HANSER

Leseprobe

Albert Zimmermann

Basismodelle der Geoinformatik

Strukturen, Algorithmen und Programmierbeispiele in Java

ISBN: 978-3-446-42091-5

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-42091-5

sowie im Buchhandel.

2

2 Grundlagen

Für die Darstellung der Zusammenhänge, die in diesem Buch behandelt werden, werden in diesem Kapitel wichtige Grundlagen vorgestellt. Zu diesen Themen gehören der objektorientierte Programmieransatz, Grundzüge der Vektorgeometrie, der Graphentheorie und der relationalen Algebra als theoretische Grundlage von Datenbanksystemen.

2.1 Der objektorientierte Ansatz

In der modernen Softwareentwicklung hat sich der *objektorientierte Ansatz* (*OO-Ansatz*) durchgesetzt. Er liegt nicht nur der Programmiersprache Java, sondern auch den modernen Modellansätzen der Geoinformatik zugrunde. Im folgenden Abschnitt werden die Grundlagen der objektorientierten Softwareentwicklung zusammen mit ihren relevanten Komponenten vorgestellt. Eine vertiefte Behandlung des objektorientierten Ansatzes der Sprache *UML* und ihrer Umsetzung in Java-Quellcode findet man in [BALZ05a] und [BALZ05b].

2.1.1 Das objektorientierte Softwaremodell

Das *objektorientierte Softwaremodell* geht davon aus, dass sich Programme oder auch einzelne Programmmodule wie gewöhnliche Objekte unserer realen Welt verhalten. Diese Sichtweise liegt dem menschlichen Erfahrungsschatz und Verständnis näher als frühere Programmieransätze. Der Erfolg der objektorientierten Softwareentwicklung ist hauptsächlich darauf zurückzuführen, dass neue Softwarekomponenten weitgehend aus bereits bestehenden Softwaremodulen abgeleitet werden können. Damit wird es möglich, komplexe Programmsysteme effizient zu entwickeln.

Objekte, Attribute und Methoden

Im Mittelpunkt des objektorientierten Ansatzes stehen *Objekte*. Dies können reale Gegenstände, Personen oder abstrakte Sachverhalte sein. Jedes Objekt lässt sich durch seine Eigenschaften und seine Verhaltensweisen beschreiben.

Auch in der Datenverarbeitung wird der Begriff Objekt verwendet. Darunter versteht man Daten (d.h. Eigenschaften), die logisch zusammengehören. Das Objekt bildet damit eine Einheit, in der sich ihre Daten, auch *Attribute* genannt, wie in einem Container befinden. In der Geoinformatik sind ebenfalls Datenobjekte bekannt. Sie werden dort als *Features* bezeichnet [OGC126]. **Abbildung 2.1** zeigt ein Objekt vom Typ Haus mit den Attributen Wohnfläche, Geschosse, Eigentümer und Straße, die für ein spezielles Anwendungsprogramm von Bedeutung sein können.

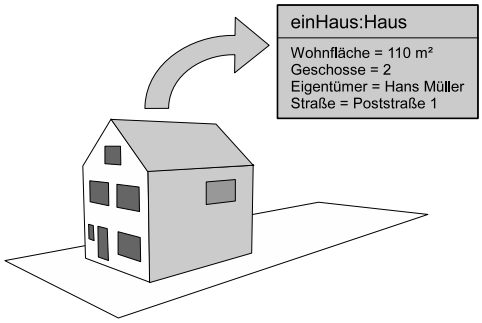

Abbildung 2.1: Ein Realweltobjekt vom Typ Haus und seine Eigenschaften.

Der OO-Ansatz geht aber über die Behandlung passiver Datencontainer hinaus: Objekte im Sinne der objektorientierten Programmierung verfügen nicht nur über Eigenschaften, sondern auch über ein vorgegebenes Verhalten, wodurch sie mit anderen Objekten kommunizieren und sogar interagieren können. In objektorientierten Programmen ist das Verhalten von Objekten durch *Methoden* bzw. *Operationen* festgelegt.

