gds2 Gottlieb-Daimler-Schule 2
Technisches Schulzentrum Sindelfingen
mit Abteilung Akademie für Datenverarbeitung

Inhaltsverzeichnis

1 Begriffsbestimmung

Collections sind Datenstrukturen, die *Mengen von Daten* aufnehmen und verarbeiten können. Die Daten werden gekapselt abgelegt, und der Zugriff ist nur über vorgegebene Methoden möglich. Damit stehen viele Anwendungsmöglichkeiten und Funktionalitäten für die Speicherung und Verwaltung großer Datenmengen zur Verfügung.

Man kann Collections als eine Weiterentwicklung der Arrays betrachten.

2 "Modernere" Collections: das Collections-Framework

Bereits in Java 1 gab es Collection Klassen. Diese „traditionellen“ Collections haben einige Nachteile, z.B. bzgl. Performance und Komfort beim Zugriff.
Mit dem Jdk 1.2 wurde daher ein neues, erweitertes Konzept eingeführt, das aus über 20 Klassen besteht, die sich jedoch alle auf die drei Grundkonzepte Set, List und Map zurückführen lassen. Alle drei Konzepte sind Interfaces und fordern somit von den implementierenden Klassen die Programmierung bestimmter Methoden.

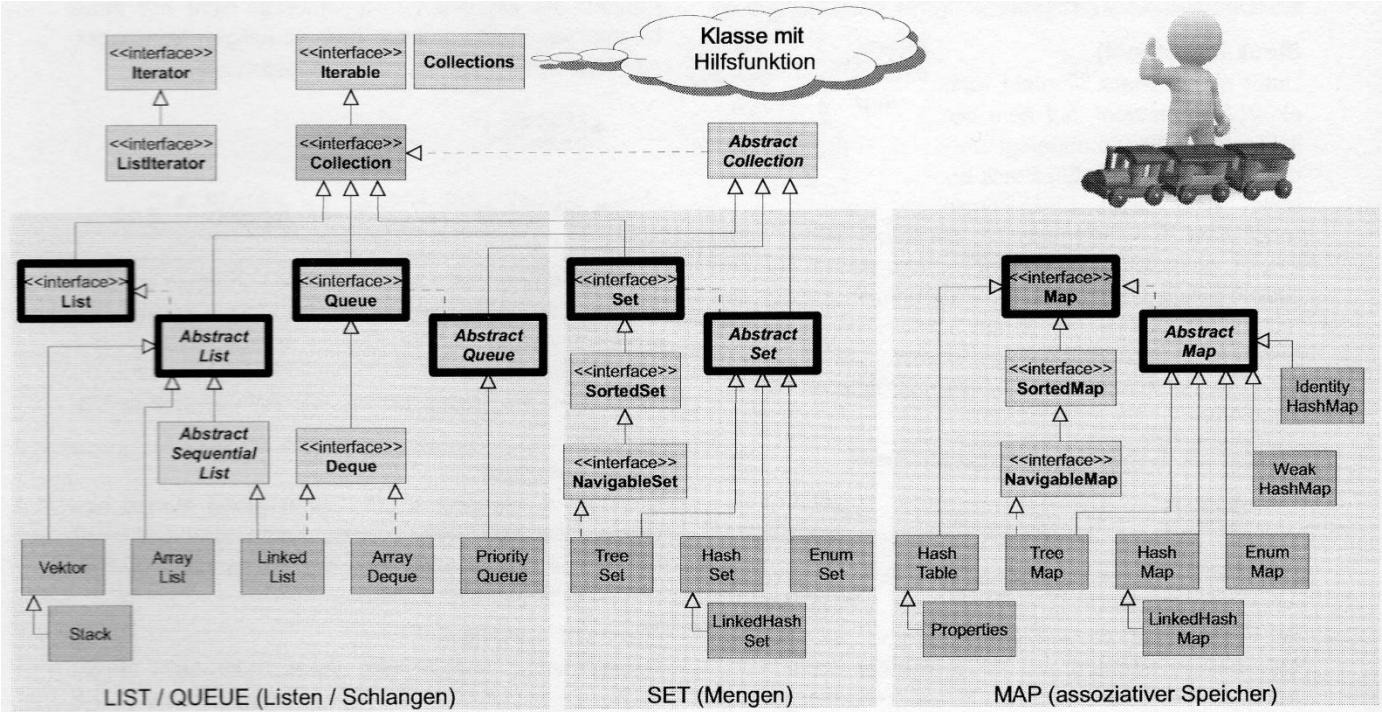

Abbildung 1 Übersicht zu Collection: Quelle Janßen, Volker : Angewandte Informatik Java, S.111

2.1 `List Interface`

Eine List Collection ist eine beliebig große Liste von Elementen beliebigen Typs, auf die sowohl wahlfrei als auch sequenziell zugegriffen werden kann.
Elemente können an beliebigen Stellen eingefügt werden.

