

Einführung

In den bisherigen Kurseinheiten lernten wir die grundlegenden Sprachelemente und Konzepte zur Erstellung objektorientierter Programme kennen.

Java besitzt noch viele weitere Elemente, wie zum Beispiel die umfangreiche API, inklusive der Unterstützung von graphischen Oberflächen, generischen Typen, Annotationen und Reflection. Die **API** und **generische Typen** werden wir im Rahmen dieser Kurseinheit noch etwas genauer betrachten. Bei der API werden wir insbesondere auf die **Ein- und Ausgabe von Daten** eingehen, da das Konsumieren und Produzieren von Daten eine wichtige Aufgabe der meisten Programme ist. Bei generischen Typen werden wir uns auf deren Verwendung konzentrieren.

Zusätzlich zu den oben genannten Sprachelementen gibt es noch Elemente, die nebenläufige Programmierung mit Java ermöglichen. Diese Konzepte können wir jedoch in diesem Kurs nicht weiter vertiefen.

Dass wir aber auch schon mit unserem bisher erworbenen Wissen erste einfache Anwendungen entwickeln können, werden wir an der **Fallstudie**, mit der wir den Kurs abschließen wollen, feststellen.

Lernziele

- Die Java-API verwenden und ihre wichtigsten Pakete kennen.
- Das Konzept der Datenströme und seine Umsetzung in Java kennen und anwenden können.
- Das Prinzip generischer Typen kennen.
- Generische Typen verwenden können.
- Das Prinzip von Iteratoren und ihre Umsetzung in Java kennen und anwenden können.
- Die erweiterte `for`-Schleife bei Objekten des Typs `Iterable` anwenden können.

38 API

Die Java-API²³ (Application Programming Interface) umfasst eine reichhaltige Bibliothek von Schnittstellen und Klassen, die Lösungen für eine Vielzahl von Aufgaben zur Verfügung stellt, die in der Praxis häufig auftreten.

Java-API

Um Sie beim Durchstöbern der Java-Bibliothek für künftige Programmieraufgaben zu unterstützen, beschreiben wir kurz einige der gängigsten Java-Pakete:

- `java.lang` (für engl. *language*),
- `java.util` (für engl. *utilities*),
- `java.io` (für engl. *input-output*),
- `java.math` (für engl. *mathematics*).

Das Paket `java.lang` offeriert Klassen, die das Kernstück der Sprache bilden. Als einziges Paket der Java-API wird es von jeder Klasse automatisch, d. h. ohne explizite `import`-Anweisung importiert. Es ist kaum möglich, ein Java-Programm zu schreiben, ohne Klassen und Schnittstellen aus diesem Paket zu benutzen. Zu den wichtigsten Klassen, die im Paket `java.lang` enthalten sind, gehören:

`java.lang`

- `Object`,
- `String` und `StringBuilder`,
- `System`,
- `Class`,
- Hüllklassen für primitive Datentypen,
- `Throwable` und diverse weitere Ausnahmeklassen sowie
- `Math`.

Die Klassen `Object`, `String`, `StringBuilder`, `System` sowie diverse Ausnahmetypen lernten wir schon in bisherigen Kurseinheiten kennen.

Objekte der Klasse `Class` repräsentieren zur Laufzeit jeweils eine existierende Klasse oder Schnittstelle. Sie speichern Informationen wie den Namen und die Oberklasse sowie die deklarierten Methoden, Attribute und Konstruktoren. Für jedes Objekt kann mit Hilfe der Methode `getClass()` das zugehörige `Class`-Objekt erfragt werden.

`Class`

Manchmal, zum Beispiel bei der Verwendung generischer Klassen (siehe Kapitel 40), werden Objekte an Stelle primitiver Daten benötigt. Für solche Zwecke gibt es sogenannte Hüllklassen (engl. *wrapper class*). Diese Klassen heißen `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character` und `Boolean`. Seit

Hüllklassen

23 <http://java.sun.com/j2se/1.5.0/docs/api/>

Version 5 können primitive Werte automatisch in entsprechende Hüllobjekte umgewandelt werden (engl. *auto boxing*) und umgekehrt ebenso:

```
int zahl = 1;
Integer zahlObjekt = zahl;
int summe = zahl + zahlObjekt;
```


Die Hüllklassen bieten weiterhin einige nützliche Methoden. Mit Hilfe der statischen Methode `parseInt()` der Klasse `Integer` kann beispielsweise ein entsprechender Wert aus einer Zeichenkette gewonnen werden.

Die Klasse `Math` stellt Konstanten und eine Reihe statischer Methoden zur Berechnung diverser mathematischer Funktionen, wie zum Beispiel Sinus und Cosinus zur Verfügung.

`java.util` Das Paket `java.util` stellt verschiedenste Klassen und Schnittstellen zur Verfügung. Diese reichen von Klassen zur Repräsentation von Daten (`Date` und `Calendar`) und Währungen (`Currency`) bis zu einer Vielzahl von Datenstrukturen, wie wir sie in der letzten Kurseinheit kennen gelernt haben. Auf einige Datenstrukturen werden wir gesondert in Kapitel 40 eingehen.

`java.io` Das Paket `java.io` enthält Klassen, die das Schreiben und Lesen von Daten über unterschiedliche Ein- und Ausgabegeräte und Dateien unterstützen. Diese werden wir in Kapitel 39 genauer untersuchen.

`java.math` Die primitiven Datentypen weisen alle nur einen begrenzten Wertebereich auf. Um mit beliebig großen Zahlen arbeiten zu können, stellt das Paket `java.math` die Klasse `BigInteger` und `BigDecimal` zur Verfügung.

 Bevor Sie eine Klasse neu schreiben, sollten Sie immer zunächst in der API nachsehen, ob dort schon eine entsprechende Klasse vorhanden ist.

In den folgenden Kapiteln werden wir einige der API-Klassen und ihre Verwendung näher kennen lernen.

Selbsttestaufgabe 38-1:

Schreiben Sie eine Methode `printLeapYears()`, die zwei Jahreszahlen als Parameter entgegen nimmt und alle Schaltjahre am Bildschirm ausgibt, die zwischen diesen Jahren liegen:

```
public void printLeapYears(int fromYear, int toYear) {
    // ...
}
```

Lösungshinweis: Finden und verwenden Sie eine geeignete Klasse der Java-API. Konsultieren Sie die Java-API-Referenz²⁴ und vergessen Sie nicht, die verwendete Klasse zu importieren.



24 <http://java.sun.com/j2se/1.5.0/docs/api/>

Selbsttestaufgabe 38-2:

Mit welcher Methode einer geeigneten Hüllklasse stellen Sie fest, ob ein char-Wert ein Kleinbuchstabe ist?



