- Java Skriptum 3 -

Java 3.Jahrgang

Verfasser: Mag. Otto Reichel
HTBLuVA St.Pölten
Abteilung Informatik

Aktuelle Version: 20. April 2016

Inhaltsverzeichnis

Kapitel 1

Grundlegende Klassen

1.1 Die Klasse `java.lang.Object`

Die Klasse `java.lang.Object` ist die Wurzel der Klassenhierarchie. Jede Klasse erbt von dieser Klasse. Alle Javaobjekte (auch Felder) haben die in `Object` definierten Eigenschaften und sind daher vom diesem Typ.
Drei für die Threadkontrolle wichtige finale Methoden in `java.lang.Object` sind `wait()`, `notify()` und `notifyAll()`. Diese werden im Abschnitt über Threads besprochen.
Zwei andere Methoden, `equals()` und `toString()` stellen allgemeine Funktionalität zur Verfügung und können in abgeleiteten Klassen überschrieben werden.

1.1.1 Die Methode `toString()`

`public String toString()`
Liefert die Stringdarstellung eines Objekts. Die Methode wird implizit immer dann aufgerufen, wenn diese Stringdarstellung benötigt wird. Die Methode `toString()` aus `java.lang.Object` hat den folgenden Sourcecode:

```
1 public String toString() {
2   return getClass().getName()+"@"+Integer.toHexString(hashCode());
3 }
```

Listing 1.1: toString() aus java.lang.Object

Das folgende Beispiel demonstriert implizite und explizite Aufrufe von `toString()`

```
1  public class ToString {
2    public static void main(String []args) {
3      Test t = new Test();
4      int []f = new int[100];
5      System.out.println("Test: " + t);      // impliziter Aufruf
6      t = null;
7      System.out.println("Nullreferenz: " + t);
8      System.out.println("Feld f: " + f.toString());
9    }
10 }
11
12 class Test {}
```

Listing 1.2: Beispiel zu toString()

Ausgabe:

```
Test: Test@1ea2dfe
Nullreferenz: null
Feld f: [I@17182c1
```

1.1.2 Die Methode equals()

Vergleicht man 2 Referenzen mit dem Vergleichsoperator ==, so erhält man genau dann true, wenn die beiden Referenzen ein und dasselbe Objekt referenzieren. Zum Vergleich von Inhalten ist die Methode equals() vorgesehen.

public boolean equals(Object obj)
Die Methode equals() aus java.lang.Object stellt genau die Funktionalität des Vergleichsoperators == zur Verfügung. Ihr Quellcode lautet:

```
1 public boolean equals(Object obj) {
2   return this == obj;
3 }
```

Listing 1.3: equals() aus java.lang.Object

In abgeleiteten Klassen kann equals() so überschrieben werden, dass auch Inhalte verglichen werden. Dabei ist zunächst zu prüfen, ob des übergebene Objekt auch vom Typ des this-Objektes ist, andernfalls ist false zu retournieren.

Das nächste Beispiel zeigt das Überschreiben der Methode equals() an Hand eines einfachen Beispiels:

```
1  class Test {
2    private int a, b;
3
4    public Test(int a, int b) {
5      this.a = a;
6      this.b = b;
7    }
8
9   @Override
10   public boolean equals(Object o) {
11     if(o == null) {
12       return false;
13     }
14     if(this == o) {       // gleiches Objekt, nichts zu vergleichen
15       return true;
16     }
17     if(o.getClass() == this.getClass()) {
18       return a == ((Test)o).a && b == ((Test)o).b;
19     }
20     else {
21       return false;
22     }
23   }
24 }
25
26 public class Equals {
27   public static void main(String []args) {
28     Test t1 = new Test(3,10);
29     Test t2 = new Test(3,10);
30     System.out.println("t1.equals(t2) liefert: " + t1.equals(t2));
31     System.out.println("t1 == t2      liefert: " + (t1 == t2));
32     t2 = new Test(4,10);
33     System.out.println("t1.equals(t2) liefert: " + t1.equals(t2));
34     t2 = t1;
35     System.out.println("t1 == t2      liefert: " + (t1 == t2));
36   }
37 }
```

Listing 1.4: Ueberschreiben von equals()

Ausgabe:

```
t1.equals(t2) liefert: true
t1 == t2      liefert: false
t1.equals(t2) liefert: false
t1 == t2      liefert: true
```

1.2 Die Klasse `java.util.Objects`

Die Instanzmethoden der Klasse `java.lang.Object` haben einen entscheidendne Nachteil. So erzeugt z.B. der Codeabschnitt

```
1 String s = null;
2 if(s.equals("quit")) {
3   // DO SOMETHING
4 }
```

Listing 1.5: NullPointerException mit equals()

eine `NullPointerException`. Dies kann zwar durch den Aufruf `"quit".equals(s)` vermieden werden, beim Vergleich von 2 Stringreferenzen wie z.B. s1.equals(s2) ist das Problem aber nicht so einfach zu lösen. Hier müsste man vor dem Vergleich zumindest `s1` auf `null` überprüfen. Zur einfacheren Handhabung socher Situationen gibt es seit Java 7 die Klasse `java.util.Objects`. Diese Klasse implementiert die wichtigstens Methoden von java.lang.Object mit Hilfe von statischen Methoden und ist daher gegen null-Referenzen abgsichert. Die wichtigsten Methoden sind:

`public static boolean equals(Object a, Object b)`
Vergleicht die Objekte `a` und `b` und liefert `false`, wenn genau eines der beiden Argumente `null` ist, `true` wenn sowohl `a` als auch `b` gleich `null` sind. Sonst wird das Ergebnis von `a.equals(b)` retourniert.

`public static String toString(Object o)`
Liefert `o.toString()` wenn `o` ungleich `null` ist und `null`, wenn `o` gleich `null` ist.
Javadoc: java.util.Objects

1.3 Die Klasse `java.lang.String`

In Java werden Zeichenketten durch die Klasse `java.lang.String` repräsentiert. Sie bietet Methoden zum Erzeugen von Zeichenketten, zur Extraktion von Teilstrings, zum Vergleich mit anderen Strings sowie zur Erzeugung von Strings aus primitiven Typen. Eine Instanz der Klasse String ist prinzipiell eine konstante Kette von Unicode-Zeichen (Feld vom Typ `char`). Nach der Initialisierung eines Strings bleiben Länge und Inhalt konstant. Stringobjekte sind also immutable. Zur Verarbeitung von veränderlichen Zeichenketten existieren in Java die Klassen `java.lang.StringBuffer` und `java.lang.StringBuilder`. Alle drei Klassen sind final, d.h. von ihnen können keine weiteren Klassen abgeleitet werden.

- Die Klasse String ist final
- Stringobjekte sind immutable

1.3.1 Wichtige Methoden der Klasse `java.lang.String`

Konstruktoren

`public String()`
Erzeugt ein leeres Stringobjekt `""`.

`public String(String value)`
Erzeugt einen neuen String durch Duplizierung von `value`.

`public String(char[] value)`
Erzeugt einen neuen String aus dem Charakterfeld `value`.

`public String(byte[] value)`
Erzeugt einen neuen String aus dem Bytefeld `value`.

Allgemeine Methoden

`public int length()`
Liefert die Länge dieses Stringobjekts.

`public String concat(String s)`
Liefert einen neuen String, der entsteht, wenn man den String `s` an das `this`-Objekt anhängt.

Zeichen- und Stringextraktion

`public char charAt(int index)`
Liefert das Zeichen an der nullbasierten Position index. Bei falschem Index (kleiner als 0 oder `>= this.length()` wird die unchecked `IndexOutOfBoundsException` geworfen.

`public String substring(int begin, int end)`
Liefert den Teilstring, der an der Position `begin` startet und an der Position `end-1` endet. Bei falschem Index wird die unchecked `IndexOutOfBoundsException` geworfen.

`public String substring(int begin)`
Liefert den Teilstring von der Position `begin` bis zum Ende des Strings. Bei falschem Index wird die unchecked `IndexOutOfBoundsException` geworfen.

`public String trim()`
Liefert jenen String, der entsteht, wenn am Beginn und Ende des `this`-Objekts alle Zeichen mit Unicode kleiner gleich 32 (`'\u0020'`) entfernt werden. Es werden speziell also alle führenden und abschließenden Leerzeichen entfernt.

`public String[] split(String regex)`
Zerlegt das `this`-Objekt in Teilstrings, wobei die Begrenzer mit dem gegebenen regulären Ausdruck `regex` matchen. Geliefert wird ein Stringfeld, das alle Teilstrings enthält.
Beispiele:
`"boo:and:foo".split(":");` liefert `{ "boo", "and", "foo" }`
`"boo:and:foo".split("o");` liefert `{ "b", "", ":and:f" }`

Vergleich von Zeichenketten

`public boolean equals(Object o)`
Überlagert die Methode `equals()` aus `java.lang.Object`. Prüft das `this`-Objekt und `o` auf inhaltliche Übereinstimmung. Sollte `o` nicht vom Typ `String` sein, wird auf jeden Fall `false` geliefert.

`public boolean equalsIgnoreCase(String s)`
Vergleicht den String `s` mit dem `this`-Objekt und ignoriert Unterschiede in der Groß-Kleinschreibung.

`public int compareTo(String s)`
Vergleicht das `this`-Objekt alphabetisch mit `s`. Retourniert bei `this < s` einen negativen Wert, bei Gleichheit den Wert 0 und bei `this > s` einen positiven Wert.

`public boolean startsWith(String s)`
Testet, ob das `this`-Objekt mit der Zeichenkette `s` beginnt.

`public boolean endsWith(String s)`
Testet, ob das `this`-Objekt mit der Zeichenkette `s` endet.

Suchen in Zeichenketten

`public int indexOf(char ch)`
Sucht das Zeichen `ch` im `this`-Objekt und liefert die Position der ersten Übereinstimmung bzw. -1, wenn `ch` im String nicht vorkommt.

`public int indexOf(String str, int fromIndex)`
Sucht das Zeichen `ch` im `this`-Objekt ab der Position `fromIndex` und liefert die Position der nächsten Übereinstimmung bzw. -1, wenn `ch` im String nicht mehr vorkommt.

`public int lastIndexOf(char ch)`
Wie `indexOf()`, liefert aber die Position der letzten Übereinstimmung.

Ersetzen von Zeichenketten

`public String toLowerCase()`
Liefert einen neuen String, der entsteht, wenn im `this`-Objekt alle Großbuchstaben durch Kleinbuchstaben ersetzt werden.

`public String toUpperCase()`
Liefert einen neuen String, der entsteht, wenn im `this`-Objekt alle Kleinbuchstaben durch Großbuchstaben ersetzt werden.

`public String replace(char oldchar, char newchar)`
Liefert einen neuen String, der entsteht, wenn im `this`-Objekt jedes Zeichen `oldchar` durch das Zeichen `newchar` ersetzt wird.

Keine dieser Methoden kann das `this`-Objekt verändern, da ein einmal erzeugtes Stringobjekt immutable ist. Methoden, die einen String liefern, erzeugen intern ein neues Stringobjekt und liefern eine Referenz auf dieses neue Objekt. Ist das Ergebnis inhaltlich mit dem `this`-Objekt identisch, so wird kein neues Objekt erzeugt, sondern die `this`-Referenz retourniert. Das folgende Beispiel demonstriert diesen Sachverhalt:

```
1  public class ImmutableStrings {
2    public static void main(String []args) {
3      String s = new String("   100 + 100 = 200");
4
5      System.out.println(s.trim()); // Veraendert s nicht
6      System.out.println(s);
7      s.replace('0', 'x');          // Veraendert s nicht
8      System.out.println(s);
9      s = s.replace('0', 'x');      // neues Objekt in s gespeichert
10     System.out.println(s);
11   }
12 }
```

Listing 1.6: Immutable-Eigenschaft von Strings

Ausgabe:

```
100 + 100 = 200
   100 + 100 = 200
   100 + 100 = 200
   1xx + 1xx = 2xx
```

1.3.2 Stringpool, Vergleichen von Strings

Beim Vergleichen von Strings gelten die gleichen Regeln wie beim Vergleich beliebiger Javaobjekte (`s1` und `s2` seien Stringreferenzen):

- `s1 == s2` liefert genau dann `true`, wenn `s1` und `s2` das gleiche Objekt referenzieren.
- `s1.equals(s2)` liefert genau dann `true`, wenn die über `s1` und `s2` gespeicherten Strings gleichen Inhalt haben.

Alle Stringkonstanten innerhalb eines Javaprogrammes bilden den sog. Stringpool. Dieser enthält nur Stringobjekte mit unterschiedlichem Inhalt. Das bedeutet, dass bei mehrfachem Vorkommen einer Stringkonstanten im Quellcode diese nur einmal im Stringpool liegt. Vergleiche von Stringkonstanten mit dem Operator `==` liefern bei gleichem Inhalt also `true`, da es sich um ein und dasselbe Objekt aus dem Stringpool handelt. Das folgende Beispiel demonstriert diesen Sachverhalt:

```
1  public class CompareStrings {
2    public static void main(String []args) {
3      String s1 = "Vergleichsstring";
4      String s2 = "Vergleichsstring";
5      // s1 und s2 verweisen auf den gleichen String im Stringpool
6      System.out.println(" 6: s1 == s2      liefert: " + (s1 == s2));
7      System.out.println(" 7: s1.equals(s2) liefert: " + s1.equals(s2));
8      s1 = new String("Vergleichsstring");
9      // nun wurde eine neues Stringobjekt erzeugt
10     System.out.println("10: s1 == s2      liefert: " + (s1 == s2));
11     System.out.println("11: s1.equals(s2) liefert: " + s1.equals(s2));
12   }
13 }
```

Listing 1.7: Vergleich von Stringobjekten

Ausgabe:

```
 6: s1 == s2      liefert: true
 7: s1.equals(s2) liefert: true
10: s1 == s2      liefert: false
11: s1.equals(s2) liefert: true
```

Jeder Stringkonstante aus dem Stringpool kann wie eine Referenz verwendet werden. Dies bedeutet, dass über Stringliterale Instanzmethoden der Klasse String aufgerufen werden können:

```
System.out.println("Java".charAt(2));
System.out.println("java".toUpperCase());
```

Ausgabe:

```
v
JAVA
```

1.3.3 Stringverkettung

Der Operator `+` ist für die Klasse String überladen. Ist im Ausdruck `a+b` wenigstens einer der beiden Operanden ein String, so wird auch der zweite in einen String konvertiert und die Strings werden verkettet. Java gewährleistet, dass es für alle Ausdrücke mit Ausnahme von `void` eine Stringdarstellung gibt. Für Objekte ist dies durch die Methode `toString()` gewährleistet, die Stringdarstellung der `null`-Referenz ist der String `"null"`. Das folgende Beispiel veranschaulicht den Sachverhalt:

```
1 System.out.println("abc" + 10 + 20);
2 System.out.println(10 + "abc" + 20);
3 System.out.println(10 + 20 + "abc");
4 System.out.println(10 + (20 + "abc"));
5 System.out.println("" + 10 + 20 + "abc");
```

Listing 1.8: Stringverkettung

Ausgabe:

```
abc1020
10abc20
30abc
1020abc
1020abc
```

1.4 Die Klassen StringBuffer und StringBuilder

Diese Klassen aus dem Paket `java.lang` dienen zur Verarbeitung von Zeichenketten, die sich dynamisch verändern können. Sie besitzen exakt die gleichen Methoden, wobei `StringBuilder`[1] im Gegensatz zu `StringBuffer` nicht threadsicher ist aber dafür in Singlethread-Umgebungen performanter arbeitet. Die Klassen besitzen nicht so viele Methoden zur Auswertung der Zeichenkette, sondern legen den Schwerpunkt auf Operationen zur Veränderung ihres Inhalts.

1.4.1 Wichtige Methoden

Konstruktoren

```
public StringBuffer()
public StringBuilder()
```
Erzeugt einen leeren `StringBuffer` bzw. `StringBuilder`.

```
public StringBuffer(String s)
public StringBuilder(String s)
```
Erzeugt einen `StringBuffer` bzw. `StringBuilder`, der eine Kopie des Strings `s` repräsentiert.

```
public StringBuffer(int capacity)
public StringBuilder(int capacity)
```
Erzeugt einen leeren `StringBuffer` bzw. `StringBuilder` mit der angegebenen Ausgangskapazität.

Einfügen von Elementen

```
public StringBuffer append(String s)
public StringBuilder append(String s)
```
Hängt den String `s` an das Ende des `this`-Objekts an. Zurückgegeben wird eine Referenz auf das auf diese Weise veränderte `this`-Objekt. Zusätzlich gibt es überladene `append()`-Methoden zum Anhängen aller Arten von primitiven Typen. Anstelle eines String-Objekts wird hier der entsprechende primitive Typ übergeben, in einen String konvertiert und an das Ende des `this`-Objekts angehängt.

```
public StringBuffer append(Object o)
public StringBuilder append(Object o)
```
Hängt die Stringdarstellung `o.toString()` an das Ende des `this`-Objektes an. Zurückgegeben wird eine Referenz auf das auf diese Weise veränderte `this`-Objekt.

```
public StringBuffer insert(int offset, String s)
public StringBuilder insert(int offset, String s)
```
Fügt den String `s` ab der Position `offset` in das `this`-Objekt ein.

Löschen von Elementen

```
public StringBuffer deleteCharAt(int index)
public StringBuilder deleteCharAt(int index)
```

[1] neu seit Java 1.5

Das an der Position `index` stehende Zeichen wird entfernt.

`public StringBuffer delete(int start, int end)`
`public StringBuilder delete(int start, int end)`
Entfernt jenen Teilstring, der von Position `start` bis Position `end-1` reicht.

Verändern von Elementen

`public void setCharAt(int index, char ch)`
Das an der Position `index` stehende Zeichen wird durch `ch` ersetzt.

`public StringBuffer replace(int start, int end, String s)`
`public StringBuilder replace(int start, int end, String s)`
Ersetzt das Teilstück von Position `start` bis Position `end-1` durch den String `s`.

`public StringBuffer reverse()`
`public StringBuilder reverse()`
Dreht die Reihenfolge der Zeichen im `this`-Objekt um und retourniert das `this`-Objekt.

Allgemeine Methoden

`public int length()`
Liefert die aktuelle Länge des `this`-Objekts.

`public String toString()`
Dient zur Konvertierung eines `StringBuffer`- bzw. `StringBuilder`-Objektes in einen String. Liefert den Inhalt des `this`-Objekts als String.