2.2 `ArrayList`

Die Klasse `ArrayList` implementiert das `List Interface`. Sie stellt die Realisierung einer linearen Liste dar und arbeitet intern mit einem Array. Eine `ArrayList` kann beliebig viele Elemente enthalten, d.h. es wird zur Laufzeit dynamisch vergrößert. Im Vergleich zu einer verketteten Liste ist das Einfügen neuer Elemente zwar aufwendiger und damit langsamer, dafür können die Elemente aber über den Index schneller angesprochen und die Liste als Ganzes schneller durchlaufen werden.

Im Vergleich zum `Vector` (s.u.) ist eine `ArrayList` nicht „synchronisiert“, das bedeutet, dass es beim gleichzeitigen Zugriff von mehreren Programm-Threads auf diese Datenstruktur zu „Konflikten“ kommen kann. Da eine Zugriffs-Synchronisierung die Zugriffsgeschwindigkeit verschlechtert, sollte man bei Programmen ohne Multithreading die `ArrayList` bevorzugen[1].

Beispiel:

```
public static void main(String[] args)
{
    // Erzeugen einer ArrayList mit Startkapazität 100
    ArrayList eineArrayList = new ArrayList(100);
    Kunde k1 = new Kunde();
    k1.setSName("Maier");

    Kunde k2 = new Kunde();
    k2.setSName("Schulze");

    // Kunden in der ArrayList speichern
    eineArrayList.add(k1);
    eineArrayList.add(k2);
    System.out.println(eineArrayList.size());

    // Kunden aus ArrayList wieder auslesen
    // 1. Sequentieller Zugriff über Iterator
    Iterator it = eineArrayList.iterator();
    while (it.hasNext())
    {
      System.out.println(((Kunde)it.next()).getSName());
    }
    // 2. Wahlfreier Zugriff:
    System.out.println(((Kunde)eineArrayList.get(1)).getSName());
}
```

2.2.1 Zugriff auf die Elemente

Der Zugriff ist sowohl sequentiell als auch wahlfrei (über Index) möglich. Der sequentielle Zugriff geschieht über einen Iterator (s.u.), der wahlfreie Zugriff über die Methode `get()`.

Das Hinzufügen von Elementen geschieht z.B. über die Methode `add`().

[1] Die Klassen ArrayList und Vector (s.u.) sind sich sehr ähnlich. Der Hauptunterschied besteht darin, dass die Klasse Vector „synchronisiert“ ist und damit beim Zugriff aus verschiedenen Threads (s. später) sinnvoller zu verwenden ist. Allerdings kann man durch Verwendung weiterer Klassen auch auf eine ArrayList synchronisiert zugreifen.

Das Löschen von Elementen kann mit der Methode `remove()` durchgeführt werden.

Wird eine ArrayList ohne Typparameter erzeugt, so ist der Typ der Elemente die Klasse **Object**. Wie man im Beispiel sieht muss deshalb an manchen Stellen ein Type Cast **(Kunde)** verwendet werden.

2.2.2 `Iterator`

Ein Iterator dient dem sequentiellen Zugriff auf die Elemente der ArrayList. Die ArrayList selbst stellt den Iterator über die Methode `iterator`() bereit. Dieser wiederum liefert über die Methode `next`() das nächste Element in der ArrayList.

Die Klasse `Listiterator` bietet für eine `ArrayList` und einige weitere Collection Klassen zusätzlich die Möglichkeit vorwärts und rückwärts in der Liste zu navigieren.

2.3 Die Klasse `ArrayList` als „generische“ Klasse `ArrayList <E>`

Wie im vorangegangenen Kapitel gesehen, kann die Klasse `ArrayList` beliebige Objekte aufnehmen. Dies gilt auch für alle anderen Collection Klassen.
Dies ist für die Programmierung bequem, es muss allerdings beim Auslesen der Werte eine entsprechende Typkonvertierung in das gewünschte Zielobjekt vorgenommen werden.