Selbsttestaufgabe 38-3:

Schreiben Sie eine Methode `radius()`, die zu einer gegebenen Kreisfläche den Radius ermittelt:

```
public double radius(double flaeche) {  
    // ...  
}
```



39 Ein- und Ausgabe

Seit dem Anfang der Softwareentwicklung dienen Programme in irgendeiner Art und Weise der Verarbeitung von Daten. Daten werden durch Programme aus einer Quelle gelesen, verarbeitet und wieder ausgegeben. Typische Beispiele für Datenquellen sind zum Beispiel:

Datenquelle

- die Tastatur,
- eine Datei,
- eine Datenbank oder
- eine Netzwerkverbindung.

Nachdem ein Programm die eingelesenen Daten verarbeitet hat, müssen die Ergebnisse wieder ausgegeben oder an eine Datensenke abgegeben werden. Typische Beispiele für Datensenken sind physische Geräte oder Datenspeicher:

Datensenke

- der Bildschirm,
- ein Drucker,
- eine Datenbank,
- eine Datei oder
- eine Netzwerkverbindung

Um in Java die Daten von einer Datenquelle zu einer Datensenke zu transportieren, existieren sog. Datenströme (engl. *data stream*). Ströme können als Kanäle gesehen werden, die die Daten jeweils in eine Richtung, also entweder lesend oder schreibend, transportieren. Diese Kanäle können weiterhin hintereinander gekoppelt werden.

Datenstrom

Das Paket `java.io` der Java-Klassenbibliothek enthält die Klassen, welche die jeweiligen Datenströme kapseln. Als abstrakte Basisklassen dienen die Klassen `InputStream` für lesenden und `OutputStream` für schreibenden Zugriff. Da die beiden Klassen lediglich das Lesen und Schreiben einzelner Bytes ermöglichen, gibt es zusätzlich die abstrakten Klassen `Reader` und `Writer`, die `InputStream`- bzw. `OutputStream`-Objekte kapseln und das Verarbeiten von Zeichen (`char`) ermöglichen (vgl. Abb. 39-1).

Sobald ein Java-Programm gestartet wird, erzeugt die virtuelle Maschine automatisch drei Standarddatenströme. Die Eingabe über die Tastatur wird im sogenannten Standardeingabestrom (engl. *standard input stream*) `System.in` übertragen. Den zugehörigen Standardausgabestrom `System.out` haben wir schon häufiger benutzt. Der Standardfehlerstrom, `System.err`, wird in der Regel für die Ausgabe von Fehlermeldungen genutzt.

Standarddatenstrom

`BufferedReader` Zum Einlesen verwenden wir in der Regel die Klasse `BufferedReader`, da die-

Lesen



Schreiben

**Abb. 39-1:** Hintereinander gekoppelte Ströme

se es ermöglicht, ganze Zeilen einzulesen. Die Konstruktor erwartet einen weiteren Reader. Wollen wir vom Standardeingabestrom lesen, so verwenden wir einen `InputStreamReader`. Wollen wir Daten aus einer Datei einlesen, so ist `FileReader` die richtige Wahl. Die folgenden Anweisungen lesen solange Eingaben von der Tastatur, bis die Zeichenkette "Ende" eingegeben wird oder keine Eingaben mehr erfolgen. Der Datenstrom sollte am Ende immer mit der Methode `close()` geschlossen werden. Wenn Daten eingelesen werden, kann es vorkommen, dass unvorhergesehene Ereignisse diese Aktion unterbrechen. In einem solchen Fall wird eine `IOException` geworfen. Diese können wir mit einer `try`-Anweisung abfangen und behandeln. Da auch das Schließen des Datenstroms eine `IOException` werfen kann, wird es im `finally`-Block einer inneren `try`-Anweisung behandelt.

```

// neuen Reader anlegen
BufferedReader br = new BufferedReader(
    new InputStreamReader(System.in));

try {
    try {
        // erste Zeile einlesen
        String line = br.readLine();
        // solange nicht "Ende" eingelesen wird
        while (line != null && !"Ende".equals(line)) {
            // Zeichenkette verarbeiten
            // ...
            // nächste Zeile einlesen
            line = br.readLine();
        }
    } finally {
        // Datenstrom schließen
        br.close();
    }
} catch (IOException e) {
    System.out.println("Fehler beim Einlesen.");
}
  
```

Selbsttestaufgabe 39-1:

Entwickeln Sie eine Klasse `Rechner`, die im Konstruktor einen Dateinamen übergeben bekommt und die eine Methode `long addiere()` besitzt, die anschließend alle Zahlen in der Datei mit Hilfe eines `BufferedReader` einliest und addiert. Gehen Sie davon aus, dass in der Datei jede Zahl in einer eigenen Zeile steht. Methoden, die eine Zeichenkette in eine Zahl umwandeln können, finden Sie in der API.



Alternativ zu der Klasse `BufferedReader` kann zum Einlesen auch die Klasse `Scanner` `java.util.Scanner` verwendet werden.

Selbsttestaufgabe 39-2:

Entwickeln Sie eine Klasse `ScannerRechner`, die sich wie die Klasse `Rechner` verhält, aber statt der Klasse `BufferedReader` die Klasse `Scanner` verwendet.



Um Daten auszugeben, können wir beispielsweise die Klasse `PrintWriter` verwenden. Diese erwartet im Konstruktor entweder einen `OutputStream`, eine Datei oder einen Dateinamen.

Selbsttestaufgabe 39-3:

Entwickeln Sie eine Klasse `Filter`, die im Konstruktor den Namen der Eingabedatei und den Namen der Ausgabedatei übergeben bekommt. Die Methode `filter()`, soll alle Zeilen aus der Eingabedatei einlesen und nur diejenigen Zeilen in die Ausgabedatei schreiben, die mit einem Großbuchstaben beginnen. Die Reihenfolge der Zeilen soll dabei nicht verändert werden.



40 Generische Typen

Bisher haben wir Containerklassen wie Listen und Bäume selber programmiert. Die Java-API bietet jedoch im Paket `java.util` eine Menge vordefinierter Containerklassen, die die Schnittstelle `java.util.Collection` implementieren.

Collection

Unsere selbst implementierten Klassen hatten das Problem, dass wir sie immer nur für eine Art von Elementen schreiben konnte. Hatten wir eine Liste mit ganzen Zahlen und benötigten aber eine Liste für Zeichenketten, so musste mit mühsamer Handarbeit alles angepasst werden. Um solche Änderungen nicht mehr vornehmen zu müssen, könnten wir allgemein eine Liste implementieren, die Objekte der Klasse `Object` speichern kann. Selbst Werte primitiver Datentypen könnten dank der Hüllklassen in solchen Containerklassen gespeichert werden. Allerdings könnten in einer solchen Liste dann auch Zeichenketten und Zahlen gleichzeitig gespeichert sein. Dies hätte zur Folge, dass nie bekannt ist, von welchem Typ die gespeicherten Objekte genau sind, und man bei der Entnahme eines Elements keine Aussage mehr treffen könnte, welchen Typ dieses Element besitzt.