1.4.2 Eigenschaften

- Die Klassen `StringBuffer` und `StringBuilder` überschreiben die Methode `equals()` aus `java.lang.Object` nicht.
- Alle Methoden verändern das `this`-Objekt und schicken gegebenfalls eine Referenz auf dieses Objekt zurück.
- Die Klassen `StringBuffer` und `StringBuilder` sind final.

1.4.3 Beispiel

Das folgende Beispiel demonstriert einige der oben beschriebenen Methoden und zeigt, wie die Stringverkettung intern mit Hilfe der Klasse `StringBuffer` implementiert ist:

```
1  public class StringBufferExample {
2    public static void main(String []args) {
3      StringBuffer sbuf = new StringBuffer("12345");
4      sbuf.reverse();
5      System.out.println(sbuf);
6      sbuf.insert(2, "xxx");
7      System.out.println(sbuf);
8      sbuf.append("yyy");
9      System.out.println(sbuf);
10
11     String a = "Hallo", b = "Welt" , c;
12     //Konventionelle Verkettung
13     c = a + ", " + b;
14     System.out.println(c);
15     //Implementierung durch StringBuffer
```

```
16      c = new StringBuffer().append(a).append(", ").append(b).toString();
17      System.out.println(c);
18    }
19  }
```

Listing 1.9: Beispiel zu StringBuffer

Ausgabe:

```
54321
54xxx321
54xxx321yyy
Hallo, Welt
Hallo, Welt
```

1.5 Die Klasse `java.util.Scanner`

Ein Objekt dieses Typs dient dazu, einen Text in einzelne Token aufzuspalten, wobei die Trennstrings mit Hilfe von regulären Ausdrücken definiert werden können. Standardmäßig wird nach Whitespacezeichen getrennt. Die gelesenen Token können mit Hilfe der verschiedenen `nextXXX()`-Methoden in verschiedene Typen konvertiert werden.

1.5.1 Eingabe von `System.in`

Ein `Scanner`-Objekt kann auch verwendet werden, um Daten von `System.in`[2] zu lesen.

Das folgende Beispiel demonstriert das Lesen einer Zeile von `System.in`:

```
1  import java.util.*;
2  public class Tastatur {
3    public static void main(String[] args) {
4      Scanner sc = new Scanner(System.in);
5      System.out.print("Eingabe --> ");
6      String s = sc.nextLine();
7      System.out.println("Eingabe: " + s);
8    }
9  }
```

Listing 1.10: Scanner zum zeilenorientiertem Lesen von System.in

In obigem Beispiel werden die folgenden Methoden verwendet:

`public Scanner(InputStream source)`
Erzeugt einen Scanner zum Lesen von `source`[3].

`public String nextLine()`
Liest alle Zeichen bis zum nächsten Zeilentrennzeichen und liefert die gelesene Zeile als String.

Zum scannen primitiver Datentypen gibt es diverse `nextXXX()`-Methoden. Diese Methoden existieren für alle Datentypen, exemplarisch werden zwei vorgestellt:

`public int nextInt()`
`public double nextDouble()`
Scannen das nächste Token und verwandeln es in einen `int`- bzw. in einen `double`-Wert. Diese Methoden werfen die unchecked `InputMismatchException`, wenn das nächste Token nicht in den entsprechenden Typ verwandelt werden kann.

[2] also von der Konsole
[3] `System.in` ist vom Typ `InputStream`

Vor Aufruf dieser Methoden kann mit Hilfe entsprechender `hasNextXXX()`-Methoden geprüft werden, ob das nächste Token in den gewünschten Datentyp verwandelt werden kann:

```
public boolean hasNextInt()
public boolean hasNextDouble()
```

Diese Methoden liefern `true`, wenn das nächste Token in den entsprechenden Datentyp verwandelt werden kann.

Das nächste Beispiel liest solange Zeilen über die Tastatur, bis `"quit"` eingegeben wird. Alle Eingaben, die in einen Integer verwandelt werden können werden summiert und diese Summe wird am Ende ausgegeben.

```
1  import java.util.*;
2  public class Tastatur2 {
3    public static void main(String[] args) {
4      Scanner sc = new Scanner(System.in);
5      int akt = 0, summe = 0;
6      String ein = null;
7      while(!"quit".equals(ein)) {
8        if(sc.hasNextInt()) {
9          akt = sc.nextInt();
10         summe += akt;
11       }
12       else {
13         ein = sc.nextLine();
14       }
15     }
16     System.out.format("Die Summe aller Werte ist %d\n", summe);
17   }
18 }
```

Listing 1.11: Scanner zum Lesen von System.in 2

Das folgende Protokoll zeigt zwei Testläufe:

```
java Tastatur2
123 hallo
17
17
quit
Die Summe aller Werte ist 157

java Tastatur2
hallo 123
17
17
quit
Die Summe aller Werte ist 34
```

1.5.2 Lesen aus einer Textdatei

In diesem Abschnitt wird an Hand eines konkreten Beispiels demonstriert, wie mit Hilfe der Klasse `java.util.Scanner` aus einer Textdatei (CSV-Datei) gelesen werden kann. Die CSV-Datei speichert Länderinformationen und hat den folgenden Aufbau:

```
Kurzzeichen;Name;Hauptstadt;Flaeche;Einwohnerzahl
AU;Australia;Canberra;7686850;19731984
...
```

```
1  import java.util.Scanner;
2  import java.io.File;
3  import java.io.FileNotFoundException;
```

```
4
5  public class ScannerReadFile {
6    public static void main(String[] args) {
7      Scanner sc = null;
8      try {
9        sc = new Scanner(new File("countries.csv"));
10     }
11     catch(FileNotFoundException e) {
12       System.err.println("Datei counries.csv nicht gefunden");
13       System.exit(1);
14     }
15
16     String line = null;
17     if(sc.hasNextLine())
18       sc.nextLine();     // Kopfzeile lesen
19     while(sc.hasNextLine()) {
20       line = sc.nextLine();
21       Scanner sc1 = new Scanner(line).useDelimiter(";");
22       Country c = new Country(sc1.next(), sc1.next(), sc1.next(), sc1.nextLong(),
             sc1.nextLong());
23       System.out.println(c);
24     }
25   }
26 }
27
28 class Country {
29   private String kz;
30   private String name;
31   private String capital;
32   private long   flaeche;
33   private long   einwohner;
34
35   public Country(String kz, String name, String capital, long flaeche, long
         einwohner ) {
36     this.kz = kz;
37     this.name = name;
38     this.capital = capital;
39     this.flaeche = flaeche;
40     this.einwohner = einwohner;
41   }
42
43   public String toString() {
44     return String.format("%2s %-20s %-20s %10d %10d", this.kz, this.name, this.
           capital, this.flaeche, this.einwohner);
45   }
46 }
```

Listing 1.12: Scanner zum Lesen einer Textdatei

Ausgabe:

```
AU Australia          Canberra              7686850     19731984
BR Brazil             Brasilia              8511965    182032604
CN China              Beijing               9596960   1286975468
DE Germany            Berlin                 357021     82398326
ES Spain              Madrid                 504782     40217413
```

1.6 Die Wrapperklassen

Zu jedem der 8 primitive Datentypen gibt es im Paket `java.lang` eine zugehörige Wrapperklasse[4]. Jede Wrapperklasse kapselt dabei einen primitiven, nicht veränderbaren Wert. Wrapperobjekte sind

[4] Hüllenklasse

also immutable, d.h. sie lassen sich nach ihrer Konstruktion nicht mehr verändern. Alle Wrapperklassen sind final, d.h. es lassen sich keine Subklassen generieren. Die folgende Tabelle gibt zu jedem primitiven Typ den Namen der entsprechenden Wrapperklasse an:

Primitver Datentyp	Wrapperklasse
`boolean`	`java.lang.Boolean`
`char`	`java.lang.Character`
`byte`	`java.lang.Byte`
`short`	`java.lang.Short`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`float`	`java.lang.Float`
`double`	`java.lang.Double`

1.6.1 Konstruktion von Wrapperobjekten

Jede Wrapperklasse besitzt einen Konstruktor, der einen entsprechenden primitiven Typ erwartet. Außerdem besitzt mit Ausnahme der Klasse `Character` jede Wrapperklasse einen Konstruktor, der einen String erwartet. Kann der String in den entsprechenden Typ geparst werden, so wird ein gültiges Objekt erzeugt, sonst wirft der Konstruktor (außer bei `Boolean`) eine `NumberFormatException`. Der Konstruktor von `Boolean` erzeugt statt dessen ein Objekt, das den Wert `false` kapselt.

Konstruktoren der Klasse `java.lang.Boolean`
```
public Boolean(boolean value)
public Boolean(String s)
```

Konstruktor der Klasse `java.lang.Character`
```
public Character(char value)
```

Konstruktoren der Klasse `java.lang.Byte`
```
public Byte(byte value)
public Byte(String s) throws NumberFormatException
```

Konstruktoren der Klasse `java.lang.Short`
```
public Short(short value)
public Short(String s) throws NumberFormatException
```

Konstruktoren der Klasse `java.lang.Integer`
```
public Integer(int value)
public Integer(String s) throws NumberFormatException
```

Konstruktoren der Klasse `java.lang.Long`
```
public Long(long value)
public Long(String s) throws NumberFormatException
```

Konstruktoren der Klasse `java.lang.Float`
```
public Float(float value)
public Float(double value)
public Float(String s) throws NumberFormatException
```

Konstruktoren der Klasse `java.lang.Double`
```
public Double(double value)
public Double(String s) throws NumberFormatException
```

1.6.2 Vergleich von Wrapperobjekten

Alle Wrapperklassen überschreiben die Methode `equals()` aus `java.lang.Object`. Wrapperobjekte des gleichen Typs lassen sich also auf Inhaltsgleichheit prüfen:

```
1  public class Wrapper {
2    public static void main(String []args) {
3      Integer i1 = new Integer(10);
4      Integer i2 = new Integer("10");
5      Long l = new Long(10);
6
7      System.out.println("i1.equals(i2) liefert: " + i1.equals(i2));
8      System.out.println("i1 == i2      liefert: " + (i1 == i2));
9      System.out.println("i1.equals(l)  liefert: " + i1.equals(l));
10   }
11 }
```

Listing 1.13: Vergleich von Wrapperobjekten

Ausgabe:

```
i1.equals(i2) liefert: true
i1 == i2      liefert: false
i1.equals(l)  liefert: false
```

1.6.3 Auslesen der gespeicherten Werte

Alle nummerischen Wrapperklassen (`Byte`, `Short`, `Integer`, `Long`, `Float`,`Double`) erben von der abstrakten Klasse `java.lang.Number`. Diese ist wie folgt definiert:

```
public abstract class Number {
  public abstract byte byteValue();
  public abstract short shortValue();
  public abstract int intValue();
  public abstract long longValue();
  public abstract float floatValue();
  public abstract double doubleValue();
}
```

Damit beinhalten alle nummerischen Wrapperklassen diese oben definierten Methoden, wodurch der gewrappte Wert in jeden der 6 primitiven Datentypen verwandelt werden kann.

Kapitel 2

Ergänzungen und Neuerungen in Java 8

2.1 Innere Klassen

Klassen können neben Datenfeldern und Methoden auch wieder Klassendefinitionen beinhalten. In diesem Fall spricht man von inneren Klassen. Java kennt vier unterschiedliche Typen:

- Top-Level Nested Classes und Interfaces
- Nicht statische innere Klassen
- Lokale Klassen
- Anonyme Klassen

2.1.1 Top-Level Nested Classes und Interfaces

Dabei handelt es sich um statische innere Klassen oder um innere Interfaces. Top-Level Nested Classes und Interfaces können in beliebiger Tiefe verschachtelt werden, aber nur in anderen statischen Toplevelklassen und Interfaces. Sie dürfen jede beliebige Zugriffsart haben.
Statische Memberklassen (nested Top-Level Classes) haben wie andere statische Klassenmitglieder keine `this`-Referenz und daher nur Zugriff auf statische Mitglieder der umgebenden äußeren Klasse. Zum Erzeugen einer Instanz einer statischen inneren Klasse benötigt man auch keine Instanz der umgebenden äußeren Klasse. Es ist nur der voll qualifizierte Name zu verwenden.
Beispiel:

```
1 public class Outer {
2   protected static class Inner {}     // Nested Top-Level-Class
3
4   public static void main(String args[]) {
5     Inner i = new Outer.Inner();
6   }
7 }
```

Listing 2.1: Statische innere Klasse

Der Compiler erzeugt aus obiger Quelldatei die folgenden `class`-Dateien:

```
Outer.class
Outer$Inner.class $
```

2.1.2 Nicht statische innere Klassen

Nicht statische innere Klassen sind vergleichbar mit anderen nicht statischen Mitgliedern einer Klasse.

- Eine Instanz kann nur mit Hilfe einer existierenden Instanz der umgebenden äußeren Klasse erzeugt werden.
- Eine nicht statische innere Klasse darf selbst keine statischen Mitglieder haben.
- Methoden einer nicht statischen inneren Klasse haben direkten Zugriff auf alle Mitglieder aller umgebenden äußeren Klassen (auch auf private), wobei keine explizite Referenz notwendig ist. In einer inneren Klasse spricht man das zugehörige Objekt der umgebenden äußeren Klasse `Outer` über `Outer.this` an.
- Eine nicht statische innere Klasse darf jeden Zugriffsmodifier haben.

Beispiel:

```
1  class Outer {
2    private int i = 10;
3    static int k = 5;
4
5    protected class Inner {
6      int i = 30;
7      // static int l = 20; - Compilerfehler
8
9      public void printVars() {
10       System.out.println(Outer.this.i + " " + i + " " + k);
11     }
12   }
13
14   public Inner makeInstanceInner() {
15     return new Inner();       // return this.new Inner();
16   }
17 }
18
19 class Main {
20   public static void main(String []args) {
21     Outer outRef = new Outer();
22     Outer.Inner inRef1 = outRef.makeInstanceInner();
23     inRef1.printVars();
24     Outer.Inner inRef2 = new Outer().new Inner();
25     inRef2.printVars();
26   }
27 }
```

Listing 2.2: Nicht statische innere Klasse

Ausgabe:

```
10 30 5
10 30 5
```

2.1.3 Lokale Klassen

Lokale Klassen sind innere Klassen, die innerhalb einer Methode oder eines Blocks der umgebenden Klasse definiert sind. Solche Klassen sind lokal bezüglich der Methode oder des Blocks und haben daher keinen Zugriffsmodifier und können auch nicht explizit als `static` definiert werden. Sie sind aber implizit `static`, wenn sie sich in einer statischen Methode befinden. In einer lokalen inneren Klasse darf es keine statischen Mitglieder geben und es darf auch nur auf finale Datenfelder der umgebenden Methode zugegriffen werden. Seit Java 8 müssen solche Variable nicht mehr mit dem Schlüsselwort `final` definiert werden, sondern es reicht, wenn sie effektiv final sind, also im Code nicht verändert werden.
Lokale Klassen können nur in dem Block oder der Methode, in der sie definiert wurden, instanziiert werden. Die Methode kann aber die Instanz der lokalen Klasse retournieren. Der Rückgabetyp dieser Methode muss dann mit dem Typ der lokalen Klasse verträglich sein (z.B. der Typ einer Superklasse oder eines Interfaces).
Beispiel:

```
interface I {
  void foo();
}

class Outer {
  private int i = 10;
  static int j = 5;

  public I fooOuter(final int k) {  // hier ist k explizit final
                                    // final darf seit Java 8 auch weggelassen
                                        werden
    int l = 20;
    class Lokal implements I {
      public void foo() {
        i = 100;       // ok
        j = 105;       // ok
        // l = 115;    Compilerfehler
        System.out.println(i + " " + j + " " + k);
      }
    }
    return new Lokal();
  }
}

public class Main {
  public static void main(String []args) {
    I iRef = new Outer().fooOuter(30);
    iRef.foo();
  }
}
```

Listing 2.3: Lokale innere Klasse

Ausgabe:
`100 105 30`

2.1.4 Anonyme innere Klassen

Anonyme innere Klassen sind immer lokal in einem Block oder in einer Methode. Es handelt sich dabei um Klassen ohne Namen, die genau einmal instanziiert werden können. Sie können entweder von einer Superklasse abgeleitet werden oder ein vorgegebenes Interface implementieren. Beides gleichzeitig verbietet die Syntax.
Klassendefinition und Instanziierung erfolgen genau an der Stelle, an der das Objekt benötigt wird. Lokale Klassen sind implizit `final`. Sie können nicht abstrakt sein, da auf jeden Fall genau ein Objekt instanziiert wird. Das folgende Beispiel erzeugt ein Objekt einer anonymen lokalen Klasse, die das Interface `EventHandler<ActionEvent>` implementiert. Dieses Inteface enthält genau eine abstrakte Methode `handle()`, die hier überschrieben werden muss.

```
btnInc.setOnAction(new EventHandler<ActionEvent>() {
  @Override
  public void handle(ActionEvent event) {
    // DO SOMETHING
  }
});
}
```

Listing 2.4: Anonyme innere Klasse

Die Definition und Instanziierung einer anonymen lokalen Klasse erfolgt also mit Hilfe der Syntax

```
new XXX() { /* Klassenkoerper */ };
```

wobei `XXX` entweder die Superklasse oder das zu implementierende Interface angibt.

Ist XXX eine Superklasse, so können dem Superklassenkonstruktor mit Hilfe der folgenden Syntax auch Argumente übergeben werden:

```
new <superclass-name> ([<argument list>]) { <class definition> };
```

Ist XXX ein Interface, so kann nur die folgende Syntax verwendet werden. Die anonyme Klasse erbt dabei implizit von `java.lang.Object`:

```
new <interface-name> () { <class definition> };
```

2.2 Autoboxing und Unboxing

Bis Java 1.4 waren primitive Typen und ihre Wrapperobjekte prinzipiell unverträglich und konnten nur durch geeignete Methoden ineinander übergeführt werden. Wollte man z.B. den in einem Objekt vom Typ `java.lang.Integer` gespeicherten Wert um 1 erhöhen, so war das nur mit folgenden Befehlen möglich:

```
Integer io = new Integer(8);
int ip = io.intValue();          // gewrappten Wert auslesen
ip++;                            // primitiven Wert erhoehen
io = new Integer(ip);            // neuen Wrapper erzeugen
```

Dies lässt sich auch kürzer mit Hilfe einer Zeile programmieren:

```
io = new Integer(io.intValue()+1);
```

In beiden Versionen wird zunächst der im Wrapperobjekt `io` gespeicherte Wert in einen primitiven Integer verwandelt, dieser wird dann inkrementiert und aus dem so um 1 erhöhten primitiven Wert wird ein neues Wrapperobjekt erzeugt. Seit Java 1.5 werden diese notwendigen Umwandlungen durch Autoboxing/Unboxing automatisch durchgeführt und brauchen daher nicht mehr explizit ausprogrammiert zu werden.

