen Klassen überschrieben werden, wenn die Objekte der Klasse in Collections verwendet werden. Beachten Sie, dass innerhalb der Methode keine Überprüfung auf `null`-Werte durchgeführt wird.

⑥ Für die Generierung des Hashcodes kann beispielsweise der Hashcode von Objekten verwendet werden, die als Attribute der Klasse definiert sind. Hier werden die Hashcodes der beiden String-Objekte und die Personalnummer addiert.

⑦ Auch die Methode `equals` wird überschrieben, um Objekte auf inhaltliche Gleichheit prüfen zu können. Der Methode wird ein beliebiges Objekt als Parameter übergeben. Als Ergebnis liefert die Methode einen `boolean`-Wert zurück.

⑧ Wenn das übergebene Objekt einen `null`-Wert hat oder ein Objekt einer anderen Klasse ist, sind die Objekte nicht gleich. Die Methode gibt den Wert `false` zurück.

⑨ Um die inhaltliche Gleichheit zu prüfen, wird ein Typecast in den Typ `Person` vorgenommen.

⑩ Die Werte der Attribute beider Objekte werden verglichen. Stimmen alle überein, dann sind die Objekte inhaltlich gleich und die Methode liefert den Wert `true` zurück, sonst `false`.

Die Klasse **HashSetDemo** mit der **main**-Methode: *com\herdt\java9\kap13\HashSetDemo.java*

Im Folgenden wird der wesentliche Teil des Hauptprogramms (Klasse `HashSetDemo`) vorgestellt. In der `main`-Methode werden mehrere Objekte vom Typ `Person` erzeugt und in ein HashSet eingefügt.

```
   public static void main(String[] args)
   {
①    HashSet<Person> personSet = new HashSet<Person>();
②    Person personA = new Person("Meier", "Heinz", 33);
③    System.out.println("HashCode A: " + personA.hashCode());
     ...
     Person personE = new Person("Baumann", "Jens", 37);
     System.out.println("HashCode E: " + personE.hashCode());
④    personSet.add(personA);
     ...
     personSet.add(personE);
⑤    Person personF = new Person("Baumann", "Jens", 37);
⑥    System.out.println("Existiert F bereits? " +
       personSet.contains(personF));
⑦    if (personSet.add(personF))
       System.out.println("F angefuegt");
     else
       System.out.println("F nicht angefuegt");
     System.out.println("HashSet mit Iterator durchlaufen:");
⑧    Iterator iter = personSet.iterator();
```

```
⑨   while (iter.hasNext())
      System.out.println("HashCode: " + iter.next().hashCode());
    System.out.println
      ("HashSet mit foreach-Schleife durchlaufen:");
⑩   for (Person eachPerson : personSet)
⑪     System.out.println("HashCode: " + eachPerson.hashCode());
  }
```

① Das HashSet `personSet` wird mithilfe des Standard-Konstruktors erzeugt. Mit dem Typargument `<Person>` stellen Sie sicher, dass nur Objekte des Typs `Person` in das HashSet aufgenommen werden.

② Es wird ein Objekt der Klasse `Person` erzeugt.

③ Mithilfe der Methode `hashCode` wird der Hashcode des Objekts ausgegeben.

④ Das `Person`-Objekt wird an das HashSet `set` angehängt.

⑤ Ein neues Objekt der Klasse `Person` mit dem gleichen Inhalt wie das Objekt `personE` wird erzeugt.

⑥ Mit der Methode `contains` wird geprüft, ob dieses Objekt bereits enthalten ist. Sie erhalten den Rückgabewert `true`.

⑦ Rufen Sie die `add`-Methode auf, dann wird das Objekt nicht angefügt, da es mit gleichem Inhalt existiert. Die Prüfung auf inhaltliche Gleichheit erfolgt mit der Methode `equals` in der Klasse `Person`.

⑧ Mit der Methode `iterator` erzeugen Sie ein `Iterator`-Objekt für das HashSet `personSet`.

⑨ Das HashSet wird mithilfe des Iterators sequenziell wie eine Liste durchlaufen.

⑩ In der Schleife wird für jedes Objekt des HashSets der Hashcode ausgegeben. Wenn Sie dessen Reihenfolge mit der zuvor angezeigten Reihenfolge vergleichen, werden Sie feststellen, dass keine Übereinstimmung besteht.