Um Container und andere Klassen flexibel, aber dennoch typsicher gestalten zu können, wurden mit Java 5 generische Klassen eingeführt, mit denen wir uns im Laufe dieses Kapitels beschäftigen wollen.

generische Klasse

40.1 Verwendung generischer Typen

Zum Einstieg werden wir die Schnittstelle `java.util.List` und die Klasse `java.util.LinkedList`, die diese Schnittstelle implementiert, betrachten.

Wir können eine solche Liste zunächst ohne das Wissen über generische Klassen erzeugen, mit der Folge, dass in der Liste beliebige Objekte gespeichert werden können. Wir erhalten allerdings eine Warnung des Übersetzers, da wir die Liste nicht als generische Klasse nutzen.

```
// neue Liste erzeugen
List l = new LinkedList();
// Zeichenkette hinzufügen
l.add("Hallo");
// Zahl hinzufügen
l.add(3); // hier erfolgt automatisch die Erzeugung eines
         // Integer-Objekts
// das erste Element entnehmen
Object o = l.get(0);
```

Generische Klassen [JLS: § 8.1.2, § 9.1.2] besitzen zusätzlich zu ihrem Namen noch eine oder mehrere Typvariablen [JLS: § 4.4]. Diese Typvariablen werden auch als Typparameter bezeichnet. Die Typparameter werden in spitzen Klammern angege-

Typvariable

Typparameter

ben. Eine Liste von Zeichenketten würde mit dem Typ `List<String>` und eine Liste von ganzen Zahlen mit dem Typ `List<Integer>` deklariert werden. Dabei ist zu beachten, dass Typparameter nur Klassen und keine primitiven Datentypen sein können. Betrachten wir die Dokumentation der Schnittstelle `List`, so sehen wir, dass der Typparameter mit `E` bezeichnet ist und auch in Methoden als Parameter oder Ergebnistyp auftritt. So können bei einem Methodenaufruf von `add()` an einer Variable vom Typ `List<String>` nur Zeichenkettenobjekte übergeben werden. Dadurch wird sichergestellt, dass die Liste jederzeit nur Zeichenketten enthält. Die Methode `get()` liefert dementsprechend auch nur Zeichenketten zurück. Die Typprüfung findet bereits zur Übersetzungszeit und nicht erst zur Laufzeit statt.

```
// neue Liste erzeugen
List<String> l = new LinkedList<String>();
// Zeichenkette hinzufügen
l.add("Hallo");
// Zahl hinzufügen würde einen Übersetzungsfehler erzeugen
// l.add(3);
// das erste Element, eine Zeichenkette, entnehmen
String zeichenkette = l.get(0);
```

Selbstverständlich können gleichzeitig mehrere Listen mit gleichen oder mit unterschiedlichen Typparametern verwendet werden.

```
// neue Liste erzeugen
List<String> l1 = new LinkedList<String>();
List<Integer> l2 = new LinkedList<Integer>();
// Zeichenkette zu l1 hinzufügen
l1.add("Hallo");
// Zahl zu l2 hinzufügen
l2.add(3);
// das erste Element von l1, eine Zeichenkette, entnehmen
String zeichenkette = l1.get(0);
// das erste Element von l2, eine Zahl, entnehmen
int zahl = l2.get(0); // hier wird automatisch das Integer-
                     // Objekt in einen int-Wert umgewandelt
l2.add(4);
l2.add(3);
l2.add(5);
// Teilliste erzeugen
List<Integer> l3 = l2.subList(1, 3);
```

Neben der Erzeugung von Teillisten bieten die Containerklassen der API noch viele weitere nützliche Methoden. Das Prinzip der Iteratoren werden wir im nächsten Abschnitt kennen lernen.

40.2 Iteratoren

Iterator Ein Iterator bietet die Möglichkeit, die Elemente einer Datensammlung linear zu

durchlaufen, ohne dabei Details über die Struktur der Datensammlung zu kennen. Die Schnittstelle `Collection` besitzt die Methode `iterator()`, die ein `java.util.Iterator`-Objekt für den aktuellen Container zurück liefert.

`Iterator`

Ein `Iterator` besitzt die drei Methoden

- `hasNext()`, die prüft, ob es noch weitere unbesuchte Elemente gibt,
- `next()`, die das nächste Element zurück liefert und dann den `Iterator` auf das folgende Element setzt, sowie
- `remove()`, die das letzte vom `Iterator` zurückgelieferte Element aus der Sammlung entfernt.

Wir können folglich mit einem `Iterator` alle Elemente einer beliebigen Sammlung von Zeichenketten durchlaufen und beispielsweise ausgeben.

```
public void print(Collection<String> c) {
    // ungültige Argumente abfangen
    if (c == null) {
        return;
    }
    // Iterator erzeugen
    Iterator<String> it = c.iterator();
    // solange Elemente vorhanden
    while (it.hasNext()) {
        // nächstes Element holen
        String s = it.next();
        // Element ausgeben
        System.out.println(s);
    }
}
```

Die Methode `remove()` wird nicht von allen Sammlungen unterstützt und kann ggf. eine `UnsupportedOperationException` liefern. Sie kann zudem, falls das letzte Element schon entfernt wurde oder `next()` noch nicht aufgerufen wurde, eine `IllegalStateException` werfen.

Wird die Methode `next()` aufgerufen, obwohl `hasNext()` `false` zurückgeliefert hat, so wird eine `NoSuchElementException` geworfen.

Nicht nur Klassen, die die Schnittstelle `Collection` implementieren, stellen einen `Iterator` zur Verfügung, sondern alle Klassen, die die generische Schnittstelle `java.lang.Iterable` implementieren. `Collection` erweitert die Schnittstelle `Iterable`.

`Iterable`

Allgemein können wir nach dem obigen Muster jede Sammlung an Daten, die die Schnittstelle `Iterable` implementiert, durchlaufen.

Alternativ lässt sich das Durchlaufen auch mit Hilfe einer `for`-Schleife implementieren.

```

public void print(Iterable<String> i) {
    // ungültige Argumente abfangen
    if (i == null) {
        return;
    }
    for(Iterator<String> it = i.iterator(); it.hasNext();) {
        String s = it.next();
        System.out.println(s);
    }
}

```

erweiterte for-Schleife
bei Iterable

Die erweiterte for-Schleife, die wir zum Durchlaufen von Feldern kennen gelernt haben (vgl. Abschnitt 20.3), lässt sich ebenfalls auf Objekte vom Typ `Iterable` anwenden [JLS: § 14.14.2]. So können wir die Methode `print()` auch mit Hilfe der erweiterten for-Schleife formulieren:

```

public void print(Iterable<String> i) {
    // ungültige Argumente abfangen
    if (i == null) {
        return;
    }
    for(String s : i) {
        System.out.println(s);
    }
}

```

Beachten Sie, dass es sich sowohl bei `Iterator` als auch bei `Iterable` um generische Typen handelt.