- **Autoboxing**
  Autoboxing bedeutet, dass aus dem Wert eines primitiven Datentyps automatisch ein entsprechendes Wrapperobjekt erzeugt wird.
- **Unboxing**
  Unboxing bedeutet, dass aus einem Wrapperobjekt automatisch ein primitiver Datentyp erzeugt wird.

Das oben angegebene Beispiel lässt sich seit Java 1.5 also wie folgt programmieren:

```
Integer io = new Integer(8);
io++;
```

Bemerkung:
Ein Nachteil dieser Technik ist, dass aus dem Code nicht mehr immer erkennbar ist, ob es sich bei einer Variablen um einen primitiven Typ oder um eine Referenz auf ein Wrapperobjekt handelt.

2.2.1 Autoboxing/Unboxing bei Funktionsparametern

Das folgende Beispiel zeigt, dass Autoboxing/Unboxing auch bei Funktionsaufrufen verwendet werden kann:

```
1 public class Autoboxing_1 {
2   public static void main(String[] args) {
3     int primitiveValue = 9;
4     Integer wrapper = new Integer(8);
5     autobox(primitiveValue);
6     unbox(wrapper);
7     wrapper = null;
8     unbox(wrapper);
```

```
9     // unbox(null);      -->    Compilerfehler
10  }
11  static void autobox(Integer val) {
12    System.out.println(val);
13  }
14
15  static void unbox(int val) {
16    System.out.println(val);
17  }
18 }
```

Listing 2.5: Autoboxing/Unboxing

Ausgabe:

```
9
8
Exception in thread "main" java.lang.NullPointerException
        at Autoboxing_1.main(Autoboxing_1.java:8)
```

Das Unboxing in Zeile 8 führt auf eine `NullPointerException`, da zu diesem Zeitpunkt der Referenz `wrapper` der Wert `null` zugewiesen ist. Der Versuch, die Methode direkt mit dem Wert `null` aufrufen (Zeile 9) führt auf einen Compilerfehler.

2.2.2 Autoboxing/Unboxing bei Funktionsüberladung

Um *legacy code*[1] robuster zu machen, wird beim Überladen von Funktionen das Widning dem Autoboxing vorgezogen. Außerdem gilt Widning vor Varargs:

- Widning vor Autoboxing
- Autoboxing vor Varargs

Das folgende Beispiel demonstriert diese Regeln:

```
1  public class Autoboxing_2 {
2
3    static void go1(Integer x) {
4      System.out.println("go1 Integer");
5    }
6    static void go1(long x) {
7      System.out.println("go1 long");
8    }
9
10   static void go2(int x, int y) {
11     System.out.println("go2 int,int");
12   }
13   static void go2(byte... x) {
14     System.out.println("go2 byte... ");
15   }
16   static void go2(Byte x, Byte y) {
17     System.out.println("go2 Byte, Byte");
18   }
19
20   public static void main(String [] args) {
21     int i = 5;
22     go1(i); // which go() will be invoked?
23
24     byte b = 5;
25     go2(b, b);
26   }
27 }
```

Listing 2.6: Overloading und Autoboxing

[1] Code vor Java 1.5

Ausgabe:

```
go1 long
go2 int,int
```

Da die einzelnen nummerischen Wrappertypen alle direkt von `java.lang.Number` erben findet in diesem Bereich kein Widning statt. Das bedeutet, dass z.B. die Methode

```
public static void foo(Long l) { }
```

nicht mit dem Aufruf

```
foo(2);
```

verwendet werden kann. Der Integer `2` kann mit Autoboxing zwar in ein `Integer`-Objekt gewrappt werden, dieser Typ wird wegen der oben beschriebenen Unverträglichkeit aber nicht in den Typ `Long` erweitert.

2.2.3 Autoboxing/Unboxing bei Vergleichen

Hier gelten die folgenden Regeln:

- Vergleicht man ein Wrapperobjekt und einen primitiven Typ mit den Operatoren `==` bzw. `!=` so findet immer Unboxing statt, d.h. der Wrappertyp wird vor dem Vergleich in einen primitiven Typ verwandelt. Der Vergleich von zwei Wrapperobjekten bzw. zwei primitiven Typen unterliegt keinen neuen Regeln.
- Die Operatoren `< <= > >=` können nun auch auf ein oder zwei Wrapperobjekte angewendet werden. Hier findet immer Unboxing statt.
- Die Vergleichsfunktion `equals()` kann nur über einen Wrappertyp und nicht über einen primitiven Typ aufgerufen werden, da in Java der Punktoperator nicht auf einen primitiven Typ angewendet werden kann. Wird der Methode `equals()` ein primitiver Typ übergeben, so findet wie oben beschrieben Autoboxing statt.

Die ersten beiden Vergleiche in obigem Beispiel demonstrieren diese Regeln:

```
1  public class Autoboxing_3 {
2    public static void main(String[] args) {
3      int primitiveValue = 9;
4      Long wrapper = new Long(9);
5      System.out.println("1: " + (primitiveValue == wrapper));
6      System.out.println("2: " + (primitiveValue < wrapper));
7      //Compilerfehler:
8      //System.out.println(primitiveValue.equals(wrapper));
9      System.out.println("3: " + wrapper.equals(primitiveValue));
10
11     Integer i1 = 100;
12     Integer i2 = 100;
13     System.out.println("4: " + (i1 == i2));
14     Integer j1 = 1000;
15     Integer j2 = 1000;
16     System.out.println("5: " + (j1 == j2));
17   }
18 }
```

Listing 2.7: Autoboxing/Unboxing bei Vergleichen

Ausgabe:

```
1: true
2: false
3: false
4: true
5: false
```

Die Ergebnisse der Vergleiche 4 und 5 sind auf den ersten Blick verblüffend. Hier werden jeweils zwei Wrapperobjekte verglichen, der Operator == liefert also genau dann `true`, wenn es sich um ein und dasselbe Objekt handelt. Um dieses Verhalten zu verstehen, muss man wissen, dass Wrapperobjekte für gewisse Wertebereiche von der VM automatisch erzeugt werden und beim Wrappen von Literalen primitiver Typen Verwendung immer auf diese automatisch erzeugten Objekte[2] zugegriffen wird. Die Wertebereiche der so generierten und damit in einer VM eindeutigen Wrapperobjekte sind:

- `Boolean`-Objekte für `true` und `false`
- Alle Objekte vom Typ `Byte`
- `Short`-Objekte im Bereich von -128 bis 127
- `Integer`-Objekte im Bereich von -128 bis 127
- `Charakter`-Objekte im Bereich von `\u0000` bis `\u007F`

2.3 Enums

2.3.1 Konzept

Bis Java 1.4 wurden Aufzählungen in Java hauptsächlich über Konstante definiert, denen einzelne Integerwerte zugewiesen wurden:

```
1 public class Marks1 {
2   public static final int EXCELLENT = 1;
3   public static final int GOOD = 2;
4   public static final int SATISFACTORY= 3;
5   public static final int SUFFICIENT = 4;
6   public static final int INSUFFICIENT= 5;
7 }
```

Diese Vorgangsweise ist nicht typsicher, da einem primitiven Integer neben den oben definierten Noten auch andere Werte zugewiesen werden können:

```
int mark1 = Marks1.GOOD;
int mark2 = 7;
```

In Java 1.5 wurden mit Enums typsichere Aufzählungen eingeführt. Diese arbeiten prinzipiell nach dem in folgendem Beispiel vorgestellten Konzept:

```
1  public class Marks2 {
2    private String markName;
3    public static final Marks2 EXCELLENT = new Marks2("sehr gut");
4    public static final Marks2 GOOD = new Marks2("gut");
5    public static final Marks2 SATISFACTORY= new Marks2("befriedigend");
6    public static final Marks2 SUFFICIENT = new Marks2("ausreichend");
7    public static final Marks2 INSUFFICIENT= new Marks2("mangelhaft");
8
9    private Marks2(String name) {
10     this.markName = name;
11   }
12
13   public String toString() {
14     return this.markName;
15   }
16 }
```

Listing 2.8: Konzept einer Enum

Die Typsicherheit ist durch den privaten Konstruktor (Zeilen 9-11) gewährleistet. Dadurch wird verhindert, dass (von außerhalb der Klasse `Marks2`) weitere Objekte erzeugt werden können.

[2] Wrapperpool

2.3.2 Definition einer Enum

In Java 1.5 wurde mit dem Aufzählungstyp `enum`[3] ein Konzept implementiert, das der oben vorgestellten Klasse `Marks2` entspricht. Die Syntax zur Erzeugung einer Enum lautet:

```
[<Modifier>] enum <Bezeichner>{<Wert1>, <Wert2>, ...};
```

Das folgende Beispiel definiert wieder Noten:

```
public enum Marks3 {EXCELLENT, GOOD, SATISFACTORY,
                    SUFFICIENT, INSUFFICIENT};
```

Im Folgenden werden die wichtigsten Eigenschaften von Enums zusammengestellt:

- `enum`-Aufzählungen sind Klassen
  Dadurch wird die Typsicherheit gewährleistet. Die Syntax zur Verwendung einer Enum entspricht jener einer Klasse. Ein Aufzählungstyp kann überall dort verwendet werden, wo ein Klassentyp erlaubt ist. Eine Enum ist dabei eine spezielle Klasse und wird auch zu einer `*.class`-Datei kompiliert. Sie darf statische Methoden enthalten, im Speziellen auch eine Startmethode `main()`.
- Enums erben von der Klasse `java.lang.Enum`
  Die Klasse `java.lang.Enum` gibt es seit Java 1.5. Sie ist selbst keine Aufzählung, definiert aber das Verhalten jeder Enum.
- Enums sind standardmäßig `final`. Damit wird verhindert, dass man Aufzählungen erweitern kann. Der Modifier `final` braucht (und darf nicht) angegeben werden.
- Enums haben standardmäßig einen privaten Konstruktor
  Damit ist es unmöglich, Objekte zu erzeugen, die nicht von der Klasse selbst zur Verfügung gestellt werden.
- Alle Aufzählungsobjekte besitzen die Modifier `public`, `static` und `final`.

Von der Klasse `java.lang.Enum` erben Enums die folgenden Eigenschaften:

- `public final boolean equals(Object other)`
  Dient zum Vergleich zweier Aufzählungsobjekte. Liefert genau dann `true`, wenn es sich um ein und dasselbe Objekt handelt.
- `implements java.lang.Comparable`
  Damit erbt jeder Aufzählungstyp die Methode
  `public final int compareTo(E o)`
  Mit dieser Methode lassen sich Aufzählungsobjekte auf kleiner, gleich und größer vergleichen. Die dabei verwendete natürliche Reihenfolge entspricht jener, die bei der Definition der einzelnen Objekte angegeben wurde.
- `implements java.io.Serializable`
  Jedes Aufzählungsobjekt ist also serialisierbar.
- `public String toString()`
  Diese Methode aus `java.lang.Enum` liefert den Namen eines Auszählungsobjektes. Zum Beispiel liefert der Aufruf `Marks3.GOOD.toString()` den String `"GOOD"`.
- `public static <T extends Enum<T>> T valueOf(String name)`
  Bietet die umgekehrte Funktionalität von `toString()`. Es wird das Aufzählungsobjekt zum angegebenen Namen `name` geliefert. Gibt es den angegebenen Namen nicht, so wird eine `IllegalArgumentExcep` geworfen. Der Aufruf `Marks3.valueOf("GOOD")` retourniert das Auszählungsobjekt `Marks3.GOOD`.
- `public final int ordinal()`
  Diese Methode liefert die nullbasierte Position des Aufzählungsobjektes in der Aufzählung.
- `public static <T extends Enum<T>> T[] values()` Diese Methode liefert alle Aufzählungsobjekte einer Aufzählung als Feld. Damit ist es möglich, über alle Aufzählungswerte zu iterieren.

[3] neues Schlüsselwort in 1.5

Das folgende Beispiel demonstriert die angegebenen Methoden an Hand der Enum `Marks3`:

```
public enum Marks3 {
  EXCELLENT, GOOD, SATISFACTORY, SUFFICIENT, INSUFFICIENT;

  public static void main(String[] args) {
    Marks3 m1 = Marks3.EXCELLENT;

    // equals()
    boolean b1, b2;
    b1 = m1.equals(Marks3.EXCELLENT);
    b2 = m1.equals(Marks3.GOOD);
    System.out.format("%b,%b\n", b1, b2);

    // compareTo()
    int i1, i2;
    i1 = Marks3.SATISFACTORY.compareTo(m1);
    i2 = Marks3.SATISFACTORY.compareTo(SUFFICIENT);
    System.out.format("%d,%d\n", i1, i2);

    // valueOf(), toString()
    m1 = Marks3.valueOf(Marks3.class, "GOOD");
    System.out.format("%s\n", m1.toString());

    // values(), ordinal()
    for(Marks3 akt : Marks3.values()) {
      System.out.format("%d: %-15s\n", akt.ordinal(),  akt);
    }
  }
}
```

Listing 2.9: Grundfunktionalität einer Enum

Ausgabe:

```
true,false
2,-1
GOOD
0: EXCELLENT
1: GOOD
2: SATISFACTORY
3: SUFFICIENT
4: INSUFFICIENT
```

2.3.3 Enums und Switch-Case

Die switch-case-Struktur wurde in Java 1.5 so erweitert, dass als Argumente auch Enum-Typen akzeptiert werden:

```
public String verbal(Marks3 m) {
  switch(m) {
    case EXCELLENT   : return "hervorragend";
    case GOOD        : return "immer noch super";
    case SATISFACTORY: return "durchschnittlich";
    case SUFFICIENT  : return "nicht so toll";
    case INSUFFICIENT: return "mal wieder was lernen";
    default:           return null;
  }
}
```

Listing 2.10: Enum in einer switch-case

2.4 Generics

Mit den seit Java 1.5 eingeführten Generics können Schablonen für Klassen und Methoden geschrieben werden, um ein und denselben Code für mehrere Datentypen verwenden zu können.
Das folgende Beispiel zeigt eine einfache generische Klasse `Box`, in die spezielle Objekte gespeichert werden können. Dabei wird bei der Klassendefinition in spitzen Klammern ein Typ-Stellvertreter angegeben. Dieser Typ-Stellvertreter wird bei der Instanziierung mit einem speziellen Typ überschrieben. Innerhalb der Klasse kann nun dieser Typ-Stellvertreter wie ein normaler Referenztyp verwendet werden:

```
1  class Box<T> {
2    private T val;
3
4    public Box() {
5    }
6
7    public Box(T val) {
8      this.val = val;
9    }
10
11   public void setValue(T val) {
12     this.val = val;
13   }
14
15   public T getValue() {
16     return val;
17   }
18 }
```

Listing 2.11: Einfache generische Klasse

In der folgenden lauffähigen Klasse wird die generische Klasse `Box` verwendet.

```
1  public class SimpleGeneric {
2    public static void main(String []args) {
3      Box objectBox = new Box();
4      objectBox.setValue("hallo");
5      String s = (String) objectBox.getValue();
6      System.out.println(s);
7
8      Box<String> stringBox = new Box<String>();
9      Box<Integer> intBox = new Box<Integer>();
10     stringBox.setValue(new String("hallo"));
11     intBox.setValue(12);  // Autoboxing
12     System.out.println(stringBox.getValue().toUpperCase());
13     System.out.println(intBox.getValue().toString());
14   }
15 }
```

Listing 2.12: Verwendung einer generischen Klasse

Ausgabe:

```
hallo
HALLO
12
```

- Versorgt man wie in Zeile 3 den Typ-Stellvertreter nicht, so erhält man eine nicht typsichere Instanz der Klasse `Box`, in der beliebige Javaobjekte gespeichert werden können. Der Typ-Stellvertreter wird implizit durch den Typ `Object` ersetzt. Beim Auslesen eines Objektes mit Hilfe der Methode `getValue()` erhält man nun den Rückgabetyp `Object` und muss das Ergebnis z.B. auf den Typ `String` casten, damit Stringmethoden angewendet werden können. Verwendet man eine generische Klasse ohne Versorgung des Typ-Stellvertreters durch einen speziellen Typ, so erzeugt der Compiler die Warnung:

```
SimpleGeneric.java uses unchecked or unsafe operations
Recompile with -Xlint:unchecked for details.
```

Aus Gründen der Aufwärtskompatibilität ist die Unterdrückung eines angegebenen Typ-Stellvertreters prinzipiell erlaubt. Die richtige Verwendung von Generics wird nur vom Compiler geprüft, das Laufzeitsystem kennt Generics nicht. Ignoriert man die oben angegebene Warnung, kann zur Laufzeit die korrekte Verwendung von Generics nicht garantiert werden.

- In Zeile 8 wird eine typsichere `Box`-Instanz erzeugt. In dieser Instanz können nur Objekte vom Typ `String` gespeichert werden. Jeder andere Versuch, z. B.
  `stringBox.setValue(new Double(3.8));`
  führt auf einen Compilerfehler.
- Die Methode `getValue()` des typsicheren Objekts `stringBox` liefert den Typ `String`, wodurch die erhaltene Referenz direkt (d.h. ohne Cast) Stringmethoden aufrufen darf.