Klassen

Objekte, die die gleichen Eigenschaften und Verhaltensweisen besitzen, gehören zum gleichen *Objekttyp* (oder *Objektart*). Der objektorientierte Softwareansatz verwendet dafür den Begriff *Klasse* (*class*). Klassen werden bei objektorientierten Programmiersprachen in sogenannten Klassendeklarationen zusammen mit ihren Eigenschaften und Verhaltensweisen festgelegt. Sie wirken wie Schablonen, die den Objekten einer Klasse die vorgesehenen Eigenschaften und Verhaltensweisen aufprägen. Umgekehrt bedeutet dies, dass andere als in der Klassendeklaration festgelegte Eigenschaften und Verhaltensweisen nicht auf die Objekte anwendbar sind. Dies entspricht der Erfahrung aus der realen Welt. Auch dort besitzt jedes Objekt nur seine typischen Eigenschaften und Verhaltensweisen. So ist es selbstverständlich, dass ein Mensch nicht aus eigener Kraft fliegen und ein Hund nicht sprechen kann.

Listing 2.1 zeigt die Java-Klassendeklaration für den Objekttyp Haus aus **Abbildung 2.1**. Sie stellt eine in sich abgeschlossene Programmeinheit dar und trägt den gleichen Dateibezeichner wie die Klassendeklaration, in diesem Fall ist dies `Haus.java`. Die Erläuterungen sind als Kommentare eingefügt.

Listing 2.1: Quellcode der Deklaration für die Klasse Haus.

```
// Beginn der Klassendeklaration
public class Haus {

     // Attribute
     int wohnflaeche;
     int geschosse;
     String eigentuemer;
     String strasse;

     // Methode zur Ausgabe der Daten Strasse und Eigentuemer
     public String getDaten() {
            return "Haus (" +
                   "Adresse: " + strasse +
                   "Eigentuemer: " + eigentuemer +")";
     }

}    // Ende der Klassendeklaration
```

Mit der oben angegebenen Klassendeklaration lassen sich Objekte vom Typ Haus in anderen Anwendungsprogrammen verwenden. **Listing 2.2** gibt ein Beispiel für die Verwendung der Klasse `Haus` in einem anderen Programm an. Es legt ein neues Objekt vom Typ Haus an und gibt anschließend den Eigentümer und die Adresse in einem Textfenster aus.

Listing 2.2: Quellcode zur Verwendung der Klasse Haus.

```
public class HausDemo {

     // Hauptmethode für den Programmaufruf
     public static void main(String[] args) {

            // Ein Objekt vom Typ Haus anlegen
            Haus einHaus = new Haus();

            // Eigenschaften zuweisen
            einHaus.wohnflaeche = 110;
            einHaus.geschosse = 2;
            einHaus.eigentuemer = "Hans Müller";
            einHaus.strasse = "Poststrasse 1";

            // Aufruf der Methode getDaten(), Ausgabe auf die Konsole
            System.out.println(einHaus.getDaten());

     } // Ende der Hauptmethode

}
```

Vererbung

Objektorientierte Programmiersprachen verdanken ihren Erfolg hauptsächlich der *Vererbung*. Darunter wird die Möglichkeit verstanden, neue Klassen aus bereits bestehenden Deklarationen zu entwickeln und dadurch funktionsfähigen Programmcode in neue Softwareprojekte zu integrieren. Eine Klasse, die als Vorlage für die Vererbung dient, nennt man eine *Elternklasse* oder *Basisklasse* (*Superclass*). Eine daraus weiter entwickelte Klasse wird als *Kindklasse* oder als *abgeleitete Klasse* (*Subclass*) bezeichnet. Für die Vererbung gibt es festgelegte Regeln:

- Die Attribute der Elternklasse werden unverändert in die Kindklasse übernommen. Sie müssen bei der Deklaration der Kindklasse nicht erneut angegeben werden.
- Methoden der Elternklasse können unverändert in die Kindklasse übernommen oder in der Kindklasse verändert werden. Erscheint eine Methode der Elternklasse nicht erneut in der Deklaration der Kindklasse, so ist sie in der Kindklasse unverändert verfügbar.
- Taucht eine Methode der Elternklasse erneut in der Kindklasse auf, so ist sie dadurch verändert bzw. überschrieben.
- In der Kindklasse können neue Attribute und Methoden hinzugefügt werden.