Ein Nachteil ist dabei, dass vom Compiler nicht sichergestellt werden kann, dass in einer Collection auch tatsächlich nur Objekte eines bestimmten Typs enthalten sind, was beim Auslesen zur Laufzeit zu einer `ClassCastException` führen kann.

Um diesen Nachteil zu beheben gibt es seit J2SE5.0 die Möglichkeit, ***typsichere Collections*** zu definieren, die nur einen ganz bestimmten Datentyp zulassen.

Dazu wird die Collection mit einem zusätzlichen Typparameter versehen, der ***in spitzen Klammern*** steht:

```
ArrayList<String> eineArrayList = new ArrayList<String> ();
```

Damit ist der Compiler in der Lage, jede Einfügeoperation auf den korrekten Datentyp – in diesem Fall String - zu prüfen. Man spricht in diesem Fall von **generischen Klassen**, weil der Typ erst bei der Erzeugung der Collection festgelegt – generiert – wird.

In der API-Dokumentation werden generische Klassen z.B. so dargestellt:

Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

Dabei steht **<E>** für Elementdatentyp.

Es gibt für alle Collection Klassen die Variante mit Angabe eines Typparameters. In den meisten Fällen ist es zu bevorzugen, die generische Variante zu verwenden, da diese zur Erkennung von Fehlern schon durch den Compiler beiträgt.

Generische Klassen können auch selbst erstellt werden. Darauf soll hier aber nicht eingegangen werden.

Beispiel

```
public static void main(String[] args)
{
    // Erzeugen einer ArrayList zum Speichern von Kunde Objekten
    // mit Standardkapazität 10
    ArrayList<Kunde> eineArrayList = new ArrayList<>();
    Kunde k1 = new Kunde();
    k1.setSName("Maier");

    Kunde k2 = new Kunde();
    k2.setSName("Schulze");

    // Kunden in ArrayList speichern
    eineArrayList.add(k1);
    eineArrayList.add(k2);
    System.out.println(eineArrayList.size());

    // Kunden aus ArrayList wieder auslesen
    // 1. sequentiell über Iterator
    Iterator<Kunde> it = eineArrayList.iterator();
    while (it.hasNext())
    {
      System.out.println(it.next().getSName());
    }
    // 2. wahlfrei:
    System.out.println(eineArrayList.get(1).getSName());
}
```

2.4 `LinkedList`

Die Klasse realisiert eine doppelt verkettete lineare Liste. Mit Hilfe der Klasse `ListIterator` kann man in einer `LinkedList` vorwärts und rückwärts wandern. Da die `LinkedList` auch das `List` Interface implementiert, bietet sie die gleichen Zugriffsmöglichkeiten wie oben für die `ArrayList` beschrieben, plus zusätliche Methoden, die für verkettete Listen typisch sind (s. Oracle Doku.). Als Programmierer sollte man immer abwägen, welche Listenart für eine gegebene Aufgabenstellung eine geeignetere und performantere Lösung bietet.

2.5 `Set` Interface

Ein Set ist, im Gegensatz zu einer List, eine Menge, die keine doppelten Elemente enthält. Die Methode add() liefert ein entsprechendes Ergebnis.
Ebenso kennt sie keinen ListIterator, sondern nur einen normalen Iterator.

2.5.1 Implementierende Klassen

Das `Set`-Interface wird durch verschiedene Klassen implementiert (s. Übersicht oben). Eine dieser Klassen ist das `HashSet`:

***Beispiel**:*

```
import java.util.*;
public class HashSetDemo
{
  public static void main(String[] args)
  {
    int iAnzDoppelte = 0;
    int iZahl;
    Iterator<Integer> it;

    HashSet<Integer> einHashSet = new HashSet<>(6);
    while (einHashSet.size() < 6)
    {
      iZahl = Eingabe.getInt("Bitte einen Lottotipp eingeben:");
      if (einHashSet.add(iZahl) == false)
      {
        iAnzDoppelte++;
      }
    }
    it = einHashSet.iterator();
    while (it.hasNext() == true)
    {
      System.out.println("Tipp: "+ it.next());
    }
    System.out.println(iAnzDoppelte + " Doppelte");
  }
}
```

2.6 `Map<K,V>` Interface

Das `Map` Interface wird von allen Schlüssel-Wert-Paar Collection Klassen implementiert (s. Übersicht oben). Ein Map-Collection kann keine doppelten Schlüsselwert enthalten. Ein Schlüssel verweist auf maximal einen Wert, wobei es sich bei einem Wert um ein Objekt handelt.