Die Vorteile bei der Verwendung generischer Klassen sind also:

1. Bessere Wiederverwertbarkeit durch generische Programmierung
2. Die Möglichkeit, Containerobjekte typsicher zu machen
3. Die Vermeidung sonst notwendiger Casts bei der Programmierung

Das Konzept von Generics kann auch auf Methodenebene herabgebrochen werden. Hier muss der Typ-Stellvertreter in der Reihe der Modifier angegeben werden. Die generische Methode `create()` im folgenden Beispiel erzeugt ein typsicheres `Box`-Objekt, wobei der gewünschte Typ über das Argument festgelegt wird.

```
1  public class GenericMethods {
2
3    public static void main(String[] args) {
4      Box<String>  b1= createBox("xxx");
5      Box<Integer> b2 = createBox(13);
6      //Box<String>  b3= createNumberBox("xxx");
7      Box<Integer> b4 = createNumberBox(13);
8
9      System.out.println(b1);
10     System.out.println(b2);
11     System.out.println(b4);
12   }
13
14   public static<T> Box<T> createBox(T f) {
15     return new Box<T>(f);
16   }
17
18   public static<T extends Number> Box<T> createNumberBox(T f) {
19     return new Box<T>(f);
20   }
21 }
```

Listing 2.13: Generische Methode

Ausgabe:

```
java.lang.String
java.lang.String
java.lang.Integer
```

Die Methode `createNumberBox` zeigt eine weitere Sprachmöglichkeit bei der Verwendung von Typ-Stellvertretern. Diese können mit `extends` so eingeschränkt werden, dass nur Typen erlaubt sind, die im Rahmen der ”is-a”-Beziehung mit `Number` verträglich sind. Zeile 6 würde nun auf einen Compilerfehler führen, da das an die Methode `createNumberBox` gesendete Objekt vom Typ `String` und damit nicht vom Typ `Number ist`. Alternativ ist auch die Einschränkung mit `super` möglich.

Das nächste Beispiel zeigt, dass die Technik auch auf Interfaces anwendbar ist. Die generische Methode `max` bestimmt mit Hilfe der Methode `compareTo` das größere von zwei übergebenen Javaobjekten, die (1) beide vom gleichen Typ sein müssen und (2) beide das Interface `java.lang.Comparable` implementieren müssen:

```
1 // Generische Methode mit Typ-Einschraenkung
2 public <C extends Comparable<C>> C max(C a, C b) {
3   return a.compareTo(b) > 0 ? a : b;
4 }
5 System.out.println(max("A", "Z"));
```

Listing 2.14: Generische Methode mit Typ-Einschränkung

Kapitel 3

Neuerungen in Java 8

3.1 Lambda - Ausdrücke

3.1.1 Functional Interface

Mit Java 8 wurde der Begriff des *Functional Interface* eingeführt. Dabei handelt es sich um ein Interface, das genau eine abstrakte Methode aufweist und grundlegend für die Verwendung von Lambda-Ausdrücken ist. Konkret stellt ein Lambda-Ausdruck eine anonyme Implementierung dieser einzigen abstrakten Methode dar, es definiert also ein Funktionsobjekt.

Eigenschaften eines Functional Interface:

- Enthält genau eine abstrakte Methode
- Wird auch *SAM - Typ* (Single Abstract Method) genannt
- Gibt es schon vor Java 8 (`Runnable`, `Comparator<T>`, `ActionListener`, ...)
- Sollte mit `@FunctionalInterface` annotiert werden

```
@FunctionalInterface
public interface Function {
  // Einzige abstrakte Methode
  double f(double x);

  // Alle im Typ Object definierten Methoden können zusätzlich angegeben werden
  @Override
  String toString();
}
```

Listing 3.1: Functional Interface

Dieses Interface kann nun mit Hife einer anonymen inneren Klasse konkretisiert werden:

```
Function sin = new Function() {

  @Override
  public double f(double x) {
    return Math.sin(x);
  }
};
```

Listing 3.2: Implementierung als anonyme innere Klasse

Man kann es aber auch als Lambda-Ausdruck implementieren:

```
Function sqrt = (double x) -> { return Math.sqrt(x); };

// Zuweisung auf Object nicht möglich!!!
// Ein Lambda-Ausdruck ist nicht vom Typ Object
```

```
5 Object sqrt = (double x) -> { return Math.sqrt(x); };   // Kompilerfehler
```

Listing 3.3: Implementierung als Lambda-Ausdruck

Dabei gilt:

- Ein Lambda-Ausdruck ist eine anonyme Implementierung der eindeutig definierten abstrakten Methode eines Functional Interfaces
- Wegen der Eindeutigkeit der Methode kann auf den Methodennamen verzichtet werden
- Es ist lediglich die Funktionalität anzugeben
- Ein Lambda-Audruck ist nicht kompatibel mit dem Typ `java.lang.Object`, da hier die zu implementierende Methode nicht identifiziert werden kann

Beispiel

Lambda-Ausdrücke werden oft als Funktionsparameter verwendet. Das folgende Beispiel demonstriert dies an Hand eines generischen Generators für Funktionstabellen. Dabei wird das oben definierte Functional Interface `Function` eingesetzt:

```
1  public class Tools {
2
3      // Tools beinhaltet nur statische Methoden --> privater Konstruktor
4      private Tools() {}
5
6      /**
7       * Diese Funktion gibt die Wertetabelle einer mathematischen Funktion
8       * auf System.out aus.
9       * @param f Functional Interface mit der Methode double f(double x)
10      * @param x0 Untere Intervallgrenze des Arguments
11      * @param x1 Obere Intervallgrenze des Arguments
12      * @param s Schrittweite des Arguments
13      */
14     public static void printValueTable(Function f, double x0, double x1, double s) {
15
16         System.out.println("          x |      f(x)    ");
17         System.out.println("------------|--------------");
18         for(double x = x0; x <= x1; x += s) {
19             System.out.format(" %10.2f | %10.2f\n", x, f.f(x));
20         }
21         System.out.println();
22     }
23
24 }
```

Listing 3.4: Functional Interface als Funktionsparameter

Eine zugehörige Anwendung:

```
1  public class FunctionTest {
2
3      public static void main(String[] args) {
4          // Anonyme innere Klasse
5          Tools.printValueTable(new Function() {
6              @Override
7              public double f(double x) {
8                  return x * x;
9              }
10         }, 0.0, 2.0, 0.25);
11
12         // Lambda-Ausdruck
13         Tools.printValueTable(x -> Math.sqrt(x), 0.0, 2.0, 0.25);
14
```

```
15          // Funktionsreferenz
16          Tools.printValueTable(Math::sin, 0.0, 2.0, 0.25);
17      }
18  }
```

Listing 3.5: Anwendung: Plotter von Funktionstabellen

Ausgabe (auszugsweise):

```
           x |       f(x)
-------------|---------------
        0.00 |         0.00
        0.25 |         0.06
 ...
        2.00 |         4.00
```

In obiger Anwendungung wurden Lambda-Ausdrücke bereits in optimierter Syntax verwendet. Näheres dazu im nächsten Abschnitt.

3.1.2 Syntax eines Lambda-Ausdrucks

Ein Lambdaausdruck ist also eine anonyme Implementierung eines Functional Interfaces, die sehr oft ”OnDemand” (also im Code genau dort, wo man die Implementierung benötigt) angegeben wird:

```
1  @FunctionalInterface
2  public Interface SAM1 {
3    RETURN_TYPE methodName(METHOD_PARAMS);
4  }
5
6  // SAM-Typ als anonyme innere Klasse
7  new SAM1() {
8    @Override
9    public RETURN_TYPE methodName(METHOD_PARAMS) {
10     METHOD-BODY
11   }
12 };
13
14 // SAM-Typ als Lambda
15 (METHOD-PARAMS) -> { METHOD-BODY };
```

Listing 3.6: Syntax eines Lambda-Ausdruckes

Kurzformen der Syntax

Es sind also folgende Kurzschreibweisen von Lambdas zugelassen:

- Typangaben für die Parameter der Methode dürfen weggelassen werden: *TypeInferenz*
- Besteht die Parameterliste nur aus einem einzigen Parameter, so dürfen auch die runden Klammern weggelassen werden.
- Falls der Funktionsrumpf nur aus einem Ausdruck besteht, so dürfen die Funktionsklammern weggelassen werden. Dann sind auch das Semikolon und das Schlüsselwort `return` wegzulassen.
- Wird die abstrakte Methode durch eine bereits implementierte Methode ersetzt, so kann dies auch in Form einer (in Java 8) neu eingeführten Methodenreferenz (z.B. `Math::max`)geschehen. Mehr dazu im Abschnitt über Methodenreferenzen.

Die Kurzformen der Syntax werden nun Hand eines konkreten Beispiel erklärt:

```
1  // Zuerst ein FunctionalInterfcae definieren
2  // File: MySAM.java
3
4  @FunctionalInterface
5  public interface MySAM {
```

```
6       int foo(int x, int y);
7   }
8
9   // Nun anwenden
10  // File: Lambdas_1.java
11  public class Lambdas_1 {
12
13      public static void main(String[] args) {
14          // Implementierung als anonyme innere Klasse
15          MySAM s1 = new MySAM() {
16
17              @Override
18              public int foo(int x, int y) {
19                  return x + y;
20              }
21
22          };
23
24          // Implementierung als Lambda-Expression mit voller Syntax
25          MySAM s2 = (int x, int y) -> {
26              return x + y;
27          };
28
29          // Type Inferenz bei den Parametern
30          MySAM s3 = (a, b) -> {
31              return a * b;
32          };
33
34          // Funktionsklammern (und Semikolon) bei einem Ausdruck weglassen
35          // Ist der einzige Ausdruck das return-Statement so ist auch return
                wegzulassen
36          MySAM s4 = (c, d) -> c > d ? c : d;
37
38          // Verwendung von Methodenreferenzen
39          MySAM s5 = Math::max;
40
41          System.out.println(s1.foo(10, 20));
42          System.out.println(s2.foo(10, 20));
43          System.out.println(s3.foo(10, 20));
44          System.out.println(s4.foo(10, 20));
45          System.out.println(s5.foo(10, 20));
46
47
48          // Lambda mit leerer Parameterliste
49          Runnable r = () -> System.out.println("Hello from Lambdas");
50
51          new Thread(r).start();
52          new Thread(() -> System.out.println("Es geht noch schneller")).start();
53
54          // Comparator mit Lambdas
55          Comparator<String> c = (String str1, String str2) -> (Integer.compare(str1.
                length(), str2.length()));
56
57          List<String> lst = Arrays.asList("Franz", "Eva", "Günter", "Alexander", "
                Richard");
58          // DefaultMethode in java.util.ArrayList
59          lst.sort(c);
60          System.out.println(lst);
61
62          // oder noch schneller
63          lst.sort((str1, str2) -> (Integer.compare(str2.length(), str1.length())));
64          System.out.println(lst);
65      }
```

```
66 }
```

Listing 3.7: Kurzformen von Lambda Ausdrücken

Ausgabe

```
30
30
200
20
20
Hello from Lambdas
Es geht noch schneller
[Eva, Franz, Günter, Richard, Alexander]
[Alexander, Richard, Günter, Franz, Eva]
```

3.1.3 Unterschiede zu anonymen inneren Klassen

Beim Arbeiten mit Lambda-Ausdrücken sind noch die folgenden Unterschiede zu anonymen inneren Klassen zu beachten:

- Verwendet man in einem Lambdaaudruck eine lokale Variable der umgebenden Methode, so ist diese (implizit) final. Obwohl das Schlüsselwort `final` nicht notwendigerweise angegeben werden muss, führt jeder Änderungsversuch einer lokalen Variablen nach Verwendung im Lambdaausdruck zu einem Compilerfehler. In Lambdaaudrücken verwendete lokale Variable der umgebenden Methode sind *effektiv final*, sie müssen also nicht mehr als final deklariert werden. es genügt, wenn sie ihren Wert zur Laufzeit des Programmes nicht ändern.
- Verwendet man in einem Lambdausdruck die `this`-Referenz, so greift man damit anders als bei nicht statischen anonymen inneren Klassen auf das `this`-Objekt der umgebenden Instanzmethode zu. Lambdas repräsentieren lediglich ein Stück Funktionalität und haben keine Objektbindung.
- Ein Lambdaausdruck wird zu keiner eigenen `*.class`-Datei compiliert.

3.1.4 Wichtige Functional Interfaces

Java 8 stellt im Paket `java.util.function` in etwa 40 Functional Interfaces zur Verfügung, die in bestimmten Programmiersituationen verwendet werden können. Die wichtigsten davon werden im Folgenden kurz vorgestellt:

Consumer<T>

Dieses Interface beschreibt eine Aktion auf einem Element vom Typ `T`:

```
1 @FunctionalInterface
2 public interface Consumer<T> {
3   void accept(T t);
4 }
```

Listing 3.8: Functional Interface `Consumer`

Predicate<T>

Dieses Interface testet das der Methode `boolean test(T)` übergebene Objekt. Damit lassen sich Filteroperationen definieren.

```
1 @FunctionalInterface
2 public interface Predicate<T> {
3   boolean test(T t);
4 }
```

Listing 3.9: Functional Interface `Predicate`

Function<T,R>

Dieses Interface definiert eine Abbildungsfunktion `R apply(T)` vom Typ `T` in den Typ `R`. Damit wird eine allgemeine Transformation beschrieben. Sehr oft wird dieses Functional Interface eingesetzt, um aus einem komplexen Typ ein Attribut zu extrahieren.

```
1 @FunctionalInterface
2 public interface Function<T,R> {
3   R apply(T t);
4 }
```

Listing 3.10: Functional Interface `Function`

Supplier<T>

Dieses Interface wird verwendet, um Objekte vom Typ `T` ohne Inputdaten zu erzeugen.

```
@FunctionalInterface
public interface Supplier<T> {
  T get();
}
```

Listing 3.11: Functional Interface `Supplier`

BiConsumer<T,U>

Wie `Consumer`, allerdings mit zwei Inputtypen `T` und `U`.

```
@FunctionalInterface
public interface BiConsumer<T,U> {
  void accept(T t, U u);
}
```

Listing 3.12: Functional Interface `BiConsumer`

BiFunction<T,U,R >

Wie `Function`, allerdings mit zwei Inputtypen `T` und `U`.

```
@FunctionalInterface
public interface BiFunction<T,U,R> {
  R apply(T t, U u);
}
```

Listing 3.13: Functional Interface `BiFunction`

Zum besseren Verständnis wird der Einsatz an einem Beispiel demonstriert:

Beispiel

In folgendem Beispiel wird demonstriert, wie einige der oben beschriebenen Functional Interfaces effizient eingesetzt werden können:

```
public class Lambdas_2 {

    public static void main(String[] args) {
        // Liste zum Testen
        List<Integer> l = Arrays.asList(10, 17, 12, 27, 18, 28, 30);

        // Teenager filtern und zeilenweise ausgeben
        filterAndConsume(l, x -> x / 10 == 1, System.out::println);
        System.out.println("-----------------");

        // Alle Werte mit Endziffer 7 filtern und als Binärzahl ausgeben
        filterAndConsume(l, x -> x % 10 == 7, x -> System.out.format("%s ", Integer.
            toBinaryString(x)));
        System.out.println();

    }

    /**
     * Die Methode übernimmt eine Liste vom Typ Integer und führt für alle Elemente
     * die Methode test() aus. Damit werden die Elemente der Liste gefiltert.
     * Für die erfolgreich gefilterten Elemente wird nun die Methode accept()
     * des Consumers c aufgerufen.
     * @param lst Die zu verarbeitende Liste
     * @param p Ein Predicate zum Filtern der Liste
```

```
24      * @param c Ein Consumer, der eine Aktion mit den gefilterten Elementen
            durchführt
25      */
26     public static void filterAndConsume(List<Integer> lst, Predicate<Integer> p,
           Consumer<Integer> c) {
27         for(Integer ii : lst) {
28             if(p.test(ii)) {
29                 c.accept(ii);
30             }
31         }
32     }
33 }
```

Listing 3.14: Einsatz von Predicate und Consumer

Ausgabe

```
10
17
12
18
-----------------
10001 11011
```

Die Mächtigkeit dieses Konzepts wird noch deutlicher, wenn man die Methode `filterAndConsume()` generisch macht. Das folgende Beispiel demonstriert dies:

```
1  public class Lambdas_3 {
2
3      public static void main(String[] args) {
4          // Liste vom Typ java.awt.Point zum Testen
5          List<Point> point = Arrays.asList(new Point(3, 4), new Point(-5, 7),
6                                            new Point(-4, 0), new Point(4, 7));
7
8          // Es werden alle Punkte im ersten Quadranten gefiltert,
9          // ihr Abstand zum Ursprung berechnet und ausgegeben
10         filterTransformConsume(point,
11                 p -> p.x > 0 && p.y > 0,
12                 p -> Math.hypot(p.x, p.y),
13                 System.out::println);
14
15     }
16
17     /**
18      * Die Methode übernimmt eine Liste vom Typ T und führt für alle
19      * Elemente die Methode test() aus. Danach werden die gefilterten Elemente
20      * in den Typ U konvertiert und für diese Daten wird die Methode accept()
21      * des Consumers c aufgerufen.
22      *
23      * @param lst Die zu verarbeitende Liste vom Typ T
24      * @param p Ein Predicate zum Filtern der Liste
25      * @param f Transformiert den Typ T in den Typ U
26      * @param c Consumer, der eine Aktion mit den gefilterten Elemente durchführt
27      */
28     public static <T, U> void filterTransformConsume(List<T> lst,
29             Predicate<T> p,
30             Function<T, U> f,
31             Consumer<U> c) {
32         for (T tt : lst) {
33             if (p.test(tt)) {
34                 U uu = f.apply(tt);
35                 c.accept(uu);
36             }
37         }
```

```
38      }
39  }
```

Listing 3.15: Generischer Einsatz von Functional Interfaces

Ausgabe

```
5.0
8.06225774829855
```

3.2 Weitere Spracherweiterungen

In diesem Abschnitt werden weitere Spracherweiterungen beschrieben, die in Zusammenhang mit Lambdaausdrücken notwendig bzw. sinnvoll wurden.