Der Quellcode von **Listing 2.3** zeigt die Klasse `Postagentur`, die die bestehende Klasse `Haus` als Elternklasse einbindet. Als ergänzendes Attribut ist die Agenturnummer vorgesehen. Für die Datenausgabe soll die Elternmethode `getDaten()` um die Filialnummer ergänzt werden. Sie ist daher durch eine veränderte Methode überschrieben, die intern die Methode der Elternklasse (Schlüsselwort `super`) aufruft.

Listing 2.3: Quellcode der Deklaration für die Klasse Postagentur.

```
public class Postagentur extends Haus {

     // Attribute
     int nummer;           // Filialnummer

     // Methode getDaten()
     public String getDaten() {
            return super.getDaten() + " " +
                   "Nummer: " + nummer }

}    // Ende der Klassendeklaration
```

2.1.2 UML-Klassenmodelle

Bereits im **Kapitel 1** wurde auf die Bedeutung der Modellierungssprache *UML* (*Unified Modeling Language*) für die Softwareentwicklung hingewiesen. Eine zentrale Rolle besitzen dabei die *UML-Klassenmodelle*. Sie begleiten den Entwicklungsprozess über die Phasen Analyse und Entwurf hinweg und dienen in der anschließenden Implementierungsphase als Vorlagen für die Generierung von Programmquellcode.

Klassen

Parallel zur textuellen Syntax verfügen *UML-Modelle* über graphische Symbole, womit sie Sachverhalte oder Abläufe in Diagrammen verdeutlichen können. *UML-Klassenmodelle* werden in *Klassendiagrammen* dargestellt und sind in zahlreichen Dokumenten wie z.B. in den *ISO-Normen* und *OGC-Standards* zu finden. Die graphischen Notationen, d.h. die Symbole für die relevanten Elemente eines UML-Klassenmodells, sind in **Abbildung 2.2** beispielhaft abgebildet.

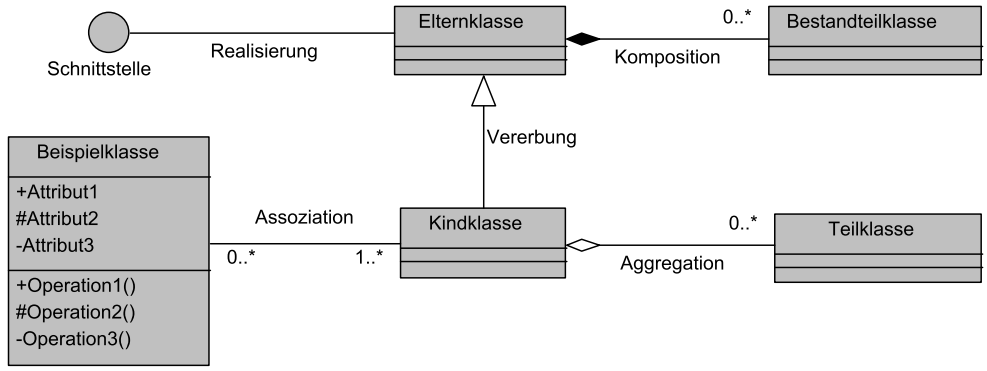

Abbildung 2.2: Notationen typischer Elemente eines UML-Klassendiagramms.

Klassen sind die zentralen Elemente eines Klassendiagramms und kommen in verschiedenen Varianten vor. Sie werden als Rechtecke dargestellt, die meist vertikal in drei Abschnitte eingeteilt sind.