Selbsttestaufgabe 40.2-1:

Implementieren Sie eine Methode

```
void findeKuerzesteUndLaengsteZeichenkette(List<String> l)
```

die die kürzeste und die längste in der Liste enthaltene Zeichenkette ausgibt. Gibt es mehrere Zeichenketten mit gleich vielen Zeichen, so wird diejenige ausgewählt, die an einer kleineren Position in der Liste steht.



40.3 Weitere generische Typen

Bisher haben wir verschiedene Container und das Iteratorkonzept in Verbindung mit generischen Klassen kennen gelernt. Weitere wichtige generische Schnittstellen sind `java.lang.Comparable<T>` und `java.util.Comparator<T>`. Mit Hilfe dieser Schnittstellen kann eine totale Ordnung auf den Typen ausgedrückt werden, die sich beispielsweise für eine Sortierung der Elemente verwenden lässt.

Implementiert eine Klasse `X` die Schnittstelle `Comparable<T>` so können Exemplare der Klasse `X` mit Objekten vom Typ `T` verglichen werden. Dabei müssen die Typen `X` und `T` nicht identisch sein. Soll eine schon existierende Klasse `Y` mit einer Ordnung versehen werden, wird eine Klasse, die die Schnittstelle `Comparator<Y>` implementiert, entworfen.

Weitere interessante Methoden bieten die Klassen `java.util.Arrays` und `java.util.Collections`. Diese beiden Klassen sind zwar keine generischen Klassen, bieten aber generische Methoden [JLS: § 8.4.4] an.

generische Methode

Selbsttestaufgabe 40.3-1:

Implementieren Sie die nachfolgende Methode `sort()`, die das gegebene Feld mit Elementen des Typs `T` aufsteigend mit Hilfe des Algorithmus Bubblesort sortiert. `T` muss auf Grund der Vorgaben das Interface `Comparable` implementieren und besitzt deshalb die Methode `compareTo(T)`. Weitere Hinweise finden Sie in Abschnitt 40.4.

```
public class Bubblesorter<T extends Comparable<T>> {
    public void sort(T[] feld) {
        // ...
    }
}
```



Weitere wichtige generische Containertypen sind die Schnittstellen `Set` (dt. Menge) und `Map` (dt. Zuordnung). In Objekten vom Typ `Set` werden Elemente, im Gegensatz zu Listen, ungeordnet gespeichert. Zudem können niemals zwei Objekte `o1` und `o2` gleichzeitig in einer Menge enthalten sein, wenn `o1.equals(o2)` `true` zurück liefert. Deshalb kann auch maximal einmal der `null`-Verweis in einer Menge gespeichert sein.

```
Set<String> menge = new HashSet<String>();
String a = "Hallo Welt!";
String b = "Hallo";
String c = "Welt!";
String d = b + " " + c;
menge.add(a); // liefert true
menge.add(b); // liefert true
menge.add(c); // liefert true
menge.add(d); // liefert false, da gleiche Zeichenkette
               // schon enthalten
menge.size(); // liefert 3
menge.contains(b); // liefert true
menge.contains(d); // liefert true
menge.contains("Test"); // liefert false
menge.remove(d); // liefert true, da zu löschender Wert
                 // enthalten
menge.size(); // liefert 2
```

Häufig benötigt man in der Programmierung auch Zuordnungen zwischen verschiedenen Elementen. So will man beispielsweise auf Grund einer eindeutigen Nummer auf ein bestimmtes Objekt zugreifen, beispielsweise von einer Artikelnummer auf das zugehörige Artikel-Objekt. In solchen Fälle können `Map<K, V>`-Objekte verwendet werden. Diese weisen einen Schlüsseltyp (engl. *key*) `K` und einen Werttyp (engl. *value*) `V` auf. Eine `Map` ermöglicht es, Schlüssel-Wert-Paar hinzuzufügen und anschließend den Wert zu einem bestimmten Schlüssel zu erfragen. Dabei kann zu jedem Schlüssel nur ein Wert gespeichert werden. Wird ein schon enthaltener Schlüssel mit einem anderen Wert hinzugefügt, so wird der alte Wert überschrieben.

```
Map<Integer, String> m = new HashMap<Integer, String>();
String a = "Hallo";
String b = "Welt!";
m.put(10, a); // liefert null, da vorher kein Wert zu 10
               // gespeichert war
m.put(12, b); // liefert null, da vorher kein Wert zu 12
               // gespeichert war
m.get(10); // liefert "Hallo"
m.get(4); // liefert null, da kein zugehöriger Wert
           // gespeichert ist
m.put(10, "Test"); // liefert "Hallo", da dieser vorher mit
                   // dem Wert 10 verknüpft war
m.get(10); // liefert "Test"
m.size(); // liefert 2, da nur zwei verschiedene Paar
           // gespeichert sind
m.remove(12); // liefert "Welt!"
m.size(); // liefert 1
```

40.4 EXKURS: Deklaration generischer Typen

Bisher haben wir uns nur mit der Verwendung generischer Typen beschäftigt. Aber auch die Deklaration eines generischen Typs ist nicht weiter kompliziert. Im Folgenden werden wir versuchen, ein Paar bzw. Tupel, bestehend aus einem Schlüssel und einem Wert, als generische Schnittstelle zu deklarieren und anschließend eine entsprechende Implementierung zu entwickeln.

Ein solches Paar besitzt zwei Typparameter. So könnte man sich beispielsweise Paare, die aus einer Zeichenkette und einer Zahl, aus zwei Zeichenketten oder aus beliebigen anderen Objekten bestehen, vorstellen.

Diese beiden Typparameter, wir nennen sie `S` und `W`, werden in spitzen Klammern direkt hinter dem Typnamen angegeben [JLS: § 8.1, § 8.1.2].

```
public interface Paar<S, W> {
    // ...
}
```

Innerhalb einer `Paar<S,W>` implementierenden Klasse können die Parameter `S` und `W` als Typen verwendet werden. So können wir die Methode `liefereSchluessel()` und `liefereWert()` mit entsprechenden Ergebnistypen deklarieren.

```
public interface Paar<S,W> {  
    public S liefereSchluessel();  
    public W liefereWert();  
}
```

Selbsttestaufgabe 40.4-1:

Ergänzen Sie die Schnittstelle um entsprechende Setter-Methoden und eine Methode `istGleich()` die ein anderes Paar mit den gleichen Typparametern erwartet und vergleicht ob die Schlüssel und Werte gleich sind. Ergänzen Sie auch Javadoc-Kommentare.