3.2.1 Default-Methoden in Interfaces

Mit Java 8 sind auch sogenannte Default-Methoden in Interfaces zugelassen. Dabei handelt es sich um Methoden, die in einem Interface bereits implementiert sind und in implementierenden Klassen überschrieben werden können aber nicht müssen. Die Einführung solcher Default-Methoden war notwendig, da das Hinzufügen neuer abstrakter Methoden (die Parameter vom Typ Functional Interface besitzen und daher Lambdaausdrücke verwenden) nicht möglich war, ohne die Abwärtskompatibilität aufzugeben. Eine solche abstrakte Methode müsste natürlich in allen Klassen implementiert werden, welche das Interface verwenden. Bei Default-Methoden besteht dieses Problem nicht, da diese implementiert sind und nur bei Bedarf überschrieben werden müssen.
Eine Default-Methode ist also implementiert und wird mit dem Schlüsselwort `default` definiert:

```
1  public interface Test {
2
3    // Herkömmliche abstrakte Methode
4    void foo();
5
6    // Defaultmethode
7    default void bar() {
8      // Do Something
9    }
10 }
```

Listing 3.16: Interface mit Default-Methode

Wichtige Default-Methoden

Das Interface `java.util.List<E>` wurde um die Default-Methode `sort()` ergänzt:

```
1  public interface List<E> extends Collection<E> {
2
3    // ...
4
5    default void sort(Comparator<? super E> c) {
6      Collections.sort(this, c);
7    }
8  }
```

Listing 3.17: Defaultmethode in `List<E>`

Das Interface `java.lang.Iterable<E>` wurde um die Default-Methode `forEach()` ergänzt:

```
1  public interface Iterable<T> {
2
3    // ...
4
```

```
5   default void forEach(Consumer<? super T> action) {
6     Objects.requireNonNull(action);
7     for (T t : this) {
8       action.accept(t);
9     }
10  }
11 }
```

Listing 3.18: Defaultmethode in `Iterable<E>`

Das folgende Beispiel demonstriert das Arbeiten mit diesen Default-Methoden:

```
1 public class Lambdas_4 {
2
3     public static void main(final String[] args) {
4         final List<String> klassen = Arrays.asList("1AHIF", "3BHIF", "2BHIF", "4AHIF
              ", "1BHIF");
5         klassen.sort((s1,s2) -> s1.compareToIgnoreCase(s2));
6         klassen.forEach(it -> System.out.print(it.substring(0,2) + " ") );
7         System.out.println();
8     }
9 }
```

Listing 3.19: Default-Methoden `sort()` und `forEach()`

Ausgabe

```
1A 1B 2B 3B 4A
```

Default-Methoden Mehrfachvererbung

Auf Grund der Existenz von Default-Methoden kann nun zumindest im Bereich von Methoden Mehrfachvererbung auftreten. Das folgende Beispiel demonstriert die Problematik:

```
1  interface I1 {
2    default int foo() {
3      return 0;
4    }
5  }
6  interface I2 {
7    default int foo() {
8      return 4711;
9    }
10 }
11
12 /*
13   Die folgende Klassendefinition compiliert mit folgender Fehlermeldung:
14   class DoNotCompile inherits unrelated defaults for foo() from types I1 and I2
15
16   class DoNotCompile implements I1, I2 { }
17 */
18
19 /*
20   Lösung 1:
21   Eigene Version der Methode foo() implementieren
22 */
23
24 class Ok1 implements I1, I2 {
25     @Override
26     public int foo() {
27         return 1111;
28     }
29 }
30
```

```
/*
  Lösung 2:
  Die eigene Implementierung der Methode foo() ruft eine Default-Methode in einem
      der Interfaces auf
*/

class Ok2 implements I1, I2 {
    @Override
    public int foo() {
        return I1.super.foo();
    }
}

public class Lambdas_5 {

    public static void main(final String[] args) {
        System.out.println(new Ok1().foo());
        System.out.println(new Ok2().foo());
    }
}
```

Listing 3.20: Default-Methoden und Mehrfachvererbung

Ausgabe

```
1111
0
```

3.2.2 Statische Methoden in Interfaces

Seite Java 8 drrfen in Interfaces auch statische Methoden stehen. Damit ist es möglich, direkt in Interfaces Hilfsmethoden aufzunehmen, die bisher in Utilityklassen ausgelagert werden mussten. So gibt es z.B. im JDK 8 im Interface `Comparator<T>` nun auch die folgende Methode `reverseOrder()`, die bisher nur in der Utilityklasse `java.util.Collections` zu finden war:

```
@FunctionalInterface
public interface Comparator<T> {

  // ...

  public static <T extends Comparable<? super T>> Comparator<T> reverseOrder() {
    return Collections.reverseOrder();
  }

  // ...
}
```

Listing 3.21: Statische Methoden in Interfaces

```
public class StaticMethods {

    public static void main(String[] args) {
        List<Integer> lst = Arrays.asList(12, 17, 11, 9, 14, 56, 8);

        lst.sort(null);                          // natural Order
        System.out.println(lst);

        lst.sort(Comparator.reverseOrder());     // reverse natural Order
        System.out.println(lst);
    }

}
```

Listing 3.22: Statische Methoden Anwendung

Ausgabe

```
[8, 9, 11, 12, 14, 17, 56]
[56, 17, 14, 12, 11, 9, 8]
```

3.2.3 Methodenreferenzen

Passt eine bereits implementierte Methode exakt zur Schnittstelle der einzigen abstrakten Methode eines Functional Interfaces, so kann der Lambdaausdruck auch durch eine Methodenreferenz ersetzt werden. Eine Methoden-Referenz ist also wie ein Lambda-Ausdruck eine Implementierung eines Functional Interfaces, wobei die Implementierung bereits existiert. Wie üblich bestimmt der Kontext die Paramterliste und den Rückgabetyp. Die Syntax lautet:

```
Classname::staticMethod            // Referenz auf eine statische Methode
Classname::instanceMethod          // Referenz auf eine Instanzmethode
Classname::new                     // Referenz auf einen Konstruktor
Arraytyp[]::new                    // Referenz zur Arrayerzeugung
```

Beispiele:

```
System.out::println
Person::getName
Person::new
Integer[]::new
```

Beispiel

```
1  import java.util.function.Function;
2  import java.util.function.Supplier;
3
4
5  class Person {
6      private final String zuname;
7      private final String vorname;
8
9      public Person(String zuname, String vorname) {
10         this.zuname = zuname;
11         this.vorname = vorname;
12     }
13
14     public String getZuname() {
15         return zuname;
16     }
17
18     public String getVorname() {
19         return vorname;
20     }
21
22     public Character getInitialen(int x) {
23         switch(x) {
24             case 1: return zuname.charAt(0);
25             case 2: return vorname.charAt(0);
26             default: throw new IllegalArgumentException("Falsches Argument");
27         }
28     }
29 }
30
```

```
@FunctionalInterface
interface FI_1 {
    StringBuilder foo();
}

@FunctionalInterface
interface FI_2<T> {
    T[] bar(int size);
}

@FunctionalInterface
interface FI_3 {
    String baz(String str, int start, int end);
}

@FunctionalInterface
interface FI_4 {
    Person createPerson(String zn, String vn);
}

@FunctionalInterface
interface FI_5 {
    String zuname(Person p);
}

public class MethodReference {

    public static void main(String[] args) {
        FI_1 f11 = () -> new StringBuilder("Hallo");
        FI_1 f12 = StringBuilder::new;

        FI_2 f21 = Integer[]::new;

        System.out.println(f21.bar(100).length);


        String s = "Hallo Java 8";
        FI_3 f31 = String::substring;
        System.out.println(f31.baz(s, 6, 11));

        FI_4 f41 = Person::new;
        Person p = f41.createPerson("Reichel", "Otto");

        FI_5 f51 = Person::getZuname;
        System.out.println(f51.zuname(p));

        Supplier<String> z = p::getVorname;
        System.out.println(z.get());

        Function<Integer, Character> c = p::getInitialen;
        System.out.println(c.apply(1));

     }
}
```

Listing 3.23: Methodenreferenzen

Ausgabe

```
100
Java
Reichel
Otto
R
```

Achtung

Links von den zwei Doppelpunkten kann auch eine Referenz stehen, was dann immer eine Objektmethode referenziert.
Während `Person::getZuname` eine Funktionsrefernz ist, ist
`p::getZuname` vom Typ `java.util.function.Supplier`
bzw.
`p::getInitialen` vom Typ `java.util.function.Function`.

Kapitel 4

Das Collection-API

4.1 Einführung

Das Collection-API stellt eine Sammlung von Interfaces und Klassen zur Verfügung, mit deren Hilfe Daten (Objekte) abgespeichert und manipuliert werden können. Es befindet sich im Package `java.util`. Seit dem JDK 1.2 gibt es ein ”neues” Collection-API, das parallel zu dem ”alten” seit dem JDK 1.0 existierenden Collection-API geführt wird und sich teilweise mit diesem überdeckt. Das ”alte” Collection-API besteht im Wesentlichen aus den Klassen `Vector`, `Stack`, `Dictionary`, `Hashtable` und `BitSet`. In diesem Abschnitt wird das ”neue” Collection-API vorgestellt, das im JDK 1.5 durch einige neue Klassen erweitert wurde (Queues).

4.2 Die grundlegenden Interfaces und Klassen

Das Collection-API basiert auf Interfaces, deren Aufgabe es ist, eine gemeinsame Schnittstelle zur Manipulation der Daten (einfügen, entfernen) und zum Datenaustausch zwischen mehreren Collections festzulegen. Es gibt zwei grundlegende Vererbungshierarchien:

Die Vererbungshierarchie für Collections

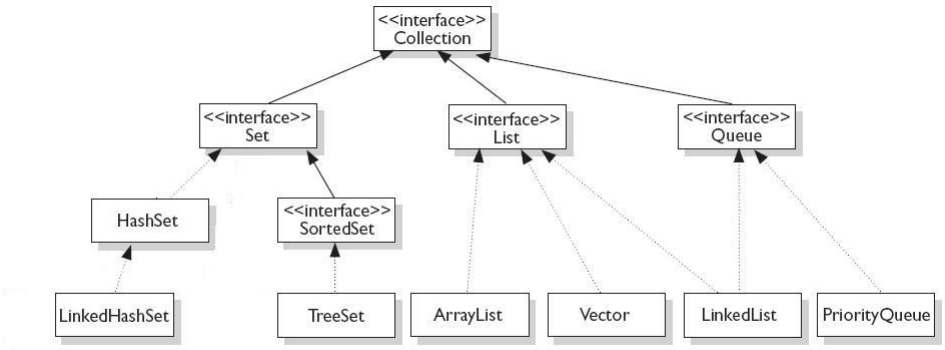


Abbildung 4.1: Vererbungshierarchie für Collections

Das Interface `Collection` definiert eine Schnittstelle zum Speichern einfacher Objekte. Es hat die folgenden Subinterfaces:

- `Set`
  In einem `Set` können nur Unikate abgespeichert werden, d.h. zwei Objekte `x` und `y`, für die `x.equals(y)` den Wert `false` liefert.

  - SortedSet
    Erweitert das Interface Set. Die Elemente werden sortiert in Form eines binären Suchbaumes abgespeichert.

- List
  Eine List speichert die Daten geordnet, d.h. die Reihenfolge der Daten wird bei der Speicherung festgelegt.

- Queue
  Eine Warteschlange realisiert ein Speichermodell nach dem FIFO (first in - first out) Prinzip.

Die Vererbungshierarchie für Maps

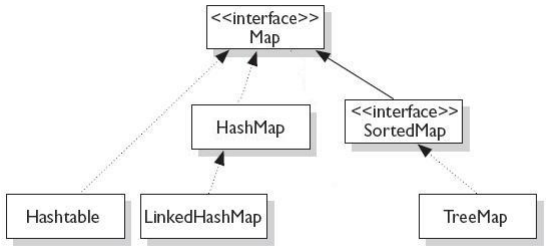


Abbildung 4.2: Vererbungshierarchie für Maps

Das Interface Map definiert eine Schnittstelle zum Speichern von Schlüssel-Werte-Paaren. Es hat das folgende Subinterface:

- SortedMap
  Erweitert das Interface Map und speichert die Schlüssel-Werte-Paare nach sortierten Schlüsseln

4.3 Die Interfaces Comparable und Comparator

Möchte man Arrays oder Collections (Listen) sortieren, so ist es notwendig, dass die gespeicherten Objekte eine kleiner - größer - Beziehung definieren. Das Java-API stellt dafür zwei Interfaces zur Verfügung:

- Das Interface java.lang.Comparable
  Implementiert eine Klasse dieses Interface, so erhalten die Objekte dieser Klasse eine sogenannte natürliche Sortierreihenfolge.

- Das Interface java.util.Comparator
  Eine Klasse, die dieses Interface implementiert, legt für die Objekte einer anderen Klasse eine benutzerdefinierte Sortierreihenfolge fest. So kann also bei Bedarf eine andere als die natürliche Sortierreihenfolge definiert werden.

4.3.1 Das Interface java.lang.Comparable

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

Die Methode compareTo() muss dabei folgende Funktionalität aufweisen:

- Das Ergebnis ist kleiner 0, wenn im Sinne der Sortierreihenfolge this kleiner als o ist.
- Das Ergebnis ist gleich 0, wenn im Sinne der Sortierreihenfolge this gleich o ist.
- Das Ergebnis ist größer 0, wenn im Sinne der Sortierreihenfolge this größer als o ist.

Wichtige Klassen der API, die dieses Interface implementieren:

- String
- Alle Wrapperklassen
- Enum

4.3.2 Das Interface java.util.Comparator<T>

Möchte man eine andere als die in der Klasse des Objektes definierte natürliche Sortierreihenfolge verwenden, so erzeugt man eine Hilfsklasse (oft als innere Klasse), die das Interface java.util.Comparator implementiert:

```
public interface Comparator<T> {
  int compare(T o1, T o2);
}
```

Die Methode compare() muss dabei folgende Funktionalität aufweisen:

- Das Ergebnis ist kleiner 0, wenn im Sinne der Sortierreihenfolge o1 kleiner als o2 ist.
- Das Ergebnis ist gleich 0, wenn im Sinne der Sortierreihenfolge o1 gleich o2 ist.
- Das Ergebnis ist größer 0, wenn im Sinne der Sortierreihenfolge o1 größer als o2 ist.

4.3.3 Beispiel

Das folgende Beispiel definiert eine Klasse Data, die zwei Integerwerte kapselt. Die Klasse erhält eine natürliche Sortierreihenfolge, bei der zunächst aufsteigend nach a und bei gleichen a aufsteigend nach b sortiert wird. Außerdem wird eine weitere Sortierreihenfolge definiert, in der zunächst aufsteigend nach der Summe a+b sortiert wird. Sollten die Summen gleich sein, so wird die natürliche Sortierreihenfolge verwendet.

```
1  import java.util.*;
2
3  public class Vergleichen {
4    public static void main(String []args) {
5      Data []f = { new Data(10,10), new Data( 3,7), new Data(3,20),
6                   new Data( 7, 3), new Data(15,5), new Data(7,13)};
7      Arrays.sort(f);
8      for(Data d : f) {
9        System.out.print(d + " ");
10     }
11     System.out.println();
12     Arrays.sort(f, new Data.Vergleich_2());
13     for(Data d : f) {
14       System.out.print(d + " ");
15     }
16     System.out.println();
17   }
18 }
19
20 class Data implements Comparable<Data> {
21   private int a, b;
22
23   public Data(int a, int b) {
24     this.a = a;
25     this.b = b;
26   }
27
28   public String toString() {
29    return String.format("%02d/%02d", a, b);
30   }
```

```
31
32    public int compareTo(Data o) {
33      int ret = new Integer(this.a).compareTo(o.a);
34      if(ret == 0)
35        return new Integer(this.b).compareTo(o.b);
36      return ret;
37    }
38
39    public static class Vergleich_2 implements Comparator<Data> {
40      public int compare(Data o1, Data o2) {
41        int ret = new Integer(o1.a + o1.b).compareTo(o2.a + o2.b);
42        if(ret == 0)
43          return o1.compareTo(o2);
44        return ret;
45      }
46    }
47  }
```

Listing 4.1: Anwendung Comparable und Comparator

Ausgabe:

```
03/07 03/20 07/03 07/13 10/10 15/05
03/07 07/03 07/13 10/10 15/05 03/20
```

4.4 Der Equals - Hashcode - Vertrag

Werden Objekte in HashSets oder Schlüssel in HashMaps gespeichert, so sind hier nur Unikate erlaubt. Die Prüfung, ob zwei Objekte `o1` und `o2` gleich sind, erfolgt wenn notwendig in einem zweistufigen Verfahren:

1. Zunächst werden die Hashcodes von `o1` und `o2` verglichen. Sind diese verschieden, dann gelten die Objekte als verschieden.
2. Sind die Hashcodes gleich, so wird mit Hilfe der Methode `o1.equals(o2)` festgestellt, ob die beiden Objekte gleich oder verschieden sind.

Daher dürfen die von `java.lang.Object` geerbten Methoden `equals()` und `hashCode()` nicht unabhängig voneinander gesehen werden. Wird `equals()` überschrieben, so muss auch `hashCode()` nach folgenden Regeln überschrieben werden:

- Wenn für zwei Objekte `o1` und `o2` die Methode `equals()` den Wert `true` liefert, so müssen die Objekte auch den gleichen Hashcode haben.
- Wenn für zwei Objekte `o1` und `o2` die Methode `equals()` den Wert `false` liefert, so können die Objekte trotzdem (zufällig) den gleichen Hashcode haben.

Die Berechnung des Hashcodes sollte einen aus den Daten des Objektes errechneten Integerwert aus einem möglichst breiten Zahlenraum haben. Transiente Datenfelder dürfen für die Berechnung des Hashcodes nicht herangezogen werden, damit das Objekt vor und nach der Serialisierung den gleichen Hashcode hat.
Das folgende Listing zeigt sauber überschriebene Methoden `equals()` und `hashCode()` der Klasse `Data_1`:

```
1  class Data_1 {
2
3    private int a, b;
4
5    public Data_1(int a, int b) {
6      this.a = a;
7      this.b = b;
8    }
9
```

```
10    @Override
11    public String toString() {
12      return String.format("%02d/%02d", a, b);
13    }
14
15    @Override
16    public boolean equals(Object obj) {
17      if (obj == null) {
18        return false;
19      }
20      if (getClass() != obj.getClass()) {
21        return false;
22      }
23      final Data_1 other = (Data_1) obj;
24      if (this.a != other.a) {
25        return false;
26      }
27      if (this.b != other.b) {
28        return false;
29      }
30      return true;
31    }
32
33    @Override
34    public int hashCode() {
35      return (17 * a) ^ (25 * b);
36    }
37  }
```

Listing 4.2: Überschreiben von `equals()` und `hashCode()`

4.5 Die Vererbungshierarchie `java.util.Collection`

In diesem Abschnitt werden die wichtigsten Methoden des Interfaces `Collection` zusammengestellt, die Subinterfaces und die konkreten Implementierungen vorgestellt.

4.5.1 Das Interface `java.util.Collection`

```
public interface Collection<E>
extends Iterable<E>
```

Basisoperationen

`public int size()`
Liefert die Anzahl der in dieser Collection aktuell gespeicherten Objekte.