- Der *obere Abschnitt* enthält den *Klassennamen* zusammen mit weiteren Eigenschaften, die als textuelle Ergänzung beigefügt werden können. In UML stehen für Klassen bereits vordefinierte Schlüsselbegriffe wie `<<abstract>>` für *abstrakte Klassen* oder `<<interface>>` für *Schnittstellen* (obwohl es auch eine graphische Alternativdarstellung gibt, siehe **Abbildung 2.2**) zur Verfügung. Darüber hinaus kann der Modellentwickler auch eigene Schlüsselwörter festlegen. So taucht in verschiedenen OGC-Dokumenten der Begriff `<<Feature>>` auf.
- Im *mittleren Abschnitt* werden alle *Attribute* einer Klasse aufgeführt. In der Analysephase reicht die Angabe des Attributnamens mit einer kurzen Beschreibung aus. In der Entwurfsphase, wenn die technischen Details des Softwaresystems feststehen, werden Attribute um weitere Angaben, wie z.B. den Datentyp, ergänzt.
- Im *unteren Abschnitt* erscheint schließlich die Liste der *Methoden*. In der Analysephase werden auch dort nur die Methodennamen aufgeführt, die um einen leeren Klammerausdruck „`()`“ ergänzt sind. Erst in der Entwurfsphase erhalten die Methoden den Rückgabedatentyp sowie die Namen und Datentypen der Übergabeparameter.

Sowohl bei Attributen als auch bei Methoden können weitere graphische Symbole auftreten, deren Bedeutung im Detail in [BALZ05a] beschrieben ist.

Jede Klasse wird nur einmal im UML-Modell gespeichert. Sie darf allerdings in verschiedenen Klassendiagrammen dargestellt werden. In der Praxis gibt es daher zu jedem etwas komplexeren Modell gleich mehrere Klassendiagramme, die unterschiedliche Aspekte oder Details des Softwaresystems beleuchten. Um umfangreiche Klassendiagramme übersichtlich zu gestalten, dürfen die Klassen auch in komprimierter Form dargestellt werden. In diesem Fall wird auf die Angabe der Attribute und Methoden verzichtet.

Assoziationen

UML-Klassenmodelle bilden die Beziehungen, die zwischen zumeist unterschiedlichen Objekten bestehen, als *Assoziationen* ab. Das Konzept der *Beziehungen* wurde von der Modellierung *relationaler Datenbanken* übernommen und in UML verfeinert. In Klassendiagrammen sind Assoziationen als Verbindungslinien zwischen den beteiligten Klassen abgebildet. In der Regel stellt eine Assoziation die Bindung zwischen zwei Klassen her, die durch weitere Angaben beschrieben wird:

- Die inhaltliche Bedeutung einer Beziehung ist häufig durch ein Verb angegeben. So kann die Lagebeziehung zwischen einem Haus und seinem Grundstück mit „liegt auf“ oder „gehört zu“ beschrieben werden. Die Angabe der Bedeutung befindet sich generell in der Mitte der Verbindungslinie.
- An der Verknüpfungsstelle zwischen dem Klassensymbol und der Verbindungslinie ist häufig die *Kardinalität* angegeben. Dies ist eine numerische Beschreibung, die die Anzahl der Objekte festlegt, die in Beziehung mit einem Objekt der assoziierten Klasse stehen können. So können zum Beispiel kein, ein oder mehrere Häuser auf einem Grundstück errichtet sein („Grundstück enthält 0...*n* Häuser“), umgekehrt ist aber jedem Haus nur ein Grundstück zuzuordnen („Haus liegt auf 1 Grundstück“). Anstelle von „0...*n*“ sind in UML-Klassendiagrammen auch die Angaben „0..*“ oder „*“ zu finden.

Spezielle Assoziationen

Anderes als das klassische relationale Datenbankmodell kennt UML weitere Varianten von Assoziationen, die in den nachfolgenden Projektphasen wichtige Entscheidungshilfen für die Realisierung des Softwaresystems bieten:

- Die *Vererbungshierarchie* als zentrales Instrument des objektorientierten Ansatzes ist stets durch ein leeres Dreieck gekennzeichnet, das sich an der Verknüpfungsstelle zwischen der Elternklasse und der Verbindungslinie zur Kindklasse befindet.
- *Aggregationen* und *Kompositionen* beschreiben besondere Abhängigkeiten zwischen Objekten von zwei Klassen, die qualitativ durch die Verben „gehört zu“ oder „enthält“ charakterisiert werden können. Dabei verkörpert das Objekt der Aggregatklasse ein zusammengesetztes Gebilde, das die Objekte der anderen Klasse gewissermaßen als Bestandteile enthält. Bei einer *Komposition* handelt es sich um eine noch stärkere Bindung zwischen den Objekten der Aggregat- und der Elementklasse. Aggregationen und Kompositionen sind an einer Raute zu erkennen, die sich im Verknüpfungspunkt zwischen der Aggregatklasse und der Verbindungslinie zur Elementklasse befindet. Der Unterschied zwischen einer Aggregation und einer Komposition ist zumeist erst in der Implementierungsphase von Bedeutung.