2.7 Die Klasse `HashMap<K,V>`[2]

2.7.1 Beschreibung

In der Klasse `HashMap` werden Begriffe auf Schlüssel abgebildet. Über den Schlüssel ist ein einfacherer Zugriff möglich. `HashMap` benutzt eine Hash-Funktion, um Schlüssel auf Indexpositionen abzubilden. Da die Verteilung gleichmäßig auf den verfügbaren Bereich erfolgt, ist eine Reihenfolge nicht definiert. Aus diesem Grund gibt es für eine `HashMap` auch keine Iterator.

2.7.2 Zugriff auf die Elemente

Einfügen

```
public V put(K key, V value);
```

Verknüpft den angegebenen `value` mit dem gegebenen `key`. Falls die Map bereits eine Zuordnung für den `key` enthält, wird der bisherige Wert ersetzt. Zurückgegeben wird der bisherige `value` oder `null`, falls noch keine Zuordnung für den `key` bestand.

Auslesen

```
public V get(K key);
```

Gibt den zum gegebenen `key` zugeordneten `value` zurück oder `null`, falls für den `key` keine Zuordnung gespeichert ist.

Prüfen:

```
public boolean containsValue(V value);
public boolean containsKey(K key);
```

Zugriff über einen keySet View:

Eine `HashMap` definiert keine Reihenfolge der Einträge. Man kann aber die Menge der enthaltenen `keys` abrufen und bekommt diese in einem Set-View geliefert. Der Set-View spiegelt alle Änderungen, die danach an der `HashMap` ausgeführt wurden wider (s. Bsp. unten).

```
public Set<K> keySet();
```

Man kann auch die Menge aller Werte abrufen und bekommt diese als Collection-View geliefert.

```
public Collection<V> values()
```

[2] Die Klassen HashMap und Hashtable (s.u.) sind sich sehr ähnlich. Der Hauptunterschied besteht darin, dass die Klasse Hashtable „synchronisiert“ ist und damit beim Zugriff aus verschiedenen Threads (s. später) sinnvoller zu verwenden ist. Allerdings kann man durch Verwendung weiterer Klassen auch auf eine HashMap synchronisiert zugreifen.

Beispiel

```
public static void main(String[] args)
{
  HashMap<String, String> eineHashMap = new HashMap<>();
  eineHashMap.put("Peter", "0711 - 12345");
  eineHashMap.put("Maria", "07031 - 65432");
  // Zugriff wahlfrei
  System.out.println("Marias Nummer " + eineHashMap.get("Maria"));
  System.out.println("Peters Nummer " + eineHashMap.get("Peter"));

  // Zugriff über alle Schlüssel
  Set<String> keySet = eineHashMap.keySet();
  for (String key : keySet)
  {
    System.out.printf("%-10s %s\n", key, eineHashMap.get(key));
  }
  eineHashMap.remove("Peter");
  System.out.println();
  for (String key : keySet)
  {
    System.out.printf("%-10s %s\n", key, eineHashMap.get(key));
  }

}
```

Ausgabe:

```
Marias Nummer 07031 - 65432
Peters Nummer 0711 - 12345
Peter      0711 - 12345
Maria      07031 - 65432

Maria      07031 - 65432
```

Anfangskapazität (initialCapacity) und Füllfaktor (loadFactor)

Wie aus Informatik Grundlagen bekannt, sollte eine HashMap aus Performanzgründen immer nur zu einem gewissen Grad gefüllt sein. Ein Füllgrad von 75% gilt als guter Kompromiss zwischen Speicherplatzverbrauch und Performanz.

Die Klasse HashMap bietet einen Konstruktor bei dem man die Anfangskapzität und den Füllfaktor festlegen kann. Übersteigt die Anzahl der Einträge in der Map den angegebenen Füllgrad (Füllfaktor in %), dann wir die HashMap automatisch vergrößert und neu aufgebaut. Dies erfordert selbstverständlich Rechenzeit und sollte durch eine Wahl einer entsprechenden Anfangskapazität optimiert werden. (Siehe Oracle Doku.)

3 "Traditionelle" Collections

Seit dem Jdk 1.0 gibt es die "traditionellen" Collection-Klassen `Vector`, `Stack`, `Dictionary` (abstrakte Klasse, obsolete – ersetzt durch `Map interface`), `Hashtable` und `BitSet`.