`public boolean isEmpty()`
Liefert `true`, wenn diese Collection leer ist.

`public boolean contains(Object o)`
Liefert `true`, wenn die Collection das Objekt `o` beinhaltet. Exakter wird genau dann `true` geliefert, wenn die Collection wenigstens ein Element `e` enthält, für das der Ausdruck
`(o==null ? e==null : o.equals(e))`
den Wert `true` liefert.

`public boolean add(E o)`
Fügt das Objekt `o` zu dieser Collection hinzu und liefert im Erfolgsfall `true`. Eine optionale, Methode, die nicht von jeder konkreten Collectionklasse unterstützt werden muss. Wirft in diesem Fall die unchecked

UnsupportedOperationException. Alle hier besprochenen Collectionklassen unterstützen alle optionalen Methoden.

public boolean remove(Object o)
Entfernt das Objekt o aus dieser Collection und liefert im Erfolgsfall true. Optional.

Mengenoperationen

public boolean containsAll(Collection<?> c)
Liefert true, wenn diese Collection alle Elemente der Collection c enthält.

public boolean addAll(Collection<? extends E> c)
Fügt alle Elemente der Collection c zu dieser Collection hinzu (Vereinigungsmenge von this und c). Liefert true, wenn das this-Objekt verändert wurde. Optional.

public boolean removeAll(Collection<?> c)
Entfernt alle Elemente der Collection c aus dieser Collection (Differenzmenge von this und c). Liefert true, wenn das this-Objekt verändert wurde. Optional.

public boolean retainAll(Collection<?> c)
Entfernt alle Elemente aus dieser Collection, die nicht auch in c liegen (Durchschnittsmenge von this und c. Liefert true, wenn das this-Objekt verändert wurde. Optional.

public void clear()
Entfernt alle Elemente aus dieser Collection. Optional.

Feldoperationen

public Object[] toArray()
Liefert ein Feld, das alle in dieser Collection gespeicherten Elemente enthält.

public <T> T[] toArray(T[] a)
Wie oben, die Elemente werden wenn möglich in a gespeichert. Ist a nicht groß genug, so wird ein neues Array mit gleichem Laufzeittyp wie a erzeugt und retourniert.

4.5.2 Das Interface java.util.Iterator

public Iterator iterator()
Diese aus java.util.Collection stammende Methode liefert einen Iterator zum Abarbeiten dieser Collection. Ein Iterator ist eine Art Cursor, mit dessen Hilfe die Collection einmal sequentiell abgearbeitet werden kann. Logisch steht er immer zwischen zwei Elementen und wird durch die Methode next() weitergerückt.

Das Interface java.util.Iterator hat den folgenden Aufbau:

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();          // Optional
}
```

Beschreibung der Methoden im Interface java.util.Iterator:

public boolean hasNext()
Liefert true, wenn die zugrundeliegende Collection noch weitere Elemente aufweist.

public E next()
Liefert das nächste Element der zugrundeliegenden Collection. Sollte kein Element mehr existieren, so

wird die unchecked NoSuchElementException geworfen.

public void remove()
Entfernt das aktuelle Objekt (das Ergebnis des letzten next()-Aufrufes) aus der zugrundeliegenden Collection. Optional. Darf pro next()-Aufruf höchstens einmal verwendet werden und wirft beim Verstoß gegen diese Regel die unchecked IllegalStateException.

Der folgende Codeauschnitt zeigt die Iteration durch die Collection c von Strings. Dabei werden mit Hilfe des Iterators alle Strings aus der Liste gelöscht, die länger als 10 Zeichen sind.

```
1 Iterator<String> ii = c.iterator();
2 while(i.hasNext()) {
3   String s = ii.next();
4   if(s != null && s.length() > 10) {
5     ii.remove();
6   }
7 }
```

Arbeitet man eine Collection mit einem Iterator ab, so darf diese Collection während der Abarbeitung nicht verändert werden. Lediglich der Iterator darf mit seiner Methode remove() das aktuelle Elemnt aus der Collection entfernen. Verstößt man gegen diese Regel, so wir die unchecked ConcurrentModifiedException geworfen.
Alternativ zur Verwendung eines Iterators kann eine Collection auch mit Hilfe einer erweiterten for-Schleife abgearbeitet werden:

```
for(String s : c) {
  // Verarbeitung von s
}
```

Dabei Handelt es sich nur um eine einfacher Schreibweise, intern wir der Code mit Hilfe eines Iterators umgesetzt. Daher ist wieder zu beachten, dass innerhalb der erweiterten for-Schleife keine Elemente aus der Liste gelöscht und auch keine Elemente zur Liste hinzugefügt werden dürfen.

4.5.3 Das Interface java.util.Set

Dieses Interface erweitert java.util.Collection

public interface Set<E> extends Collection<E>

und beinhaltet keine neuen Funktionen. Ein Set darf keine gleichen Objekte und nicht mehr als eine null-Referenz beinhalten. Der Vergleich zweier Objekte o1 und o2 wird wenn nötig zweistufig durchgeführt:

1. Zuerst werden die Hashcodes verglichen
2. Bei gleichen Hashcodes wird die Methode o1.equals(o2) verwendet

Die Reihenfolge, in der die Elemente in einem Set abgespeichert werden, wird durch die Hashcodes der gespeicherten Objekte definiert und ist daher unbestimmt. Damit hat eine Collection vom Typ Set genau das Verhalten einer mathematischen Menge.
Werden in einem Set veränderliche (mutable) Objekte gespeichert, so können nach Veränderung eines Objekts zwei gleiche Objekte in dem Set auftreten. Ein solches Set gilt als korrupt und darf nicht mehr benutzt werden, da es undefiniertes Verhalten hat. Aus diesem Grund ist die Methode equals() für die mutable Klassen StringBuffer und StringBuilder nicht überschrieben. Objekte dieser Klassen gelten also nur dann als gleich, wenn es sich um ein und dasselbe Objekt handelt. Damit können auch solche mutable Objekte problemlos in Sets gespeichert werden.
Implementiert man also eine Klasse, deren Objekte in einem HashSet oder einer HashMap abgespeichert werden sollen, so kann man eine von 2 Strategien wählen:

1. Man macht die Objekte der Klasse immutable und überschreibt die Methoden **equals()** und **hashCode()** unter Berücksichtigung des Equals-Hashcode-Vertrages.
2. Man macht die Objekte mutable und überschreibt die von **Object** geerbten Methoden **equals()** und **hashCode()** nicht.

4.5.4 Die Klasse java.util.AbstractSet

Eine abstrakte Klasse, die das Interface java.util.Set implementiert. Sie kann als Schablone für eigene Set-Implementierungen herangezogen werden.

4.5.5 Die Klasse java.util.HashSet

Eine konkrete Set-Implementierung, die im Hintergrund eine HashMap verwendet (Name). Die Ordnung der Elemente ist vom Hashcode dieser Elemente abhängig und damit undefiniert. Die null-Referenz darf (höchstens einmal) gespeichert werden.

```
public class HashSet<E> extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

Konstruktoren

`public HashSet()`
Erzeugt ein HashSet mit Standardwerten: Initialkapazität 16 und Loadfactor 0,75.

`public HashSet(Collection<? extends E> c)`
Erzeugt ein HashSet, das alle Elemente von c (höchstens einmal) enthält. Loadfactor 0,75.

`public HashSet(int initialCapacity)`
Erzeugt ein HashSet mit der angegebenen Initialkapazität und dem Loadfactor 0,75.

`public HashSet(int initialCapacity, float loadFactor)`
Erzeugt ein HashSet mit den übergebenen Parametern.

Kapazität und Loadfactor sind zwei Parameter, die eng mit der Performance beim Iterieren durch das Set verknüpft sind. Die Kapazität gibt die Anzahl der Felder des Sets an, während der Loadfactor aussagt, zu wieviel Prozent das Set gefüllt sein muss, damit die Kapazität automatisch vergrößert wird.
Greifen mehrere Threads auf ein und dasselbe HashSet zu, so sind die Zugriffe zu synchronisieren, d.h. die Klasse HashSet ist nicht threadsafe.

4.5.6 Die Klasse java.util.LinkedHashSet

Eine konkrete Set-Implementierung, die von HashSet abgeleitet ist. Die Elemente werden zusätzlich über eine doppelt verkettete Liste gespeichert, wodurch ihre Ordnung definiert ist. Ein Iterator liefert die Elemente immer in der gleichen Reihenfolge, in der sie eingefügt wurden. Die Klasse LinkedHashSet ist nicht threadsafe.

```
public class LinkedHashSet<E> extends HashSet<E>
implements Set<E>, Cloneable, Serializable
```

Konstruktoren

`public LinkedHashSet()`
Erzeugt ein LinkedHashSet mit Standardwerten: Initialkapazität 16 und LoadFactor 0,75.

`public LinkedHashSet(Collection<? extends E> c)`
Erzeugt ein HashSet, das alle Elemente von c (höchstens einmal) enthält. Loadfactor 0,75.

`public LinkedHashSet(int initialCapacity)`
Erzeugt ein HashSet mit der angegebenen Initialkapazität und dem Loadfactor 0,75.

`public LinkedHashSet(int initialCapacity, float loadFactor)`
Erzeugt ein HashSet mit den übergebenen Parametern.

4.5.7 Das Interface java.util.SortedSet

```
public interface SortedSet<E> extends Set<E>
```

Dieses Interface ist von `java.util.Set` abgeleitet und speichert die Objekte sortiert. Damit dies möglich ist, müssen alle Objekte vergleichbar sein, d.h. ihre Klassen müssen das Interface `java.lang.Comparable` implementieren. Alternativ kann dem `SortedSet` auch ein `Comparator` zugeordnet sein. Das Interface `SortedSet` stellt die folgenden neuen Methoden zur Verfügung:

`public E first()`
Liefert das erste (kleinste) Element im SortedSet.

`public E last()`
Liefert das letzte (größte) Element im SortedSet.

`public SortedSet<E> headSet(E toElement)`
Liefert alle Elemente als SortedSet, die echt kleiner als `toElement` sind. Das Ergebnis ist dabei eine View auf das Original, so dass alle Änderungen im Ergebnis Auswirkungen auf das Original und umgekehrt haben.

`public SortedSet<E> tailSet(E fromElement)`
Liefert eine View auf alle Elemente des SortedSet, die größer oder gleich als `fromElement` sind.

`public SortedSet<E> subSet(Object fromElement, Object toElement)`
Liefert eine View auf alle Elemente des SortedSet, die größer oder gleich `fromElement` und echt kleiner als `toElement` sind.

`public Comparator<? super E> comparator()`
Liefert den zugrundeliegenden Comparator oder `null`, wenn zur Sortierung die natürliche Reihenfolge der Elemente verwendet wird, also kein Comparator auf dem SortedSet definiert ist.

4.5.8 Die Klasse java.util.TreeSet

Eine konkrete SortedSet-Implementierung, welche die Elemente über einen ausbalanzierten binären Baum speichert.

```
public class TreeSet<E> extends AbstractSet<E>
implements SortedSet<E>, Cloneable, Serializable
```

Konstruktoren

`public TreeSet()`
Erzeugt ein leeres `TreeSet`, in dem die Elemente in natürlicher Reihenfolge gespeichert werden.

`public TreeSet(Comparator<? super E> c)`
Erzeugt ein leeres `TreeSet`, wobei die Elemente mit Hilfe des Comparators `c` verglichen werden.

`public TreeSet(Collection<? extends E> c)`
Erzeugt ein `TreeSet`, in dem alle Elemente der Collection `c` (höchstens einmal) gespeichert sind. Verglichen wird nach der natürlichen Reihenfolge.

`public TreeSet(SortedSet<E> s)`
Erzeugt ein `TreeSet`, das dieselben Elemente wie `s` enthält. Verglichen wird nach der in `s` definierten Reihenfolge.

Beispiel: Lottogenerator

Im folgenden Beispiel wird ein **TreeSet** verwendet, um 6 Lottozahlen beim Spiel 6 aus 45 zu erzeugen.

```java
import java.util.*;

public class Lotto {
  public static void main(String []args) {
    Random rd = new Random();
    Collection<Integer> tipp = new TreeSet<>();
    while(tipp.size() < 6) {
      tipp.add(rd.nextInt(45) + 1);  // Boxing
    }
    System.out.println(tipp);
  }
}
```

Listing 4.3: Anwendung von TreeSet: Lottogenerator

Beispiel: TreeSet mit Comparator

Im folgenden Beispiel wird eine Liste mit 20 Zufallszahlen im Bereich von 1 bis 50 gefüllt. Danach wird ein Set (konkret ein `TreeSet`) angelegt, das im Konstruktor mit Hilfe einer Lambda-Expression einen Comparator definiert. Dieser sortiert aufsteigend nach der Einerziffer der hinzugefügten Integerwerte. Das `TreeSet` weist also jede Zahl ab, deren Einerziffer mit der einer bereits im `TreeSet` gespeicheten Zahl übereinstimmt.

```java
public class TreeSet_2 {

  private static Random rd = new Random();

  public static void main(String[] args) {
    List<Integer> lst = new ArrayList<>();
    for(int i = 0; i < 20; i++) {
      lst.add(rd.nextInt(50) + 1);
    }

    System.out.println(lst);

    Set<Integer> ss = new TreeSet<>((i1,i2) -> Integer.compare(i1 % 10, i2 % 10));
    ss.addAll(lst);
    System.out.println(ss);
  }
}
```

Listing 4.4: Anwendung von TreeSet: Lottogenerator

Ausgabe:

```
[38, 28, 38, 8, 31, 43, 38, 11, 19, 25, 47, 43, 1, 39, 6, 9, 15, 35, 10, 43]
[10, 31, 43, 25, 6, 47, 38, 19]
```

4.5.9 Das Interface `java.util.List`

Listen sind geordnete Collections, die auch Duplikate speichern können. Geordnet bedeutet dabei, dass bei der Abarbeitung der Liste durch einen Iterator oder eine erweiterte `for`-Schleife die Elemente immer in jener Reihenfolge geliefert werden, in der sie abgespeichert wurden. Außerdem lassen sich die einzelnen Elemente über einen Index ansprechen. Listen stellen auch einen speziellen Iterator, den `ListIterator` zur Verfügung, mit dem die Liste in zwei Richtungen abgearbeitet werden kann.
Das Interface `java.util.List` erweitert das Interface `java.util.Collection`.

```
public interface List<E> extends Collection<E>
```

Es definiert die folgenden neuen Methoden:

Indizierter Elementzugriff

Alle Methoden werfen bei ungültigem Index die unchecked `IndexOutOfBoundsException`.

`public E get(int index)`
Liefert das Element mit Index `index`.

`public E set(int index, E element)`
Ersetzt das Element an der Position `index` durch `element`. Optional.

`public void add(int index, E element)`
Fügt das Element `element` an der Position `index` ein und schiebt die nachfolgenden Elemente um eine Position nach hinten. Optional.

`public E remove(int index)`
Entfernt das Element an der Position `index`. Optional.

`public boolean addAll(int index, Collection<? extends E> c)`
Fügt alle Elemente der Collection `c` ab der Position `index` ein und schiebt die nachfolgenden Elemente nach hinten. Optional.

Suchen von Elementen

`public int indexOf(Object o)`
Sucht das erste Auftreten von `o` in der Liste. Liefert den gefundenen Index oder -1.

`public int lastIndexOf(Object o)`
Sucht das letzte Auftreten von `o` in der Liste. Liefert den gefundenen Index oder -1.

Teilliste

`public List<E> subList(int fromIndex, int toIndex)`
Liefert eine Teilliste inklusive dem Element mit Index `fromIndex` und exklusive dem Element mit Index `toIndex`.

4.5.10 Das Interface `java.util.ListIterator`

Neben dem aus `Collection` bekannten Iterator unterstützen Listen auch einen zweiten Iterator, den sogenannten ListIterator, mit dem die Liste bidirektional abgearbeitet werden kann.

`public ListIterator<E> listIterator()`
Liefert einen `ListIterator` zum Abarbeiten dieser Liste.

`public ListIterator<E> listIterator(int index)`
Liefert einen `ListIterator` zum Abarbeiten dieser Liste ab der Position `index`.

Das Interface `java.util.ListIterator` hat den folgenden Aufbau:

```
public interface ListIterator<E> extends Iterator<E> {
  boolean hasNext();      // bereits in Iterator definiert
  boolean hasPrevious();
  E next();               // bereits in Iterator definiert
  E previous();
  int nextIndex();
  int previousIndex();
  void remove();          // Optional - bereits in Iterator definiert
  void set(E o);          // Optional
  void add(E o);          // Optional
}
```

Beschreibung der Methoden im Interface `java.util.ListIterator`:

`public boolean hasPrevious()`
Liefert `true`, wenn es vor der aktuellen Iteratorposition noch Elemente gibt.

`public E previous()`
Liefert das vor der Iteratorposition liegende Element der zugrundeliegenden Liste. Sollte kein Element mehr existieren, so wird die unchecked `NoSuchElementException` geworfen.

`public int nextIndex()`
Liefert den Index des nach der Iteratorposition liegenden Listenelementes bzw. die Größe der Liste, wenn der Iterator bereits nach dem letzten Element steht.

`public int prevoiusIndex()`
Liefert den Index des vor der Iteratorposition liegenden Listenelementes bzw. -1, wenn der Iterator bereits vor dem ersten Element steht.

`public void set(E o)`
Ersetzt jenes Listenelement, das durch den letzten Aufruf von `next()` oder `previous()` geliefert wurde durch des Element `o`. Kann nur verwendet werden, wenn seither kein Aufruf von `remove()` oder `add()` erfolgte. Optional.

`public void add(E o)`
Fügt das Element `o` an der aktuellen Iteratorposition ein.