Nun wollen wir auch eine Klasse `PaarImpl` entwickeln, die das Interface `Paar` implementiert. Dafür benötigt auch die Klasse `PaarImpl` entsprechende Typparameter. Genauso wie wir die Typparameter als Ergebnis- und Parametertypen verwendet haben, können wir diese auch nutzen, um Attribute zu deklarieren.

```
public class PaarImpl<S,W> implements Paar<S,W> {  
    private S schluessel;  
    private W wert;  
  
    public S liefereSchluessel() {  
        return this.schluessel;  
    }  
  
    public W liefereWert() {  
        return this.wert;  
    }  
  
    // ...  
}
```

Selbsttestaufgabe 40.4-2:

Implementieren Sie die restlichen Methoden aus Selbsttestaufgabe 40.4-1 und ergänzen Sie auch einen passenden Konstruktor.



In manchen Fällen ist es auch nötig auszudrücken, dass ein Typparameter Subtyp einer bestimmten Menge an Typen ist. Solche Einschränkungen werden bei der Einführung des Typparameters angegeben [JLS: § 4.4].

```
public class X<E extends F & G> {  
    // ...  
}
```

Als Typparameter E wären nur Typen erlaubt, die Subtypen von F und G sind. Dabei darf lediglich der erste der Obertypen eine Klasse sein.

Selbsttestaufgabe 40.4-3:

Entwickeln Sie aus der Listenimplementierung aus Kapitel 35 eine generische Listenklasse. Welche Einschränkungen müssen die Typparameter erfüllen, wenn die Elemente automatisch beim Einfügen sortiert werden sollen?



Weiterführende Literatur

[Bracha04] Gilad Bracha

Generics in the Java Programming Language²⁵

Sun Java Tutorials, 2004

²⁵ <http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>

41 Fallbeispiel

Zum Abschluss dieses Kurses wollen wir noch einmal zurückschauen auf unsere Implementierungsbeispiele für den Blumenladen. Wir wollen unsere inzwischen erworbenen Kenntnisse einsetzen, um eine eigenständige Anwendung zu entwickeln, mit der wir Rechnungen für unseren Blumenladen erstellen und abspeichern können.

Dieses abschließende Fallbeispiel soll der Wiederholung dienen und Sie ermutigen, das im Kurs Gelernte auch in etwas umfangreicheren Klassengeflechten anzuwenden. Wir werden sehen, dass wir einige Implementierungen aus vorhergehenden Kurseinheiten nahezu unverändert verwenden können. An anderen Stellen wird uns auffallen, dass wir inzwischen bessere Lösungen kennen und auch neue sinnvolle Funktionalität ergänzen können. So werden wir uns auch in diesem Kapitel noch einmal mit generischen Datenstrukturen und der Verwendung des Dateisystems beschäftigen.

Unser Fallbeispiel mündet in eine lauffähige Anwendung, deren vollständigen Quelltext Sie auf der Webseite des Kurses finden. Schlagen Sie aber auch im Kurs-text nach, um die Implementierungen in diesem Kapitel mit früheren Lösungen und Beispielen zu vergleichen.

41.1 Verwendung generischer Datenstrukturen

Betrachten wir noch einmal unsere frühere Implementierung der Klasse `Rechnung` (vgl. Kapitel 20), so fällt uns auf, dass wir damals die Beschränkung eingeführt hatten, dass eine Rechnung nicht mehr als 100 Rechnungsposten enthalten darf. Da wir inzwischen variable Datenstrukturen kennen, können wir statt eines Feldes für Rechnungsposten eine Liste benutzen und so diese Einschränkung entfernen.

In der Regel lassen sich für solche Fälle geeignete Klassen der API verwenden. Für unseren Fall reicht eine lineare Liste aus. Um das Ganze möglichst variabel zu gestalten, verwenden wir als Typ für unser Attribut `posten` die Schnittstelle `List`. So können wir später die konkrete Implementierung leicht austauschen. Für den Anfang wählen wir eine `LinkedList`.

Neben der Änderung des Attributs müssen wir auch alle Zugriffe auf dieses Attribut entsprechend anpassen. Die Klassen `Artikel`, `Rechnungsposten` und `Kunde` können wir unverändert übernehmen.

```
class Rechnung {
    static int naechsteRechnungsnummer = 10000;

    double rabatt;
    Kunde rechnungsempfaenger;
    List<Rechnungsposten> posten
        = new LinkedList<Rechnungsposten>();
    final int rechnungsnummer;

    Rechnung(Kunde empfaenger) {
        this.rechnungsempfaenger = empfaenger;
        this.rechnungsnummer
            = Rechnung.berechneNaechsteRechnungsnummer();
    }

    void fuegePostenHinzu(final Rechnungsposten rp) {
        this.posten.add(rp);
    }

    double berechneNettopreis() {
        double summe = 0;
        for (Rechnungsposten rp : posten) {
            summe += rp.berechneGesamtbetrag();
        }
        return summe * (1 - this.liefereRabatt());
    }

    double berechneMehrwertsteuer() {
        double summe = 0;
        for (Rechnungsposten rp : posten) {
            summe += rp.berechneGesamtbetrag()
                * rp.liefereArtikel().liefereMehrwertsteuer();
        }
        return summe * (1 - this.liefereRabatt());
    }

    double berechneBruttoPreis() {
        return this.berechneNettopreis()
            + this.berechneMehrwertsteuer();
    }

    void gebeAus() {
        System.out.println("Rechnung Nr. " + this.rechnungsnummer);
        System.out.println("An:");
        System.out.println(this.liefereRechnungsempfaenger().
            liefereName());
        System.out.println(this.liefereRechnungsempfaenger().
            liefereAnschrift());
        System.out.println("Artikel:");
    }
}
```

```

    for (Rechnungsposten rp : posten) {
        System.out.println(rp.liefereAnzahl() + " x Nr. "
            + rp.liefereArtikel().liefereArtikelnummer() + " "
            + rp.liefereArtikel().liefereBeschreibung());
    }

    System.out.println("Netto: " + this.berechneNettopreis());

    System.out.print("MwSt: ");
    System.out.println(this.berechneMehrwertsteuer());
    System.out.print("Brutto: ");
    System.out.println(this.berechneBruttoPreis());
}

// weitere Methoden
}

```

Mit Hilfe der erweiterten `for`-Schleife konnten wir das Durchlaufen der Rechnungsposten weiter vereinfachen.