3.1 Die Klasse `Vector`

3.1.1 Beschreibung

Die Klasse `Vector` stellt die Implementierung einer linearen Liste dar und arbeitet intern mit einem Array. Ein `Vector` kann beliebige Elemente enthalten, d.h. er wird zur Laufzeit dynamisch vergrößert. Im Vergleich zu einer verketteten Liste ist das Einfügen neuer Elemente zwar aufwendiger und damit langsamer, dafür können die Elemente aber über den Index schneller angesprochen und die Liste als Ganzes schneller durchlaufen werden.

Im Vergleich zur `ArrayList` (s.u.) ist ein `Vector` „synchronisiert“, das bedeutet, dass diese Datenstruktur von mehreren Programm-Threads gleichzeitig verwendet werden kann. Da die Synchronisierung allerdings die Zugriffsgeschwindigkeit verschlechtert, sollte man bei Programmen ohne Multithreading die `ArrayList` bevorzugen.

3.1.2 Zugriff auf die Elemente

Der Zugriff ist sowohl sequentiell als auch wahlfrei (über Index) möglich. Der sequentielle Zugriff geschieht über einen Iterator, der wahlfreie Zugriff über die Methoden `firstElement()`, `lastElement()` oder `get()`[3].

Das Hinzufügen von Elementen geschieht z.B. über die Methode `add`().

Das Löschen von Elementen kann mit der Methode `remove()` durchgeführt werden.

3.1.3 Iterator

Ein Iterator dient dem sequentiellen Zugriff auf die Elemente des Vectors. Der Vector selbst stellt den Iterator über die Methode `iterator`() bereit. Dieser wiederum liefert über die Methode `next`() das nächste Element im Vector.

[3] Da die Klasse `vector` zu den „traditionellen“ Collections von JDK 1.0 gehört, besitzt die Klasse aus Gründen der Rückwärtskompatibilität für manche Operationen mehrere Methoden mit identischer Funktionalität. Die hier genannten Methoden entsprechen denen, die im `List interface` definiert sind und somit für alle Listen Klassen existieren.

Beispiel

```
public class VectorDemo
{
  public static void main(String[] args)
  {
    Vector einVector = new Vector();
    Kunde k1 = new Kunde();
    k1.setName("Maier");
    Kunde k2 = new Kunde();
    k2.setName("Schulze");
    // Kunden im Vector speichern
    einVector.add(k1);
    einVector.add(k2);
    // Kunden aus Vector wieder auslesen
    // 1. sequentiell über Iterator
    Iterator it = einVector.iterator();
    while (it.hasNext())
    {
      System.out.println(((Kunde)it.next()).getName()); // Typecast
    }
    // 2. wahlfrei:
    System.out.println(((Kunde)einVector.get(1)).getName());
  }
}
```

3.2 Die Klasse `Vector` als „generische“ Klasse `Vector<E>`

Wie im vorangegangenen Kapitel gesehen, kann die Klasse `Vector` beliebige Objekte aufnehmen. Dies gilt auch für alle anderen Collection Klassen.
Dies ist für die Programmierung bequem, es muss allerdings beim Auslesen der Werte eine entsprechende Typkonvertierung in das gewünschte Zielobjekt vorgenommen werden.

Ein Nachteil ist dabei, dass vom Compiler nicht sichergestellt werden kann, dass in einer Collection auch tatsächlich nur Objekte eines bestimmten Typs enthalten sind, was beim Auslesen zur Laufzeit zu einer `ClassCastException` führen kann.

Um diesen Nachteil zu beheben gibt es seit J2SE5.0 die Möglichkeit, ***typsichere Collections*** zu definieren, die nur einen ganz bestimmten Datentyp zulassen.

Dazu wird die Collection mit einem zusätzlichen Typparameter versehen, der ***in spitzen Klammern*** steht:

```
Vector<String> einVector = new Vector<String> ();
```

Damit ist der Compiler in der Lage, jede Einfügeoperation auf den korrekten Datentyp – in diesem Fall String - zu prüfen. Man spricht in diesem Fall von generischen Klassen, weil der Typ erst bei der Erzeugung eines Objekts festgelegt – generiert – wird.

In der API-Dokumentation werden generische Klassen z.B. so dargestellt:

Dabei steht **<E>** für Elementdatentyp.

Es gibt für alle Collection Klassen die Variante mit Angabe eines Typparameters. In den meisten Fällen ist es zu bevorzugen, die generische Variante zu verwenden, da diese zur Erkennung von Fehlern schon durch den Compiler beiträgt.