⑪ Auch für HashSets kann die **foreach-Schleife** angewendet werden.

Für alle Datenstrukturen, die das Interface `Iterable` implementieren, kann alternativ die foreach-Scheife angewendet werden.

Mit `TreeSet` Baumstrukturen erstellen

Bei der Verwendung der Klasse `TreeSet` werden die Elemente innerhalb einer Baumstruktur **geordnet** gespeichert. Für die Elemente muss daher eine Sortierreihenfolge festgelegt sein. Neue Daten können hinzugefügt werden, ohne dass das TreeSet neu sortiert werden muss. Auch wenn Zugriffe über einen Baum etwas langsamer sind als bei Hash-Tabellen, sind sie doch wesentlich schneller als bei Arrays oder verketteten Listen.

Die Sortierreihenfolge ist bei bereits implementierten Klassen, wie z. B. bei der Klasse `String`, festgelegt. Zum Vergleich von zwei Objekten wird die Methode `compareTo` verwendet. Sie ist als einzige Methode im Interface `Comparable` (im Package `java.lang`) enthalten. Bei selbst definierten Klassen ist der Programmierer für die Sortierung der Objekte verantwortlich. Für eigene Klassen, deren Instanzen in einem `TreeSet` verwaltet werden sollen, müssen Sie daher das Interface `Comparable` implementieren.

- Als Übergabeparameter erwartet die Methode `compareTo` ein Objekt des entsprechenden Typs.

```
int compareTo(T o)
```

- Die Methode liefert einen Integer-Wert zurück. Ist der Wert gleich 0, sind die Objekte gleich, sonst kann der Wert entweder > 0 oder < 0 sein. Dabei ist nur das Vorzeichen entscheidend, der Wert selbst hat keine Bedeutung. Die Klasse `TreeSet` stellt weitere nützliche Methoden bereit, von denen im Folgenden einige aufgeführt sind.

`E first()`	Liefert das erste (kleinste) Element des TreeSets
`E last()`	Liefert das letzte (größte) Element des TreeSets
`SortedSet<E> headSet(E toElement)`	Gibt den Teil des TreeSets zurück, dessen Elemente kleiner sind als `toElement`
`SortedSet<E> subSet(E fromElement, E toElement)`	Gibt den Teil des TreeSets zurück, dessen Elemente größer als `fromElement` und kleiner als `toElement` sind
`SortedSet<E> tailSet(E fromElement)`	Gibt den Teil des TreeSets zurück, dessen Elemente größer oder gleich `fromElement` sind

Die von den letzten drei Methoden zurückgegebenen SortedSets werden auch als Sicht (View) auf das TreeSet bezeichnet.

Beispiel für die Verwendung eines TreeSets: *com\herdt\java9\kap13\TreeSetDemo.java*

Im folgenden Beispiel wird ein TreeSet für die Speicherung von Personen (Objekte der Klasse `Person2`) verwendet. Für das TreeSet wird ein Iterator erzeugt und damit das TreeSet durchlaufen.

Die Klasse Person2: *com\herdt\java9\kap13\Person2.java*

Die für das Beispiel verwendete Klasse `Person2` ist von der Klasse `Person` aus dem vorherigen Beispiel zu HashSets abgeleitet. Die Klasse `Person2` selbst enthält nur die Konstruktoren und die Methode `compareTo`. Alle anderen Methoden erbt sie von der Klasse `Person`.

```
public class Person2 extends Person
① implements Comparable<Person2>
{
② public Person2()
  {
    super();
  }
  public Person2
    (String lastname, String prename, int personalNr)
  {
    super(lastname, prename, personalNr);
  }
```

```
③   public int compareTo(Person2 o)
    {
④     if ((o == null) || (o.getClass() != getClass()))
        return -1;
⑤     int i = getLastname().compareTo(o.getLastname());
      if (i != 0)
⑥       return i;
      else
      {
⑦       i = getPrename().compareTo(o.getPrename());
        if (i != 0)
⑧         return i;
        else
        {
⑨         i = getPersonalNr() - o.getPersonalNr();
          return i;
        }
      }
    }
}
```

① Das Interface `Comparable` ist ein sogenanntes generisches Interface. In spitzen Klammern <> geben Sie den Argumenttyp an.