4.5.11 Die Klasse `java.util.AbstractList`

Eine abstrakte Klasse, die das Interface `java.util.List` implementiert und die als Schablone für eigene List-Implementierungen verwendet werden kann.

```
public abstract class AbstractList<E> extends AbstractCollection<E>
implements List<E>
```

4.5.12 Die Klasse `java.util.ArrayList`

Ist eine konkrete Implementierung eines dynamisch wachsenden Feldes. Wird verwendet, wenn oft direkte Zugriffe über einen Index benötigt werden. Die Kapazität stellt die Größe des Feldes dar. Ist nicht threadsafe.

```
public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Konstruktoren

`public ArrayList()`
Erzeugt eine leere Liste mit der Kapazität 10.

`public ArrayList(int initialCapacity)`
Erzeugt eine leere Liste mit der Kapazität `initialCapacity`.

`public ArrayList(Collection <? extends E> c)`
Erzeugt eine Liste, die alle Elemente von `c` enthält und deren Kapazität 110% der Größe von `c` beträgt.

4.5.13 Die Klasse `java.util.Vector`

Hat genau die gleiche Funktionalität wie `java.util.ArrayList`, allerdings sind alle Methoden threadsicher, d.h. synchronisiert. Diese Collectionklasse gibt es bereits seit dem JDK 1.0. Ihre Konstruktoren entsprechen jenen der Klasse `java.util.ArrayList`.

```
public class Vector<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

4.5.14 Die Klasse java.util.LinkedList

Stellt nach außen genau die gleiche Funktionalität wie java.util.ArrayList zur Verfügung, speichert die Elemente aber intern mit Hilfe einer doppelt verketteten Liste. Daher gibt es Performancevorteile, wenn oft Elemente eingefügt oder gelöscht werden sollen, während Zugriff über Indizes linear steigenden Zeitaufwand erfordert. Ist nicht threadsafe.

```
public class LinkedList<E> extends AbstractSequentialList<E>
implements List<E>, Queue<E>, Cloneable, Serializable
```

Konstruktoren

`public LinkedList()`
Erzeugt eine leere Liste.

`public LinkedList(Collection <? extends E> c)`
Erzeugt eine Liste, die alle Elemente von `c` in jener Reihenfolge enthält, wie sie der Iterator von `c` liefert.

4.5.15 Das Interface java.util.Queue

```
public interface Queue<E> extends Collection<E>
```

Eine Queue realisiert eine Warteschlange. Obwohl es andere Möglichkeiten gibt, werden sie typischerweise für FIFO-Container verwendet. Neben den Standard-Collection-Methoden unterstützt das Interface Queue die folgenden Methoden:

`public E element()`
Liefert das Element am Kopf der Queue, ohne es zu entfernen. Wirft eine `NoSuchElementException`, wenn die Queue leer ist.

`public boolean offer(E o)`
Fügt wenn möglich das Element `E` zu dieser Queue hinzu. Liefert im Erfolgsfall `true`, sonst `false`.

`public E poll()`
Entfernt und liefert das Element am Kopf der Queue bzw. `null`, wenn die Queue leer ist.

`public E remove()`
Entfernt und liefert das Element am Kopf der Queue. Unterscheidet sich von `poll()` dadurch, dass bei leerer Queue eine `NoSuchElementException` geworfen wird.

`public E peek()`
Liefert das Element am Kopf der Queue, ohne es zu entfernen. Liefert `null` bei einer leeren Queue.

4.5.16 Die Klasse java.util.PriorityQueue

Neben der bereits beschriebenen Klasse `LinkedList` implementiert unter anderem auch diese Klasse das Interface `Queue`:

```
public class PriorityQueue<E> extends AbstractQueue<E>
implements Serializable
```

Diese Queue speichert Elemente, die entweder eine natürliche Sortierreihenfolge aufweisen oder durch einen definierten Comparator vergleichbar sind. Kleinere Elemente werden am Kopf der Queue eingefügt. Die `null`-Referenz darf nicht gespeichert werden. Die Klasse unterstützt die folgenden Konstruktoren:

`public PriorityQueue()`
Erzeugt eine PriorityQueue mit der Initialkapazität 11 unter Verwendung der natürlichen Sortierreihenfolge der Elemente.

`public PriorityQueue(Collection<? extends E> c)`
Erzeugt eine PriorityQueue aus den Elementen von `c` und verwendet die natürliche Sortierreihenfolge.

`public PriorityQueue(int initialCapacity)`
Erzeugt eine PriorityQueue mit der angegebenen Initialkapazität unter Verwendung der natürlichen Sortierreihenfolge der Elemente.

`public PriorityQueue(int initialCap, Comparator<? super E> comparator)`
Erzeugt eine PriorityQueue mit der angegebenen Initialkapazität und verwendet zur Sortierung der Elemente den übergebenen Comparator.

`public PriorityQueue(PriorityQueue<? extends E> c)`
Erzeugt eine PriorityQueue aus `c` und verwendet die in `c` definierte Sortierreihenfolge.

`public PriorityQueue(SortedSet<? extends E> c)`
Erzeugt eine PriorityQueue aus `c` und verwendet die in `c` definierte Sortierreihenfolge.

4.6 Die Vererbungshierarchie `java.util.Map`

Eine Map speichert Schlüssel-Werte-Paare, wobei die Schlüssel eindeutig sein müssen. Das Verhalten einer Map wird durch das Interface `java.util.Map` definiert.

4.6.1 Das Interface `java.util.Map`

`public interface Map<K,V>`

Wichtige Methoden dieses Interfaces sind:

Basisoperationen

`public V put(K key, V value)`
Fügt der Map den Eintrag `<key, value>` hinzu. Wenn der Schlüssel schon existiert, so wird der alte Wert durch den neuen Wert `value` ersetzt. Liefert den alten (also den aus der Map entfernten) Wert oder `null`, wenn es zu dem Schlüssel `key` noch keinen Eintrag gibt. Optional.

`public V get(Object key)`
Liefert den zum Schlüssel `key` gehörigen Wert bzw. `null`, wenn `key` nicht gefunden wird. Liefert auch `null`, wenn der zu `key` gehörige Wert `null` ist.

`public V remove(Object key)`
Entfernt das zum Schlüssel `key` gehörige Schlüssel-Werte-Paar aus der Map. Liefert den aus der Map entfernten Wert oder `null`, wenn es zu dem Schlüssel `key` keinen Eintrag gibt.Optional.

`public boolean containsKey(Object key)`
Liefert `true`, wenn der Schlüssel `key` in dieser Map gespeichert ist.

`public boolean containsValue(Object value)`
Liefert `true`, wenn der Wert `value` in dieser Map gespeichert ist.

`public int size()`
Liefert die Anzahl der Schlüssel-Werte-Paare dieser Map.

`public boolean isEmpty()`
Liefert `true`, wenn diese Map leer ist.

Mengenoperationen

`public void putAll(Map<? extends K, ? extends V> t)`
Speichert alle Schlüssel-Wert-Paare von `t` in dieser Map.

`public void clear()`
Löscht alle Schlüssel-Werte-Paare aus dieser Map.

Views

Diese Methoden liefern bestimmte Sichten der Map. Änderungen an der Map werden in die gelieferte Sicht übernommen und umgekehrt.

`public Set<K> keySet()`
Liefert alle Schlüssel dieser Map als Set. Dabei wird ein zum konkreten Maptyp analoges Set geliefert.

`public Collection<V> values()`
Liefert alle Werte dieser Map als Collection.

`public Set<Map.Entry<K,V>> entrySet()`
Liefert alle Einträge dieser Map als Set. Dabei wird ein zum konkreten Maptyp analoges Set geliefert. Ein Eintrag ist also ein Objekt vom Typ `Map.Entry`. Dabei handelt es sich um ein `Interface`, das als inneres Interface von `java.util.Map` definiert ist. Zum Zugriff auf die Schlüssel und die Werte besitzt Map.Entry die folgenden Methoden:

```
K getKey()
V getValue()
```

Abarbeiten einer Map

Maps sind nicht Iterable und können daher nicht direkt mit einem Iterator oder einer erweiterten for-Schleife abgearbeitet werden. Zum abarbeiten einer Map gibt es zwei Möglichkeiten:

1. Man besorgt sich mit Hilfe der Methode `keySet()` alle Schlüssel als Set und arbeitet dieses Set ab.
2. Man besorgt sich mit Hilfe von `entrySet()` alle Mapeinträge als Sert vom Typ Map.Entry¡K,V¿ und arbeitet dieses Set ab.

4.6.2 Die Klasse `java.util.HashMap`

```
public class HashMap<K,V> extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

Diese Klasse stellt eine konkrete Implementierung des Interfaces `java.util.Map` dar. Sie unterstützt sowohl den Schlüssel `null` wie auch Werte `null` und stellt die folgenden Konstruktoren zur Verfügung:

`public HashMap()`
Erzeugt eine `HashMap` mit Standardwerten: Initialkapazität 16 und Loadfactor 0,75.

`public HashMap(Map<? extends K, ? extends V> m)`
Erzeugt eine `HashMap`, die alle Schlüssel-Werte-Paare von `m` enthält. Loadfactor 0,75.

`public HashMap(int initialCapacity)`
Erzeugt eine `HashMap` in der angegebenen Initialkapazität und dem Loadfactor 0,75.

`public HashMap(int initialCapacity, float loadFactor)`
Erzeugt eine `HashMap` mit den übergebenen Werten.

Beispiel zu HashMap

Das folgende Beispiel verwendet eine `HashMap`, um abzuspeichern, welche Zahlen (keys) und wie oft diese Zahlen (values) in einem Feld vom Typ `int` abgespeichert sind. Auf zweifache Art und Weise wird auch eine eine Zusammenfassung ausgegeben. Zunächst wird das KeySet in ein TreeSet gespeichert, hier werden die Schlüssel also sortiert abgearbeitet. Dann wird die Map mit Hilfe ihres EntrySets abgearbeitet, hier erhält man die gleiche Reihenfolge wie bei der Ausgabe mit Hilfe der `toString()` - Methode.

```
1  import java.util.*;
2
3  public class Maps_1 {
4    public static void main(String []args) {
5      int []f = {75, 10, 75, 13, 10, 12, 13, 75, 80, 10, 80};
6      Map<Integer,Integer> m  = new HashMap<Integer,Integer>();
7      for(int i : f) {
8        if(m.containsKey(i)) {
9          Integer v = m.get(i);
10         m.put(i, ++v);
11       }
12       else {
13         m.put(i, 1);
14       }
15     }
16     System.out.println(m);
17     // Abarbeiten mit Hilfge des KeySets
18     Set<Integer> sort = new TreeSet<>(m.keySet());
19     for(Integer i : sort) {
20       System.out.format("%3d kommt %3d mal vor.\n", i, m.get(i));
21     }
22     System.out.println("---------------------------");
23     // Abarbeiten mit Hilfe der Einträge Map.Entry
24     for(Map.Entry<Integer,Integer> e : m.entrySet()) {
25         System.out.format("%3d kommt %3d mal vor.\n", e.getKey(), e.getValue());
26     }
27   }
28 }
```

Listing 4.5: Beispiel zu HashMap

Ausgabe:

```
{80=2, 10=3, 75=3, 12=1, 13=2}
 10 kommt   3 mal vor.
 12 kommt   1 mal vor.
 13 kommt   2 mal vor.
 75 kommt   3 mal vor.
 80 kommt   2 mal vor.
---------------------------
 80 kommt   2 mal vor.
 10 kommt   3 mal vor.
 75 kommt   3 mal vor.
 12 kommt   1 mal vor.
 13 kommt   2 mal vor.
```

4.6.3 Die Klasse `java.util.LinkedHashMap`

Eine konkrete Map-Implementierung, die von `HashMap` abgeleitet ist. Die Schlüssel werden zusätzlich über eine doppelt verkettet Liste gespeichert, wodurch ihre Ordnung definiert ist. Arbeitet man die

Schlüssel über einen Iterator ab, so erhält man sie in der gleichen Reihenfolge, in der sie eingefügt wurden. Die Klasse `LinkedHashMap` ist nicht threadsafe.

4.6.4 Die Klasse `java.util.Hashtable`

Diese Klasse stellt genau die gleiche Funktionalität wie `java.util.HashMap` zur Verfügung. Sie war bereits im JDK 1.0 vorhanden und wurde für das JDK 1.2 erweitert und der Klasse `java.util.HashMap` angepasst. Der Unterschied ist, dass die Klasse `Hashtable` threadsicher ist, d.h. dass alle Methoden synchronisiert sind.

```
public class Hashtable<K,V> extends Dictionary<K,V>
implements Map<K,V>, Cloneable, Serializable
```

4.6.5 Das Interface `java.util.SortedMap`

Dieses Interface ist von `java.util.Map` abgeleitet und speichert die Schlüssel sortiert. Damit dies möglich ist, müssen alle Schlüssel vergleichbar sein, d.h. ihre Klassen müssen das Interface `java.lang.Comparable` implementieren.

```
public interface SortedMap<K,V>
extends Map<K,V>
```

Das Interface `SortedMap` stellt die folgenden Methoden zur Verfügung:

`public K firstKey()`
Liefert den ersten Schlüssel der SortedMap.

`public K lastKey()`
Liefert den letzten Schlüssel der SortedMap.

`public SortedMap<K,V> headMap(K toKey)`
Liefert alle Schlüssel-Werte-Paare als SortedMap, deren Schlüssel echt kleiner als `toKey` sind.

`public SortedMap<K,V> tailMap(K fromKey)`
Liefert alle Schlüssel-Werte-Paare als SortedMap, deren Schlüssel größer oder gleich als `fromKey` sind.

`public SortedMap<K,V> subMap(K fromKey, K toKey)`
Liefert alle Schlüssel-Werte-Paare als SortedMap, deren Schlüssel größer oder gleich `fromKey` und echt kleiner als `toKey` sind.

`public Comparator comparator()`
Liefert den zugrundeliegenden Comparator oder `null`, wenn die natürliche Reihenfolge verwendet wird.

4.6.6 Die Klasse `java.util.TreeMap`

Eine konkrete SortedMap-Implementierung.

```
public class TreeMap<K,V> extends AbstractMap<K,V>
implements SortedMap<K,V>, Cloneable, Serializable
```

Konstruktoren

`public TreeMap()`
Erzeugt eine leere TreeMap, in der die Schlüssel in natürlicher Reihenfolge gespeichert werden.

`public TreeMap(Comparator<? super K> c)`
Erzeugt eine TreeMap, wobei die Schlüssel mit Hilfe des Comparators `c` verglichen werden.

`public TreeMap(Map<? extends K, ? extends V> m)`
Erzeugt eine TreeMap, in dem alle Schlüssel-Werte-Paare der Map `m` gespeichert sind. Verglichen wird nach der natürlichen Reihenfolge.

`public TreeMap(SortedMap<K,? extends V> m)`
Erzeugt eine TreeMap, die dieselben Elemente wie `m` enthält. Verglichen wird nach der durch `m` festgelegten Reihenfolge.

4.7 Die Klassen `Arrays` und `Collections`

Diese Klassen aus dem Paket `java.util` enthalten nur statische Methoden, die den Umgang mit Feldern und Objekten vom Typ `Collection` erleichtern. Beide Klassen erben direkt von `java.lang.Object`.

4.7.1 Wichtige Methoden in `java.util.Arrays`

Die Klasse `Array` beinhaltet mehrfach überladene Methoden zum Sortieren von Feldern. Die wichtigsten davon sind:

`public static void sort(Object[] a)`
Sortiert das Feld `a` nach der natürlichen Reihenfolge der enthaltenen Elemente. Alle Elemente müssen das Interface `Comparable` implementieren und untereinander vergleichbar sein. Andernfalls wird eine `ClassCastException` geworfen.

`public static <T> void sort(T[] a, Comparator<? super T> c)`
Sortiert das Feld `a` und verwendet dabei die in `c` definierte Sortierreihenfolge.

In sortierten Feldern kann auch binär gesucht werden:
Alle `binarySearch()`-Methoden suchen den Schlüssel `key` im Feld `a`. Existiert der Schlüssel im Feld, so wird der Index der Übereinstimmung geliefert, andernfalls der Wert `(-insert - 1)`, wobei `insert` den Index des ersten Elementes angibt, das größer als `key` ist. Ist das Feld unsortiert, so liefern die Methoden `binarySearch()` undefinierte Ergebnisse.

`public static int binarySearch(Object[] a, Object key)`
Sucht nach oben beschriebener Vorgangsweise den Schlüssel `key` im Feld `a`. Alle Elemente inklusive `key` müssen das Interface `Comparable` implementieren und untereinander vergleichbar sein. Andernfalls wird eine `ClassCastException` geworfen.

```
public static <T> int binarySearch(T[] a, T key,
                                   Comparator<? super T> c)
```
Sucht im über `c` sortierten Feld `a` nach `key`. Ist `c` gleich `null`, so wird die natürliche Reihenfolge der Elemente vorausgesetzt.

Diese Methoden gibt es auch in Versionen für alle primitiven Typen. Beispiele:
```
public static int binarySearch(int[] a, int key)
public static int binarySearch(float[] a, float key)
```

Ein Feld kann auch in eine Liste fixer Größe verwandelt werden:

`public static <T> List<T> asList(T... a)`
Dabei handelt es sich bei der erzeugten Liste um eine View auf das Array, d.h. Änderungen in der Liste wirken sich auch auf das Feld aus.

Die Klasse `Arrays` beinhaltet für alle Felder primitiver Typen und für Felder vom Typ `Object` überladene Methoden, mit denen die Inhalte eines Feldes bequem in einen String verwandelt werden können. Beispiele:

```
public static String toString(double[] a)
public static String toString(Object[] a)
```

Das folgende Beispiel demonstriert das Arbeiten mit den oben angegebenen Methoden:

```
1  import java.util.Arrays;
2  import java.util.Comparator;
3  import java.util.List;
4
5  public class ArraysTest {
6
7      public static void main(String []args) {
8        Integer [] x = { 10, 6, 17, 20, 3, 15 };
9
10       System.out.println(Arrays.toString(x));
11       Arrays.sort(x);
12       System.out.println(Arrays.toString(x));
13       System.out.println(Arrays.binarySearch(x,17));
14       System.out.println(Arrays.binarySearch(x,18));
15       Arrays.sort(x, new Comparator<Integer>() {
16         public int compare(Integer o1, Integer o2) {
17           return o2.compareTo(o1);
18         }
19       });
20       System.out.println(Arrays.toString(x));
21       List<Integer> l = Arrays.asList(x);
22       l.set(3,100);
23       System.out.println(Arrays.toString(x));
24       l.add(15);
25     }
26 }
```

Listing 4.6: Methoden in `java.util.Arrays`

Ausgabe:

```
[10, 6, 17, 20, 3, 15]
[3, 6, 10, 15, 17, 20]
4
-6
[20, 17, 15, 10, 6, 3]
[20, 17, 15, 100, 6, 3]
Exception in thread "main" java.lang.UnsupportedOperationException
        at java.util.AbstractList.add(AbstractList.java:151)
        at java.util.AbstractList.add(AbstractList.java:89)
        at ArraysTest.main(ArraysTest.java:24)
Press any key to continue...
```

Die Methode `add()` in Zeile 24 wirft eine `UnsupportedOperationException`, weil die Liste fixer Größe die Methode `add()` nicht unterstützt.

4.7.2 Wichtige Methoden in `java.util.Collections`

Die Klasse `Collections` beinhaltet mehrfach überladene Methoden zum Sortieren von Listen. Die wichtigsten davon sind:

`public static <T extends Comparable<? super T>> void sort(List<T> list)`
Sortiert die Liste `list` nach der natürlichen Reihenfolge der enthaltenen Elemente. Alle Elemente müssen das Interface `Comparable` implementieren und untereinander vergleichbar sein. Andernfalls wird eine `ClassCastException geworfen`.

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```
Sortiert die Liste `list` und verwendet dabei die in `c` definierte Sortierreihenfolge.

In sortierten Listen kann auch binär gesucht werden. Die Funktionen arbeiten analog zum binären Suchen in Arrays:

```
public static <T> int binarySearch(List<? extends Comparable<? super T>>
                                   list, T key)
```
Sucht den Schlüssel `key` in der Liste `list`. Alle Elemente inklusive `key` müssen das Interface `Comparable` implementieren und untereinander vergleichbar sein. Andernfalls wird eine `ClassCastException` geworfen.