Generische Klassen können auch selbst erstellt werden. Darauf soll hier aber nicht eingegangen werden.

Beispiel

```
public class VectorDemo
{
  public static void main(String[] args)
  {
    Vector<Kunde> einVector = new Vector<>();

    Kunde k1 = new Kunde();
    k1.setName("Maier");

    Kunde k2 = new Kunde();
    k2.setName("Schulze");

    // Kunden im Vector speichern
    einVector.add(k1);
    einVector.add(k2);
    // Kunden aus Vector wieder auslesen
    // 1. sequentiell über Iterator
    // der Iterator muss auch parametriert werden
    Iterator<Kunde> it = einVector.iterator();
    while (it.hasNext())
    {
      System.out.println(it.next().getName()); // ohne Typcast
    }
    // 2. wahlfrei:
    System.out.println(einVector.get(1).getName());
  }
}
```

3.3 Die Klasse `Stack`

3.3.1 Beschreibung

Die Klasse `Stack` realisiert die Datenstruktur des Stack (Stapel), der nach dem LIFO-Prinzip[4] arbeitet. Im Prinzip ist die Klasse `Stack` eine Erweiterung des Vectors, er kennt also die gleichen Methoden.

3.3.2 Zugriff auf die Elemente

Einfügen

Das Einfügen erfolgt wie beim Stack üblich, am oberen Ende des Stacks:

```
public Object push(Object einEintrag)
```

Auslesen

Das Auslesen erfolgt wie beim Stack üblich, ebenfalls vom oberen Ende des Stacks:

```
public Object pop()   // Oberstes Element entnehmen
public Object peek()  // Oberstes Element lesen
```

Beispiel

```
import java.util.*;
public class StackDemo
{
  public static void main(String[] args)
  {
    Stack<Kunde> einStack = new Stack<>();
    Kunde k1 = new Kunde();
    k1.setName("Maier");
    Kunde k2 = new Kunde();
    k2.setName("Schulze");
    // Kunden im Stack speichern
    einStack.push(k1);
    einStack.push(k2);
    // Kunden aus dem Stack auslesen
    System.out.println(einStack.pop().getName()); //Schulze
    System.out.println(einStack.pop().getName()); //Maier
  }
}
```

[4] LIFO Last In First Out: Das Objekt, das als letztes mit `push()` im Stack gespeichert wurde, wird mit dem nächsten `pop()` wieder ausgelesen und entnommen. Das Verhalten ist also wie bei einem Stapel, bei dem man immer nur ganz oben etwas drauflegen kann und auch nur von oben wieder lesen oder entnehmen kann.

3.4 Die Klasse `Hashtable`

3.4.1 Beschreibung

In der Klasse `Hashtable` werden Begriffe auf Schlüssel abgebildet. Über den Schlüssel ist ein einfacherer Zugriff möglich. Hashtable benutzt eine Hash-Funktion, um Schlüssel auf Indexpositionen abzubilden. Da die Verteilung gleichmäßig auf den verfügbaren Bereich erfolgt ist eine Reihenfolge nicht definiert. Index-Kollisionen werden intern mit verketteten Listen gelöst.

3.4.2 Zugriff auf die Elemente

Einfügen

```
public Object put (Object schlüssel, Object wert);
```

Auslesen

```
public Object get (Object schlüssel);
```

Prüfen:

```
public boolean contains (Object wert);
public boolean containsKey (Object schlüssel);
```

Zugriff über Iterator:

Die Hashtable kann auch sequentiell über einen Iterator (=Enumeration) durchlaufen werden, entweder über die Schlüssel,

```
public Enumeration keys();
```

oder über die Werte:

```
public Enumeration elements();
```

Beispiel

```
public class HashtableDemo
{
  public static void main(String[] args)
  {
    Hashtable<String,String> eineHashtable = new Hashtable<>();
    eineHashtable.put("Peter", "0711 - 12345");
    eineHashtable.put("Maria", "07031 - 65432");
    // Zugriff wahlfrei
    System.out.println("Peter hat die Nummer"
                       + eineHashtable.get("Peter"));

    // Zugriff sequentiell
    Iterator<String> it = (Iterator<String>)eineHashtable.keys();
    while (it.hasNext())
    {
      System.out.println(it.next());  // Schlüssel
    }
    eineHashtable.remove("Peter");
  }
}
```