② Die beiden Konstruktoren rufen die entsprechenden Konstruktoren der Basisklasse auf.

③ Der Methode `compareTo` muss ein Objekt übergeben werden. Sie liefert einen Integer-Wert zurück.

④ Wenn das übergebene Objekt einen `null`-Wert hat oder ein Objekt einer anderen Klasse ist, sind die Objekte nicht inhaltlich vergleichbar. Die Methode gibt den Wert `-1` zurück.

⑤ Für den inhaltlichen Vergleich wird zuerst der Name herangezogen. Für die Klasse `String` ist die Methode `compareTo` bereits implementiert und kann für den Vergleich genutzt werden.

⑥ Sind beide Namen ungleich, wird ein Wert ungleich `0` zurückgegeben.

⑦ Stimmen die Namen überein, wird der Vorname zum Vergleich herangezogen.

⑧ Sind die Vornamen verschieden, wird ein Wert ungleich `0` zurückgegeben.

⑨ Sind Name und Vorname identisch, liefert der Vergleich der Personalnummern das entsprechende Ergebnis. Bei den Personalnummern handelt es sich hier um positive Zahlen. Zum Vergleich der Personalnummern kann daher beispielsweise deren Differenz einen verwendbaren `int`-Wert liefern.

Mit der Methode `compareTo` legen Sie eine Sortierreihenfolge für die Objekte der Klasse fest. Falls Sie zusätzlich noch eine andere Sortierung benötigen oder die Schnittstelle `Comparable` in der Klasse nicht implementiert ist, kann das Interface `Comparator` verwendet werden, das die nebenstehende Methode `compare` implementiert.

```
int compare(T o1, T o2)
```

Die Klasse **TreeSetDemo** mit der **main**-Methode:
com\herdt\java9\kap13\TreeSetDemo.java

```
   public static void main(String[] args)
   {
①    TreeSet<Person2> personTreeSet = new TreeSet<Person2>();
②    Person2 personA = new Person2("Meier", "Heinz", 33);
     ...                                          //weitere Objekte
definieren
     Person2 personE = new Person2("Geier", "Norbert", 35);
③    personTreeSet.add(personA);
     ...  // alle weiteren Objekte hinzufuegen
     personTreeSet.add(personE);
④    System.out.println("Ausgabe mit Positionszeiger");
⑤    Iterator iter = personTreeSet.iterator();
     while (iter.hasNext())
⑥    {
⑦      Person2 x = (Person2)iter.next();
       System.out.println(x.getLastname() + ", " + x.getPrename()
                          + " - Personal-Nr: " +
x.getPersonalNr());
     }
     System.out.println("\nAusgabe in einer foreach-Schleife");
⑧    for (Person2 x : personTreeSet)
       System.out.println(x.getLastname() + ", " + x.getPrename()
                          + " - Personal-Nr: " +
x.getPersonalNr());
   }
```

① Das `TreeSet`-Objekt `personTreeSet` zur Verwaltung von Elementen des Datentyps `Personen2` wird erzeugt.

② Fünf Objekte der Klasse `Person2` werden erzeugt.

③ Die Objekte werden in das TreeSet `personTreeSet` eingefügt. Die Methode `add` verwendet die Methode `compareTo`, um die Objekte in das TreeSet einzusortieren.

④ Für das TreeSet `personTreeSet` wird ein `Iterator`-Objekt erzeugt.

```
Ausgabe mit Positionszeiger
Geier, Norbert - Personal-Nr: 35
Geier, Norbert - Personal-Nr: 49
Meier, Heinz - Personal-Nr: 33
Schneider, Bernd - Personal-Nr: 41
Schneider, Guenther - Personal-Nr: 25

Ausgabe in einer foreach-Schleife
Geier, Norbert - Personal-Nr: 35
Geier, Norbert - Personal-Nr: 49
Meier, Heinz - Personal-Nr: 33
Schneider, Bernd - Personal-Nr: 41
Schneider, Guenther - Personal-Nr: 25
```

Die Ausgabe des Programms `TreesetDemo`

⑤ Das TreeSet wird mithilfe des Iterators sequenziell durchlaufen.