41.2 Verwendung des Dateisystems

In den meisten Anwendungen müssen Daten dauerhaft gespeichert werden. Für solche Fälle können entweder einfache Dateien oder Datenbanken verwendet werden. Wir wollen in unserer Anwendung Rechnungen dauerhaft speichern, indem für jede Rechnung eine Textdatei erstellt wird, deren Name die Rechnungsnummer enthält. Wir ergänzen die Klasse `Rechnung` um eine Methode `void speichern()`. Diese erzeugt zunächst ein neues `File`-Objekt mit dem passenden Namen. Anschließend wird überprüft, ob es diese Datei schon gibt. Sollte dies der Fall sein, so brechen wir ab, da wir sonst die alte Datei überschreiben würde. Existiert die Datei noch nicht, so erzeugen wir sie.

```

void speichern() {
    File f = new File(rechnungsnummer + ".txt");
    if (f.exists()) {
        System.out.println("Rechnungsdatei existiert schon, "
            + "Rechnung wird deshalb nicht gespeichert.");
    }
    // ...
}

```

Um die Informationen in die Datei schreiben zu können, erzeugen wir ein `PrintWriter`-Objekt. Die Daten, die ausgegeben werden sollen, sind mit denen der Methode `gebeAus()` identisch. Deshalb führen wir eine gemeinsame Methode ein, die ein `PrintWriter`-Objekt erwartet. Die bisherige Methode `gebeAus()` passen wir so an, dass sie die gemeinsame Methode nutzen kann.

```

void speichern() {
    File f = new File(rechnungsnummer + ".txt");
    if (f.exists()) {
        System.out.println("Rechnungsdatei existiert schon, "
            + "Rechnung wird deshalb nicht gespeichert.");
    }
    PrintWriter pw;
    try {
        pw = new PrintWriter(f);
        gebeAus(pw);
        pw.close();
    } catch (FileNotFoundException e) {
        System.out.println("Rechnungsdatei konnte nicht "
            + "gefunden werden, Rechnung wird deshalb "
            + "nicht gespeichert.");
    }
}

private void gebeAus(PrintWriter pw) {
    pw.println("Rechnung Nr. " + this.rechnungsnummer);
    pw.println("An:");
    pw.println(this.liefereRechnungsempfaenger().
        liefererName());
    pw.println(this.liefereRechnungsempfaenger().
        liefererAnschrift());
    pw.println("Artikel:");
    for (Rechnungsposten rp : posten) {
        pw.println(rp.liefereAnzahl() + " x Nr. "
            + rp.liefereArtikel().liefererArtikelnummer() + " "
            + rp.liefereArtikel().liefererBeschreibung());
    }
    pw.println("Netto: " + this.berechneNettopreis());
    pw.println("MwSt: " + this.berechneMehrwertsteuer());
    pw.println("Brutto: " + this.berechneBruttoPreis());
    pw.flush();
}

void gebeAus() {
    gebeAus(new PrintWriter(System.out));
}

```

Die Methode `flush()` stellt sicher, dass alle Daten wirklich in die Datei geschrieben wurden, bevor diese ggf. geschlossen wird.

Um Rechnungen ausstellen zu können, benötigen wir die Informationen für die verschiedenen Artikel. Diese wollen wir in einer einfachen Textdatei vorhalten, wobei in jeder Zeile ein Artikel gespeichert ist und die Attribute jeweils mit einem Tabulator getrennt sind. Nach der Artikelnummer folgen der Preis und der Mehrwertsteuersatz und abschließend die Beschreibung. Alle Artikel sollen zur Laufzeit in einem Map-Objekt zur Verfügung stehen. Dieses Objekt wird im Exemplarattribut `artikeldaten` unserer Kassenanwendung gespeichert. Das Einlesen der

Textdatei erfolgt in der Methode `leseArtikelEin()`, die einmalig beim Start der Anwendung ausgeführt wird. Sie liest die einzelnen Zeilen ein, anschließend identifiziert eine gesonderte Methode die einzelnen Artikeleigenschaften.

```
public class Kassenanwendung {
    private Map<Long, Artikel> artikeldaten;
    // ...
    private void leseArtikelEin() {
        this.artikeldaten = new HashMap<Long, Artikel>();
        File f = new File("artikelliste.txt");
        try {
            BufferedReader br = new BufferedReader(
                new FileReader(f));

            try {
                String zeile = br.readLine();
                while (zeile != null) {
                    Artikel a = erzeugeArtikel(zeile);
                    artikeldaten.put(a.liefereArtikelnummer(), a);
                    zeile = br.readLine();
                }
            } finally {
                br.close();
            }
        } catch (FileNotFoundException e) {
            System.err.println("Die Datei " + f.getName()
                + " mit den Artikelnoten "
                + "konnte nicht gefunden werden.");
        } catch (IOException e) {
            System.err.println("Fehler beim Einlesen der "
                + "Artikelnoten.");
        }
    }

    private Artikel erzeugeArtikel(String zeile) {
        // ...
    }
}
```

In der Methode `erzeugeArtikel()` verwenden wir die Klasse `Scanner`, um die einzelnen Werte in der Zeile zu identifizieren.

```
private Artikel erzeugeArtikel(String zeile) {
    Scanner sc = new Scanner(new StringReader(zeile));
    long nummer = sc.nextLong();
    double preis = sc.nextDouble();
    double mwst = sc.nextDouble();
    String beschreibung = "";
    if (sc.hasNext()) {
        beschreibung = sc.next();
    }
}
```

```

        // falls die Beschreibung Leerzeichen enthaelt
        // alle Zeichenketten bis zum Ende der Zeile einlesen
        while (sc.hasNext()) {
            beschreibung += " " + sc.next();
        }
        return new Artikel(nummer, beschreibung, preis, mwst);
    }
}

```

Die Klasse `Scanner` berücksichtigt die Systemsprache und erwartet deswegen bei einem deutschen System für `double`-Werte das Komma als Trennzeichen. Eine mögliche Zeile der Datei `artikelliste.txt` sähe demnach wie folgt aus:

```
123      12,0      0,19      Blauer Blumentopf
```

Ein weiteres Problem tritt bei der eindeutigen Nummerierung der Rechnungen auf. Mit unserer bisherigen Implementierung beginnt die Zählung der Nummern wieder von vorne, sobald unsere Anwendung beendet und wieder neu gestartet wird. Um dies zu vermeiden, speichern wir die nächste Rechnungsnummer vor dem Beenden der Anwendung in einer Datei `rechnungsnummer.txt` und lesen beim nächsten Start diese Nummer wieder aus.