Beispiel: Streckenzug

Die geometrische Figur eines Streckenzugs ist durch ihre Stützpunkte und durch einfache geradlinige Streckensegmente festgelegt, die die Stützpunkte in einer vorgegebenen Reihenfolge miteinander verbinden. Zwischen einem Streckenzug und den Punkten besteht

somit eine Aggregation, denn die Stützpunkte sind im Streckenzug gewissermaßen als Bestandteile enthalten. Bleiben die Stützpunkte beim Löschen des Streckenzugs erhalten, können sie für einen neuen Streckenzug weiter verwendet werden. In diesem Fall spricht man von einer *Aggregation*. Eine *Komposition* liegt dann vor, wenn beim Löschen des Streckenzugs auch die darin enthaltenen Stützpunkte untergehen und danach nicht mehr verfügbar sind.

2.1.3 Java

Der Inhalt dieses Abschnitts beschränkt sich darauf, verschiedene Aspekte der Programmiersprache Java herauszugreifen, die für die Implementierung der UML-Modelle von Bedeutung sind. Dabei wird vorausgesetzt, dass der Leser bereits über grundlegende Kenntnisse zur Programmiersprache Java verfügt und sich auch in der Struktur von Java-Quellcode zurechtfinden kann. Einführungen in Java und ihre Anwendung für typische Problemlösungen bei der Programmentwicklung bieten u.a. [DRBWS03], [RSSW11] und [ULLE11].

Abstrakte Klassen

In verschiedenen UML-Modellen sind sogenannte *abstrakte Klassen* vorgesehen. Sie unterscheiden sich von gewöhnlichen Klassen dadurch, dass sie keine Objekte bilden können und dass ihre Klassendeklarationen *abstrakte Methoden* enthalten. Abstrakte Methoden sind zwar als solche deklariert; jedoch fehlen ihnen die notwendigen Programmanweisungen, um sie zur Ausführung zu bringen. Kindklassen, die von abstrakten Klassen erben, müssen die abstrakten Methoden durch eigene Methoden überschreiben, die ausführbaren Programmquellcode enthalten. Abstrakte Klassen überlassen es so ihren Kindklassen, für eine vollständige Klassendeklaration zu sorgen. In Java werden abstrakte Klassen und Methoden mit dem Schlüsselwort `abstract` gekennzeichnet.

In verschiedenen Modellen der Geoinformatik sind die geometrischen Objekte zu einer abstrakten Elternklasse zusammengefasst, durch die sie unter einem gemeinsamen Datentyp angesprochen werden können. Allerdings unterscheidet sich das Verhalten von Punkten, Linien und Flächen so sehr voneinander, dass es nicht möglich ist, in der Elternklasse einen gemeinsamen Ausführungscode zu formulieren. Daher sind zahlreiche Methoden der geometrischen Elternklasse nur abstrakt deklariert.

Schnittstellen

Bei *Schnittstellen* (*Interfaces*) handelt es sich um eigenständige Java-Sprachelemente mit einer separaten Syntax, die in UML-Klassendiagrammen meist als Kreise symbolisiert sind (siehe auch **Abbildung 2.2**). Nach außen hin wirken sie wie leere Deklarationen von Elternklassen, da sie grundsätzlich nur abstrakte Methoden enthalten. Wie der Begriff Schnittstelle schon andeutet, bilden sie dadurch einen äußeren Rahmen, der das Verhalten eines Objekttyps festlegt. Praktisch verwendbar ist ein Interface erst dann, wenn es dazu

eine passende *Adapterklasse* gibt, die zu jeder abstrakten Methode des Interface eine implementierte Methode mit ausführbarem Programmcode bereitstellt.