⑥ Um auf die Eigenschaften des aktuellen Elements des TreeSets zuzugreifen, wird ein Objekt der Klasse `Person2` verwendet.

⑦ Die Eigenschaften des aktuellen Objekts werden ausgegeben. Dazu werden die Getter-Methoden verwendet, die die Klasse `Person2` von der Klasse `Person` erbt.

⑧ Die Implementierung des TreeSet-Durchlaufs vereinfacht sich durch die Verwendung einer **foreach-Schleife**.

Seit der Version Java 6 lässt sich mit der Methode `descendingIterator` entsprechend der Methode `iterator` ein Positionszeiger erzeugen, mit dem der TreeSet in umgekehrter Richtung durchlaufen werden kann.

Maps – Collections vom Typ `Map<K,V>`

Was sind Maps?

Eine Map ist eine Tabelle mit Schlüssel-Wert-Paaren (`<K,V>`).

- Jeder Wert (**Value**) wird zusammen mit einem Schlüssel (**Key**) gespeichert. Das heißt, zu jedem Schlüssel gibt es genau einen oder keinen Wert.
- Über den Schlüssel kann ein Wert schnell gefunden werden.
- Wird ein neuer Wert (ein neues Objekt) in die Tabelle eingefügt, dessen Schlüssel bereits existiert, wird der vorhandene Wert mit dem neuen Wert überschrieben.

`<K,V>` bedeutet, dass Sie für eine Map entsprechend dem Datentyp der Schlüssel-Wert-Paare zwei Typargumente angeben. `K` steht für **KeyType**, `V` steht für **ValueType**.

Methoden des Interface `Map`

Das Interface `Map` gehört zum Collections-Framework, ist aber nicht vom Interface `Collection` abgeleitet. Die ersten sechs der nachfolgend aufgeführten Methoden sind jedoch mit denen des `Collection`-Interface identisch.

`void clear()`	Entfernt alle Wertepaare der Map	*
`boolean isEmpty()`	Liefert den Wert `true`, wenn die Map kein Schlüssel-Wert-Paar enthält	
`V remove(Object key)`	Entfernt das Objekt, das durch den übergebenen Schlüssel bezeichnet ist	*
`boolean equals(Object o)`	Vergleicht das übergebene Objekt mit dieser Map	
`int hashCode()`	Gibt den Hashcode für diese Map zurück	
`int size()`	Gibt die Anzahl der Wertepaare der Map zurück	
`boolean containsKey(Object key)`	Liefert den Wert `true`, wenn der übergebene Schlüssel in der Map enthalten ist	

boolean containsValue(Object value)	Liefert den Wert true, wenn die Map einen oder mehrere Schlüssel zu dem übergebenen Objekt besitzt	
Set <Map.Entry<K,V>> entrySet()	Gibt ein Set zurück, das die Schlüssel-Wert-Paare der Map in Entry-Objekten enthält. Entry-Objekte sind Objekte, die Schlüssel-Wert-Paare speichern. Über die Methoden getKey, getValue und setValue kann auf Schlüssel und Wert zugegriffen werden.	
V get(Object key)	Liefert den Wert zu dem übergebenen Schlüssel aus dieser Map	
Set<K> keySet()	Gibt ein Set zurück, das die Schlüssel der Map enthält	
V put(K key, V value)	Verbindet den übergebenen Wert mit dem übergebenen Schlüssel in dieser Map	*
Collection<V> values()	Erzeugt ein Collection-Objekt mit den Werten dieser Map	

Die mit * gekennzeichneten Methoden werden von den Klassen, die das Interface Map implementieren, zwar bereitgestellt, jedoch werden sie in einigen Klassen nicht unterstützt und lösen eine sogenannte Exception aus. In der Java-Dokumentation finden Sie die Erläuterungen, welche Methoden für die jeweilige Klasse zur Verfügung stehen und genutzt werden können.

Eine Methode, die einen Iterator erzeugt, ist nicht implementiert. Sofern die Daten nur ausgelesen werden, kann häufig eine foreach-Schleife verwendet werden, um eine Map zu durchlaufen. Wenn Sie aber einen Iterator benötigen, können Sie mit einer der folgenden Methoden ein Set bzw. eine Collection erstellen und dazu einen Iterator erzeugen.