41.3 Erzeugung einer eigenständigen Anwendung

Zum Abschluss wollen wir die eigentliche Kassenanwendung näher betrachten. Eine eigenständige Anwendung benötigt immer eine `main()`-Methode. In unserem Fall versehen wir die Anwendung mit einem privaten Konstruktor, der nur von der `main()`-Methode aufgerufen wird. Im Konstruktor erzeugen wir ein `BufferedReader`-Objekt, um so Benutzereingaben einlesen zu können.

```

public class Kassenanwendung {
    private BufferedReader br;
    private Map<Long, Artikel> artikeldaten;
    // ...

    private Kassenanwendung() {
        br = new BufferedReader(new InputStreamReader(System.in));
    }

    public static void main(String[] args) {
        new Kassenanwendung().ausfuehren();
    }

    // ...
}

```

In der Methode `ausfuehren()` lesen wir zunächst die Artikeldaten ein, initialisieren die Rechnungsnummer und starten anschließend das Kassenmenü. Wird dieses später beendet, so speichern wir noch die Rechnungsnummer für die nächste Ausführung. Da bei diesen Schritten auch Ausnahmesituationen auftreten können, gibt es einen eigenen Ausnahmetyp `InitException`. Die Methode `leseArtikelEin()` würde deshalb Ausnahmen werfen, statt direkt die Fehlermeldungen auszugeben.

```
private void leseArtikelEin() throws InitException {
    this.artikeldaten = new HashMap<Long, Artikel>();
    File f = new File("artikelliste.txt");
    try {
        BufferedReader br = new BufferedReader(
            new FileReader(f));

        try {
            String zeile = br.readLine();
            while (zeile != null) {
                Artikel a = erzeugeArtikel(zeile);
                artikeldaten.put(a.liefereArtikelnummer(), a);
                zeile = br.readLine();
            }
        } finally {
            br.close();
        }
    } catch (FileNotFoundException e) {
        throw new InitException("Die Datei " + artikeldatei
            + " mit den Artikeldaten konnte nicht "
            + "gefunden werden.");
    } catch (IOException e) {
        throw new InitException("Fehler beim Einlesen der "
            + "Artikeldaten.");
    }
}
```

Die Methode `ausfuehren()` würde diese Ausnahmen entsprechend abfangen und behandeln.

```
private void ausfuehren() {
    try {
        leseArtikelEin();
    } catch (InitException e) {
        System.err.println("Fehler beim Laden der "
            + "Artikeldaten:");
        System.err.println(e.getMessage());
        System.err.println("Das Programm wird beendet");
        return;
    }
}
```

```
try {
    initialisiereRechnungsnummer();
} catch (InitException e) {
    System.err.println("Fehler bei der Initialisierung "
        + "der Rechnungsnummer:");
    System.err.println(e.getMessage());
}
starteKassenmenu();
try {
    speichereRechnungsnummer();
} catch (InitException e) {
    System.err.println("Fehler beim Abspeichern der "
        + "Rechnungsnummer:");
    System.err.println(e.getMessage());
}
}
```

Dieser Mechanismus wird auch an anderen Stellen, zum Beispiel bei ungültigen Benutzereingaben verwendet.

Wir sehen, dass wir unsere bisherigen Klassen nahezu unverändert übernehmen können und lediglich die Benutzungsschnittstelle und Datenspeicherung ergänzen mussten.

42 Zusammenfassung

Die **Java-API** bietet viele verschiedene Klassen für verschiedenste Zwecke an. Im Rahmen der näheren Betrachtung lernten wir unter anderem **Hüllklassen** für primitive Datentypen kennen.

Das Paket `java.io` bietet Klassen zum **Einlesen und Ausgeben** von Daten an. Das zentrale Konzept sind dabei die Datenströme, die als Kanäle für den Transport von Daten gesehen werden können. Solche Kanäle können auch hintereinander gehängt werden.

Java bietet drei **Standardströme** an, einen zum Einlesen von Tastatureingaben, einen zur Ausgabe regulärer Meldungen und einen für die Ausgabe von Fehlermeldungen.

Zum Einlesen lernten wir die Klassen `BufferedReader` und `Scanner` kennen.

Um Klassen allgemeingültiger formulieren zu können, gibt es in Java **generische Typen**. Diese ermöglichen es **Typparameter** einzufügen. So können wir beispielsweise Datenstrukturen unabhängig vom Typ der gespeicherten Elemente definieren. Typparameter werden in Java in spitzen Klammern hinter dem Typnamen angegeben. Bei der Verwendung eines solchen Typs muss der Typparameter durch einen konkreten Objekttyp ersetzt werden. Solche generischen Klassen für die Verwaltung einzelner Elemente finden sich zum Beispiel im Paket `java.util`.

Zum Durchlaufen aller Elemente beliebiger Sammlungen gibt es das Konzept des **Iterators**. Iteratoren werden in Java durch Objekte des generischen Typs `Iterator` repräsentiert. Alle Objekte vom Typ `Iterable` bieten einen Iterator an. Zusätzlich zur direkten Verwendung des Iterators können die Elemente solcher Objekte auch mit Hilfe der erweiterten `for`-Schleife durchlaufen werden.

In unserer Fallstudie beschäftigten wir uns noch einmal mit der Verwendung generischer Datenstrukturen und des Dateisystems. Auch den Einsatz von selbst definierten Ausnahmetypen zur Behandlung unerwarteter Ereignisse, zum Beispiel ungültiger Benutzereingaben, betrachteten wir genauer.

Lösungen zu Selbsttestaufgaben der Kurseinheit

Lösung zu Selbsttestaufgabe 38-1:

Wir verwenden die Klasse `java.util.GregorianCalendar`, die die abstrakte Klasse `java.util.Calendar` spezialisiert und eine Methode `isLeapYear()` anbietet.

```
public void printLeapYears(int fromYear, int toYear) {
    GregorianCalendar calendar = new GregorianCalendar();
    for (int year = fromYear; year <= toYear; year++) {
        if (calendar.isLeapYear(year)) {
            System.out.println(year);
        }
    }
}
```

Lösung zu Selbsttestaufgabe 38-2:

Die Methode

```
public static boolean isLowerCase(char ch)
```

der Klasse `java.lang.Character` leistet das Gewünschte.