Bedeutung von Schnittstellen

Ihren Nutzen entfalten Schnittstellen in komplexen Softwarearchitekturen, in denen die Unabhängigkeit der Software von speziellen Komponenten gewährleistet sein muss. Das zugrunde liegende Prinzip ist auch im Java-Paket `java.sql` realisiert, das zum Zugriff eines Java-Anwendungsprogramms auf ein Datenbanksystem dient. Der Java-Quellcode des Anwendungsprogramms greift nur auf Schnittstellenobjekte zu, die in `java.sql` deklariert sind. Die meisten Datenbankanbieter stellen ihrerseits eigene Treibermodule bereit, die spezielle, an die `java.sql`-Schnittstellen angepasste Adapterklassen enthalten. Soll nun das Datenbanksystem der Java-Anwendung gewechselt werden, dann reicht es im Prinzip aus, das bestehende Java-Treibermodul gegen ein passendes auszutauschen, ohne dass dabei der Programmquellcode des Anwendungsprogramms angepasst werden muss. Auf diese Weise bleibt die Unabhängigkeit des Java-Programms vom Datenbanksystem erhalten.

Delegationsprinzip

Bei der praktischen Softwareentwicklung treten Fälle auf, in denen es sinnvoll erscheint, mehrere Elternklassen in eine neue Klassendeklaration einzubinden, um die dort angebotenen Methoden nicht erneut programmieren zu müssen. Diese sogenannte *Mehrfachvererbung* wird aber von der Programmiersprache Java aus prinzipiellen Gründen unterbunden. Der Einschränkung kann stattdessen durch das *Delegationsprinzip* entgegengewirkt werden: Für jede Klasse, deren Methoden genutzt werden sollen, wird in der neuen Klassendeklaration ein eigenes Attribut angelegt. Für die Methoden der neuen Klassendeklaration, zu denen es bereits funktionsfähigen Programmcode gibt, wird die Ausführung dann an das entsprechende Attribut weitergeleitet.

Collection-Klassen

Bei der Entwicklung komplexer Software treten immer wieder ähnliche Probleme auf, die mit der Massenverarbeitung gleichartiger oder ähnlicher Objekte zusammenhängen. Statische Felder (Arrays), die als einziges Mengen-Konstrukt vom Java-Sprachstandard unterstützt werden, erweisen sich häufig als zu unflexibel. Aus diesem Grund enthält die Java-Standardbibliothek ergänzende *Collection-Klassen* (siehe auch [SASA10]). Sie gehören zum Paket `java.util` und implementieren je nachdem, wie die Objekte strukturiert werden sollen, unterschiedliche Schnittstellen:

- Eine *Menge* (`Set`)fasst Objekte zu einer Gesamtheit zusammen. Versuche, identische Objekte mehrfach einer Menge zuzuweisen, werden abgewiesen. Die in einer Menge erfassten Objekte besitzen keine Ordnung; die Reihenfolge der Ausgabe ihrer Elemente ist nicht vorhersagbar. Die bekannteste Implementierungsklasse von Set ist `HashSet`.
- In einer *Liste* (`List`) werden Objekte geordnet nach einem fortlaufenden Index gespeichert. Die gespeicherten Objekte können über ihren zugeordneten Index angesprochen werden. Listen besitzen daher Ähnlichkeit mit *statischen Feldern* (*Array*). Im Gegen-

satz zu einem Array ist aber die Speicherkapazität einer Liste nicht begrenzt, sondern wird bei Bedarf automatisch erweitert. Auch das Einfügen oder Löschen eines Objektes an einer beliebigen Position der Liste erfordert keinen zusätzlichen Programmieraufwand, der bei statischen Arrays erforderlich ist. Als Implementierung einer Liste stellt das Paket `java.util` u.a. die Adapterklasse `ArrayList` zur Verfügung, deren Elemente mit den in **Tabelle 2.1** angegebenen Methoden verwaltet werden können.

Tabelle 2.1: Häufig verwendete Methoden von Listen.