- Erzeugen Sie mithilfe der Methode entrySet ein Set. Das erzeugte Set enthält Entry-Objekte, die im Interface Map.Entry definiert sind.
- Auch die Methode keySet gibt ein Set zurück, das für eine Iteration genutzt werden kann.
- Die Methode values liefert eine Collection. Diese lässt sich ebenfalls für die Erzeugung eines Iterators verwenden.

Das Interface Map ist in der abstrakten Klasse AbstractMap implementiert, von der beispielsweise die Klassen HashMap, TreeMap abgeleitet sind.

Mit HashMaps arbeiten

In einer `HashMap` werden die Schlüssel-Wert-Paare in Form einer Hash-Tabelle gespeichert. Der Hashcode wird aus dem Schlüssel berechnet. Tritt beim Speichern eine Kollision auf (Wertepaare mit dem gleichen Hashcode), werden diese kollidierenden Wertepaare in verketteten Listen gespeichert.

Wie auch HashSets haben HashMaps mehrere Konstruktoren, mit denen Sie die Kapazität und den Ladefaktor der erzeugten HashMap vorgegeben können. Ein parameterloser Konstruktor dient zum Erzeugen einer HashMap mit Standardwerten.

```
HashMap()
HashMap(int initialCapacity)
HashMap(int initialCapacity, float loadFactor)
```

Mit TreeMaps arbeiten

TreeMaps speichern die Schlüssel-Wert-Paare in Baumform. Für die Sortierung wird auch hier nur der Wert des Schlüssels herangezogen und nicht das zugehörige Element.

Über die Konstruktoren dieser Klasse legen Sie die Sortierreihenfolge fest.

- Der parameterlose Konstruktor erzeugt eine TreeMap, deren Schlüssel in natürlicher Reihenfolge sortiert werden.
- Dem zweiten Konstruktor wird ein `Comparator`-Objekt übergeben, in dem die Sortierreihenfolge über die Methode `compare` selbst festgelegt wird.
- Ein weiterer Konstruktor erzeugt aus dem übergebenen `Map`-Objekt ein neues `TreeMap`-Objekt, in das die Wertepaare aus der übergebenen Map in natürlicher Reihenfolge eingeordnet werden.

```
TreeMap()
TreeMap(Comparator c)
TreeMap(Map m)
```

Beispiel für die Verwendung von HashMaps und TreeMaps: *com\herdt\java9\kap13\MapDemo.java*

```
  HashMap<String, Person2> hashmap = new HashMap<String,
Person2>(11);
①  HashMap<String, Person2> hashmap =
     new HashMap<String, Person2>(11);
②  Person2 personA = new Person2("Meier", "Heinz", 33);
③  String keyA = "16233686";
   ...   //weitere Schluessel und Objekte erzeugen
④  hashmap.put(keyA, personA);
   ...  //weitere Schluessel-Wert-Paare hinzufügen
   System.out.println("\nHashMap sequentiell durchlaufen:");
```

```
⑤   for (Map.Entry<String, Person2> e : hashmap.entrySet())
    {
⑥     String key = e.getKey();
⑦     Person2 person = e.getValue();
      System.out.println("Schluessel: " + key +
        "   Name: " + person.getLastname() + ", " +
        person.getPrename() + "   Personal-Nr: " +
        person.getPersonalNr());
    }
    //Aus dem HashMap eine TreeMap erzeugen
⑧   TreeMap<String, Person2> treemap =
      new TreeMap<String, Person2>(hashmap);
    System.out.println("\nTreeMap sequentiell durchlaufen:");
⑨   for (Map.Entry<String, Person2> e : treemap.entrySet())
    {
      String key = e.getKey();
      Person2 person = e.getValue();
      System.out.println("Schluessel: " + key +
        "   Name: " + person.getLastname() + ", " +
        person.getPrename() + "   Personal-Nr: " +
        person.getPersonalNr());
    }
}
```

① Es wird eine HashMap mit einer Kapazität von 11 Schlüssel-Wert-Paaren erzeugt. Für eine HashMap können Sie zwei Typargumente angeben. Mit dem ersten Typargument legen Sie den Datentyp für den Schlüssel (`String`) und mit dem zweiten den Datentyp für den Wert (`Person2`) fest.

② Ein Objekt `personA` der Klasse `Person2` wird erzeugt.