Lösung zu Selbsttestaufgabe 38-3:

```
public double radius(double flaeche) {
    return Math.sqrt(flaeche / Math.PI);
}
```

Lösung zu Selbsttestaufgabe 39-1:

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;

public class Rechner {

    private String dateiname;

    public Rechner(String filename) {
        this.dateiname = filename;
    }
}
```

```
public long addiere() {
    long summe = 0;
    try {
        BufferedReader br = new BufferedReader(
            new FileReader(this.dateiname));
        try {
            String line = br.readLine();
            while (line != null) {
                summe += Long.parseLong(line);
                line = br.readLine();
            }
        } finally {
            br.close();
        }
    }
    catch (FileNotFoundException e) {
        System.out.println("Datei wurde nicht gefunden.");
    } catch (IOException e) {
        System.out.println("Datei kann nicht gelesen werden.");
    } catch (NumberFormatException e) {
        System.out.println("Datei enthält unzulässige "
            + "Zeichen.");
    }
    return summe;
}
```

Lösung zu Selbsttestaufgabe 39-2:

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class ScannerRechner {

    private String dateiname;

    public ScannerRechner(String filename) {
        this.dateiname = filename;
    }

    public long addiere() {
        long summe = 0;
        try {
            Scanner sc = new Scanner(new File(this.dateiname));
            while (sc.hasNextLong()) {
                summe += sc.nextLong();
            }
        } catch (FileNotFoundException e) {
            System.out.println("Datei wurde nicht gefunden.");
        }
    }
}
```

```

        return summe;
    }
}

```

Lösung zu Selbsttestaufgabe 39-3:

```

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.PrintWriter;
import java.io.FileNotFoundException;
import java.io.IOException;

public class Filter {

    private String dateinameIn;
    private String dateinameOut;

    public Filter(String infile, String outfile) {
        this.dateinameIn = infile;
        this.dateinameOut = outfile;
    }

    public void filter() {
        try {
            BufferedReader br = new BufferedReader(
                new FileReader(this.dateinameIn));
            PrintWriter pw = new PrintWriter(this.dateinameOut);
            try {
                String line = br.readLine();
                while (line != null) {
                    if (line.length() > 0
                        && Character.isUpperCase(
                            line.charAt(0))) {
                        pw.println(line);
                    }
                    line = br.readLine();
                }
            } finally {
                br.close();
                pw.flush();
                pw.close();
            }
        } catch (FileNotFoundException e) {
            System.out.println("Datei wurde nicht gefunden.");
        } catch (IOException e) {
            System.out.println("Datei kann nicht gelesen werden.");
        }
    }
}

```

Lösung zu Selbsttestaufgabe 40.2-1:

```

void findeKuerzesteUndLaengsteZeichenkette(List<String> l) {
    if (l == null || l.isEmpty()) {
        return;
    }
    String min = l.get(0);
    String max = l.get(0);
    for (String s : l) {
        if (s.length() < min.length()) {
            min = s;
        }
        if (s.length() > max.length()) {
            max = s;
        }
    }
    System.out.println(min);
    System.out.println(max);
}

```

Lösung zu Selbsttestaufgabe 40.3-1:

```

class Bubblesorter<T extends Comparable<T>> {
    public void sort(T[] feld) {
        // es werden maximal feld.length - 1 Durchläufe benötigt
        for (int i = 0; i < feld.length - 1; i++) {
            // solange keine Vertauschungen vorgenommen werden
            // ist das Feld sortiert
            boolean sorted = true;
            // Durchlaufe das Feld, in jedem Durchlauf muss
            // ein Element weniger berücksichtigt werden
            for (int j = 0; j < feld.length - 1 - i; j++) {
                if (feld[j].compareTo(feld[j + 1]) > 0) {
                    // wenn linkes größer
                    // dann vertausche
                    T temp = feld[j];
                    feld[j] = feld[j + 1];
                    feld[j + 1] = temp;
                    // Feld ist nicht sortiert
                    sorted = false;
                }
            }
            if (sorted) {
                // keine Vertauschungen, Feld ist
                // folglich vollständig sortiert
                break;
            }
        }
    }
}

```

Lösung zu Selbsttestaufgabe 40.4-1:

```

public interface Paar<S,W> {
    /**
     * liefert den gespeicherten Schluessel
     * @return den gespeicherten Schluessel
     */
    public S liefereSchluessel();
    /**
     * liefert den gespeicherten Wert
     * @return den gespeicherten Wert
     */
    public W liefereWert();
    /**
     * speichert einen neuen Schluessel
     * @param schluessel der neue Schluessel
     */
    public void setzeSchluessel(S schluessel);
    /**
     * speichert einen neuen Wert
     * @param wert der neue Wert
     */
    public void setzeWert(W wert);
    /**
     * vergleiche das gegebene Paar mit diesem
     * @param p das zu vergleichende Paar
     * @return true, wenn sowohl Schlüssel als auch Wert gleich
     * sind, d.h. equals() true liefert, sonst false
     */
    public boolean istGleich(Paar<S,W> p);
}

```

Lösung zu Selbsttestaufgabe 40.4-2:

```

public class PaarImpl<S,W> implements Paar<S,W> {
    private S schluessel;
    private W wert;

    public PaarImpl(S schluessel, W wert) {
        this.setzeSchluessel(schluessel);
        this.setzeWert(wert);
    }

    public S liefereSchluessel() {
        return this.schluessel;
    }

    public W liefereWert() {
        return this.wert;
    }
}

```

```
public void setzeSchluessel(S schluessel) {
    this.schluessel = schluessel;
}

public void setzeWert(W wert) {
    this.wert = wert;
}

public boolean istGleich(Paar<S,W> p) {
    return this.istSchluesselGleich(p.liefereSchluessel())
        && this.istWertGleich(p.liefereWert());
}

private boolean istSchluesselGleich(S andererSchluessel) {
    if (schluessel == null) {
        return andererSchluessel == null;
    }
    return schluessel.equals(andererSchluessel);
}

private boolean istWertGleich(W andererWert) {
    if (wert == null) {
        return andererWert == null;
    }
    return wert.equals(andererWert);
}
}
```

Lösung zu Selbsttestaufgabe 40.4-3:

Damit die Elemente direkt beim Einfügen sortiert werden können, muss der Typparameter E die folgende Einschränkung `E extends Comparable<E>` erfüllen. Um eine generische Listenklasse zu erhalten, müssen wir zunächst die Klasse `ListNode` anpassen.

```
public class ListNode<T> {
    private T entry;
    private ListNode<T> next;

    public ListNode(T value) {
        this(value, null);
    }

    public ListNode(T value, ListNode<T> nextNode) {
        this.entry = value;
        this.next = nextNode;
    }

    public void setEntry(T value) {
        this.entry = value;
    }
}
```

```

public void setNext(ListNode<T> nextNode) {
    this.next = nextNode;
}

public T getEntry() {
    return this.entry;
}

public ListNode<T> getNext() {
    return this.next;
}
}

```

Anschließend können wir die Klasse `LinkedList` oder `SortedList` ebenfalls anpassen. Exemplarisch ist nachfolgend nur die Klasse `SortedList` gezeigt.

```

public class SortedList<E extends Comparable<E>> {
    private ListNode<E> head;

    public SortedList() {
        this.head = null;
    }

    public void add(E value) {
        this.head = add(this.head, value);
    }

    private ListNode<E> add(ListNode<E> node, E value) {
        if (node == null) {
            return new ListNode<E>(value, node);
        }
        if (node.getEntry().compareTo(value) > 0) {
            return new ListNode<E>(value, node);
        }
        node.setNext(add(node.getNext(), value));
        return node;
    }
}

```