Methode	Beschreibung
add()	Fügt ein neues Element (entweder am Ende oder an der Speicherstelle von Index *i*) in die Liste ein.
contains()	Gibt an, ob ein Element in der Liste enthalten ist.
iterator()	Übergibt die Elemente in eine Folge (Iterator), mit der sie nacheinander bearbeitet werden können.
get()	Gibt das unter dem Index *i* gespeicherte Objekt zurück.
remove()	Entfernt ein Element aus der Liste.
size()	Gibt die Anzahl der gespeicherten Elemente an.
toArray()	Übergibt die Elemente in ein Feld (Array), z.B. vom Typ Objekt[] .

- *Tabellen* (`Map`) speichern Zuordnungspaare der Form *Schlüssel* (*Key*) – *Wert* (*Value*). Die gespeicherten Objekte werden, anders als bei Listen, über einen frei wählbaren Schlüssel angesprochen. Ansonsten verhalten sich Maps ähnlich wie Listen. Im Paket `java.util` gibt es u.a. die Klassen `HashMap` und `Hashtable`, mit der Zuordnungspaare gespeichert und verwaltet werden können (siehe **Tabelle 2.2**).

Tabelle 2.2: Häufig verwendete Methoden für Tabellen.

Methode	Beschreibung
put()	Fügt der Tabelle ein neues Element hinzu Parameter sind key und value.
containsKey()	Gibt an, ob ein Element unter dem gegebenen Schlüssel in der Tabelle enthalten ist.
containsValue()	Gibt an, ob ein Element unter dem gegebenen Wert in der Tabelle enthalten ist.
get()	Gibt das Objekt an, das unter dem Schlüssel gespeichert ist.
iterator()	Übergibt die Elemente in eine Folge (Iterator), mit der sie nacheinander bearbeitet werden können.
remove()	Entfernt das Element mit dem Schlüssel key aus der Tabelle.
size()	Gibt die Anzahl der gespeicherten Elemente an.
values()	Gibt alle gespeicherten Werte als Collection zurück.

Häufig besteht die Notwendigkeit, Mengen nach einem vorgegebenen Kriterium zu sortieren. Das Paket `java.util` enthält u.a. die Klassen `Arrays` und `Collections`, die zu diesem Zweck effiziente Sortieralgorithmen bereitstellen. Die Elemente, die sortiert werden sollen, müssen allerdings die Schnittstelle `Comparable` implementieren, die ihrerseits die zentrale Methode `compareTo()` vorschreibt. Sie wird dazu verwendet, um eine relative Ordnung zwischen den Objekten des gleichen Typs festzulegen.

Listing 2.4 demonstriert die Verwendung der Klassen `ArrayList` und `Arrays`. Das Beispielprogramm erzeugt zunächst zehn Zufallszahlen vom Typ `Double` und speichert sie in der Liste `list`. Mit der Klasse `Arrays` werden die Elemente zuerst in der Reihenfolge ihrer Eingabe und danach sortiert ausgegeben. Die Klasse `Double` implementiert zu diesem Zweck die Schnittstelle `Comparable`.

Listing 2.4: Beispielprogramm zur Verwendung der Klassen ArrayList und Arrays.

```
import java.util.ArrayList;
import java.util.Arrays;

public class ArrayListDemo {

    public static void main(String[] args) {

            // Anzahl der Elemente
            int n = 10;
            // Erzeugt eine ArrayList zur Speicherung von Zufallszahlen
            ArrayList list = new ArrayList();

            // Erzeugt Zufallszahlen und fügt sie in die Liste ein
            for(int i=0; i<n; i++) {
                    Double random = new Double(Math.random());
                    list.add(random);
            }

            // Gibt die Liste in der Eingabereihenfolge aus.
            System.out.println("Liste in Eingabereihenfolge:");
            for(int i=0; i<n; i++) {
                    System.out.println(list.get(i));
            }

            // Sortiert die Liste
            Object[] randoms = list.toArray();
            Arrays.sort(randoms);

            // Gibt die Liste sortiert aus.
            System.out.println("Liste in Sortierreihenfolge:");
            for(int i=0; i<n; i++) {
                    System.out.println(randoms[i]);
            }
    }
}
```