③ Der String `keyA` ist ein Schlüssel. Er gehört zum Objekt `personA`. Dies könnte beispielsweise die Personalausweisnummer der Person sein. In den folgenden Zeilen werden weitere Schlüsselwerte und `Person2`-Objekte erzeugt und initialisiert.

④ Mit der Methode `put` werden der Schlüssel `keyA` und das Objekt `personA` als Wertepaar an die HashMap angefügt. In den folgenden Zeilen werden die weiteren Wertepaare, für die zuvor Schlüssel und Objekt erzeugt wurden, an die HashMap angefügt.

⑤ Für die sequenzielle Ausgabe der HashMap-Werte können Sie eine foreach-Schleife einsetzen. Dazu erzeugen Sie aus der HashMap `hashMap` mit der Methode `entrySet` ein Set. Die Wertepaare des Sets sind vom Typ `Map.Entry`. Die Wertepaare enthalten einen Schlüssel vom Typ `String` und als Werte Objekte vom Typ `Person2`. Diese beiden Datentypen werden als Typargument angegeben (`<String, Person2>`).

⑥ Das Interface `Map.Entry` besitzt die Methode `getKey`, die den Schlüssel liefert.

⑦ Die Methode `getValue` des Interface `Map.Entry` liefert das zugehörige Objekt.

⑧ Dem Konstruktor der Klasse `TreeMap` wird die HashMap übergeben. Dadurch wird eine TreeMap erzeugt und die Schlüssel-Wert-Paare werden in diese TreeMap in der natürlichen Ordnung (Sortierreihenfolge) einsortiert.

⑨ Die TreeMap wird wie zuvor die HashMap über eine foreach-Schleife sequenziell durchlaufen.

Die Ausgabe des Programms zeigt, dass die Wertepaare der TreeMap nach dem Schlüssel geordnet ausgegeben werden:

```
HashMap sequentiell durchlaufen:
Schluessel: 35243534   Name: Geier, Norbert   Personal-Nr: 35
Schluessel: 64376657   Name: Schneider, Guenther   Personal-Nr: 25
Schluessel: 45674576   Name: Schneider, Bernd   Personal-Nr: 41
Schluessel: 16233686   Name: Meier, Heinz   Personal-Nr: 33
Schluessel: 68832346   Name: Geier, Norbert   Personal-Nr: 49

TreeMap sequentiell durchlaufen:
Schluessel: 16233686   Name: Meier, Heinz   Personal-Nr: 33
Schluessel: 35243534   Name: Geier, Norbert   Personal-Nr: 35
Schluessel: 45674576   Name: Schneider, Bernd   Personal-Nr: 41
Schluessel: 64376657   Name: Schneider, Guenther   Personal-Nr: 25
Schluessel: 68832346   Name: Geier, Norbert   Personal-Nr: 49
```

Die Ausgabe des Programms `MapDemo`

Übung

Collections

Level		Zeit	ca. 30 min
Übungsinhalte	✔ Verwendung von HashMap ✔ Verwendung von Set		
Übungsdatei	--		
Ergebnisdateien	*Book.java, Exercise.java*		

Fortsetzung der Übungen ⑤ und ⑥ aus dem Buchkapitel:

7. Verwenden Sie für die Bücherverwaltung eine HashMap. Füllen Sie diese mit den Büchern des Arrays. Als Schlüssel soll der laufende Index verwendet werden. Geben Sie anschließend die Bücher der HashMap aus (ohne Index).

   Definieren Sie nun eine TreeMap, die die gleichen Elemente enthält wie die HashMap, und geben Sie sie ebenfalls aus.

8. In dieser Aufgabe sollen die Bücher in Collections vom Typ `Set` verwaltet werden. Arbeiten Sie zuerst mit einem HashSet. Übernehmen Sie die Bücher aus dem Array und zeigen Sie diese an. Fügen Sie dann ein neues Buch hinzu, das in der Liste bereits existiert. Erscheint dieses Buch in der Anzeige doppelt, ist das ein Fehler (ein Set kann jedes Objekt nur einmal enthalten). Lesen Sie in diesem Kapitel nach, wie Sie dieses Problem lösen können.

   Speichern Sie nun die Objekte des HashSets in einem TreeSet und zeigen Sie die Objekte des TreeSets an.