```
public static <T> int binarySearch(List<? extends T> list, T key,
                                   Comparator<? super T> c)
```
Sucht in der über `c` sortierten Liste `list` nach `key`. Ist `c` gleich `null`, so wird die natürliche Reihenfolge der Elemente vorausgesetzt.

Weitere Methoden sind:

```
public static void reverse(List<?> list)
```
Dreht die Reihenfolge der Elemente in der angegebenen Liste um.

```
public static <T> Comparator<T> reverseOrder()
```
Liefert einen `Comparator`, der die natürliche Sortierreihenfolge der zu sortierenden Elemente umdreht. So sortiert z.B. der Aufruf
`Arrays.sort(a, Collections.reverseOrder());`
das Feld `a` in der zur natürlichen Sortierreihenfolge des Typs `a` entgegengesetzten Reihenfolge.

```
public static <T> Comparator<T> reverseOrder(Comparator<T> cmp)
```
Diese Methode liefert einen Comparator, die die Sortierreihenfolge des übergebenen Comparators `cmp` umdreht.

4.8 Ergänzungen zu Generics

Die folgenden Beispiele verwenden die folgende Vererbungshierarchie:

```
class Animal {}
class Cat extends Animal {}
class Dog extends Animal {}
```

4.8.1 Invarianz

Bei Verwendung von generischen Klassen ist z.B. die folgende Zuweisung nicht erlaubt:

```
List<Animal> lst = new LinkedList<Cat>();  // Compilerfehler
```

In Bezug auf den Typparameter verhält sich die Zuweisungsverträglichkeit bei den Generics in Java also invariant, d.h. der Typparameter des linken Ausdrucks (oben `Animal`) muss exakt mit jenem des rechten Typparameters (oben `Cat`) übereinstimmen. Die Begründung für diese Regel kann leicht erklärt werden. Wäre die obige Zuweisung möglich, so würde der folgende Abschnitt compilieren:

```
List<Cat> catList = new LinkedList<Cat>();
List<Animal> list = catList;    // angenommen erlaubt
list.add(new Dog());            // erlaubt
```

Man könnte also in eine für den Typ `Cat` definierte Liste z.B. `Dog`-Instanzen speichern, da diese im Rahmen der is-a Beziehung auch vom Typ `Animal` sind.

4.8.2 Wildcards

Das oben beschriebene Zuweisungsproblem kann durch Benutzung der sogenannten Wildcards `<?>` umgangen werden. Hierbei legt man bei der Definition der Referenz den Typparameter nicht exakt fest, sondern verwendet eine der folgenden Schreibweisen:

- `<? extends Type>`
  Dies bedeutet, dass bei der Zuweisung eines konkreten Objektes dieses den Typparameter `Type` oder einen Subtyp von `Type` aufweisen muss. Damit nun das oben beschriebene Problem des Einfügens falscher Typen nicht möglich ist, können über eine mit `<? extends Type>` definierte Referenz keine Objekte in die Collection eingefügt werden. Lediglich das Hinzufügen von `null` ist erlaubt.

  Beispiel:

  ```
  List<Cat> catList = new ArrayList<Cat>();
  List<? extends Animal> list = catList;
  list.add(new Dog()));     // nicht erlaubt
  list.add(new Animal());   // nicht erlaubt
  list.add(new Cat());      // nicht erlaubt
  list.add(null);           // erlaubt
  ```

  Ähnlich verhält es sich beim Auslesen von Objekten per `get()`-Methode:

  ```
  List<Cat> catList = new ArrayList<Cat>();
  catList.add(new Cat());
  List<? extends Animal> list = catList;
  Cat c = list.get(0);          // nicht erlaubt
  Animal a = list.get(0);       // erlaubt
  ```

  Der Compiler kann nur garantieren, dass jedes Objekt, welches von `get()` geliefert werden kann, zuweisungskompatibel zu `Animal` ist. Die Schreibweise `<?>` ist eine Kurzschreibweise für `<? extends Object>`.

- `<? super Type>`
  Dies bedeutet, dass bei der Zuweisung eines konkreten Objektes dieses den Typparameter `Type` oder einen Supertyp von `Type` aufweisen muss. Hier liefern `get()` - Methoden prinzipiell `Object` und `add()` - Methoden gestatten nur Argumente vom Typ `Type` (oder Subtypen) bzw. `null`. Die Begründung lässt sich leicht aus dem folgenden Beispiel ablesen:

  Beispiel:

  ```
  List<Animal> list = new ArrayList<Animal>();
  list.add(new Cat());
  List<? super Cat> list1 = list;

  Cat c = list1.get(0);      // nicht erlaubt
  Animal a = list1.get(0);   // nicht erlaubt
  Object obj = list1.get(0); // erlaubt

  list1.add(obj); // nicht erlaubt
  list1.add(a);   // nicht erlaubt
  list1.add(c);   // erlaubt
  ```

  Zunächst wird eine Liste `list` erzeugt, in die `Animal` bzw. Subtypen (als auch Objekte vom Typ `Cat` und `Dog`) geschrieben werden können. Konkret wird eine `Cat`-Instanz eingefügt.
  Der über den Typ `<? super Cat>` definierten Referenz `list1` können nun Listen zugewiesen werden, die mit dem Typ `Cat` oder einem Supertyp (als `Animal` oder `Object`) parametrisiert wurden. Damit kann auch `list` zugewiesen werden.
  Nun kann beim Auslesen von Objekten aus `list1` nichts garantiert werden, außer dass sie zuweisungskompatibel zu `Object` sind (dies wäre ja der allgemeinste Typ Parameter, der zugewiesen

werden könnte). Daher liefert die `get()`-Methode auch nur eine Object-Referenz.
Beim Hinzufügen von Objekten verhält es sich umgekehrt: Es werden nur noch Objekte vom Typ `Cat` (und Subtypen) akzeptiert. Im Extremfall hat ja die zugewiesene Liste als speziellen Typ den Typ `Cat`.

Die oben beschriebenen Einschränkung kann man auch direkt für den Typstellvertreter einer Klasse vornehmen. Somit schränkt man bereits den Nutzungsbereich dieser Klasse von vornherein ein. Es ist z. B. denkbar, den Typstellvertreter einer Klasse auf den Typ `java.lang.Number` einzuschränken, damit man mit den entsprechenden Objekten rechnen kann. Das folgende Beipiel demonstriert eine generische Klasse, die nur mit Objekten, die im Rahmen der is-a Beziehung vom Typ `Number` sind, arbeiten kann.

Beispiel:

```
 1. public class Add<E extends Number> {
 2.   private E x;
 3.   private E y;
 4.
 5.   public Add(E x, E y) {
 6.     this.x = x;
 7.     this.y = y;
 8.   }
 9.
10.   public double add() {
11.     return x.doubleValue() + y.doubleValue();
12.   }
13. }
```

In Zeile 1 wird für den Typparameter `E` festgelegt, dass er mindestens vom Typ `Number` oder eine Unterklasse sein muss. Daher kann in der Methode `add()` in Zeile 11 vorausgesetzt werden, dass die Referenzen `x` und `y` auf jeden Fall auf Objekte verweisen, welche die Methode `doubleValue()` besitzen, da diese bereits in `Number` definiert ist.

http://de.wikipedia.org/wiki/XML-Deklaration

Kapitel 5

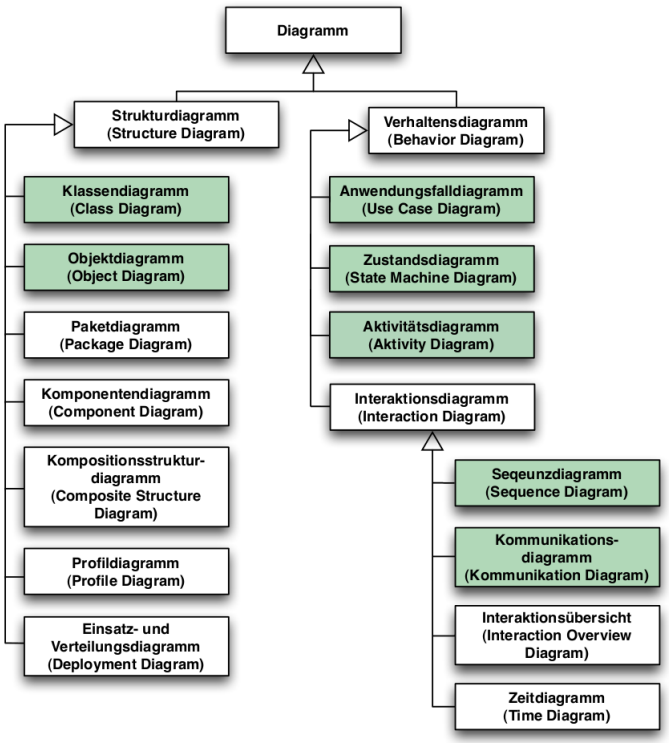
Die Unified Modeling Language - UML

5.1 Überblick

Die Unified Modeling Language (UML) ist eine Sprache und Notation zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen für Softwaresysteme. [1] Sie ist durch die Object Management Group standardisiert. Die Modelle können durch verschiedene Diagrammtypen dargestellt werden. Grundsätzlich werden zwei Hauptdiagrammtypen unterschieden:

- Strukturdiagramme (Structure Diagram)
- Verhaltensdiagramme (Behavior Diagram)

Diese fassen jeweils weitere Diagrammformen zusammen. Die Abbildung zeigt eine Übersicht:



Die in der Abbildung grün makierten Diagrammarten werden in diesem Skriptum näher beschrieben.

[1] Bernd Oestereich. Die UML 2.0 Kurzreferenz für die Praxis. Oldenburg, 2004.

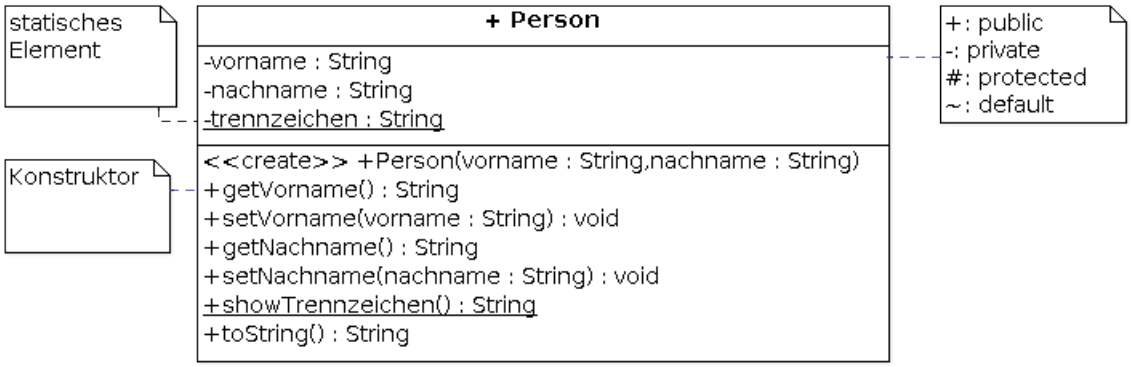
5.2 Strukturdiagramme

5.2.1 Klassendiagramm

Der Diagrammtyp Klassendiagramm dient dazu, eine oder mehrere Klassen abzubilden. Die Darstellung erfolgt dabei weitgehend unabhängig von der verwendeten Programmiersprache. Es können sowohl die Klasse(n) selbst als auch ihre Beziehungen zu anderen Klassen dargestellt werden. Die Idee hinter diesem Diagrammtyp ist es, eine übersichtliche Form zu finden, die unabhängig von Kenntnissen über Details der Programmiersprache oder des Programms zu verstehen ist. Klassendiagramme können sowohl zur Beschreibung eines schon in Code umgesetzten Programms (Reverse Engineering) als auch zur Modellierung eines Sachverhalts vor der konkreten Umsetzung in einer Programmiersprache verwendet werden. Alle in diesem Skriptum abgebildeten UML-Diagramme wurden mit dem Tool ArgoUML erzeugt.

Einfache Klassen

Das erste Beispiel zeigt das UML-Diagramm für eine einfache Klasse:



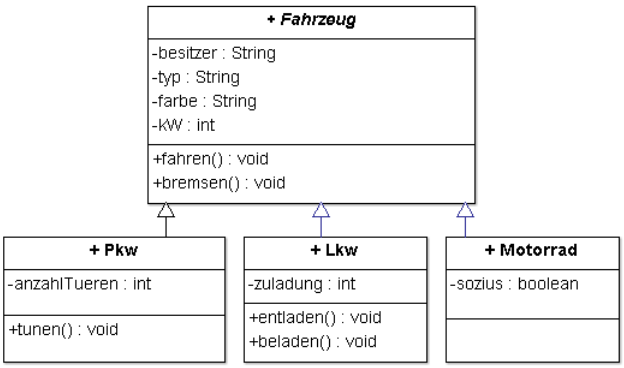
Dabei sind folgende Regeln bei der Erstellung eines solchen einfachen Klassendiagrammes zu beachten:

- Ein Klassendiagramm ist in drei Bereiche aufgeteilt. Der Kopf enthält den Namen der Klassen, er ist fett geschrieben. Ist die Klasse abstrakt, so wird der Klassenname *kursiv* gesetzt.
- Der nächste Abschnitt beinhaltet die Attribute.
- Der dritte Abschnitt beinhaltet die Methoden (das Verhalten).
- Sichtbarkeiten werden durch die Verwendung der Zeichen `+`, `-`, `#` und `~` notiert. Lässt man diese Zeichen weg, so sind Attribute automatisch `private` und Methoden `public`.
- Typen, Parameter und Namen können weggelassen werden, ebenso Attribute und Methoden. Ein Klassendiagramm kann somit auch ausschließlich aus Rechtecken mit Namen bestehen.
- Statische Attribute oder Methoden werden durch Unterstreichen gekennzeichnet.
- Kommentare werden durch Kommentarrechtecke mit einer gestrichelten Bezugslinie gekennzeichnet.
- In geschweiften Klammern können zusätzliche Eigenschaften (z.B. Implementierungsdetails) angegeben werden.
- **Stereotypen** werden durch spitze Klammern gekennzeichnet. Sie dienen dazu den Verwendungskontext zu spezifizieren. In obigem Beispiel ist der Konstruktor durch das Stereotyp `<<create>>` gekennzeichnet.

Klassendiagramme mit Vererbung

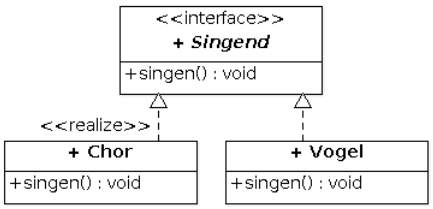
Klassendiagramme können auch Vererbungsbeziehungen zwischen Klassen abbilden. Eine Vererbungsbeziehung wird durch eine Linie, die am Ende eine ungefüllte Pfeilspitze besitzt, dargestellt. Man spricht in diesem Zusammenhang von Generalisierung und Spezialisierung. Vererbungsbeziehungen werden im englischen Sprachgebrauch auch als ”is-a” bezeichnet. Der speziellere Typ (also die Subklasse) erfüllt auch alle Eingenschaften des generelleren Typs (der Superklasse).

Das folgende Beispiel zeigt eine in UML notierte Vererbungshierarchie. Dabei erben von der abstrakten Superklasse `Fahrzeug` die drei Subklassen `Pkw`, `Lkw` und `Motorrad`.



Bei der Darstellung eines Interfaces wird das Stereotyp `<<interface>>` verwendet. Die Vererbungslinien sind strichliert (veranschaulichen also `implements`) und können zusätzlich durch das Stereotyp `<<realize>>` gekennzeichnet werden.

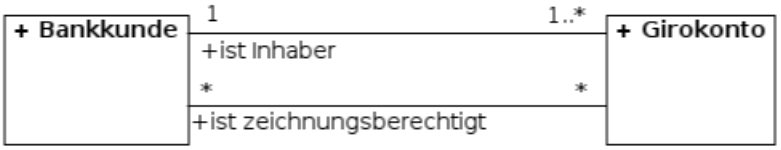
Beispiel:



Klassendiagramme mit Assoziationen

Klassendiagramme können neben Klassen und Kommentaren auch Assoziationen enthalten. Assoziationen bilden die Beziehungen zwischen Klassen ab. Zur genaueren Spezifizierung können sie (müssen aber nicht) mit Kardinalitäten versehen werden. Eine Kardinalität bezeichnet die Anzahl der zulässigen Objekte, die an der Assoziation teilnehmen. Die Kardinalität 1 kann auch weggelassen werden. Neben der Kardinalität können Assoziationen auch durch einen Namen gekennzeichnet werden. Dies ist sinnvoll, wenn zwischen zwei Objekten zwei unterschiedliche Beziehungen bestehen.

Beispiel:



Das obige Beispiel zeigt zwei Assoziationen zwischen einem Bankkunden und einem Girokonto:

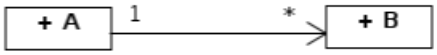
- Jedes Girokonto hat genau einen Inhaber und jeder Bankkunde hat ein bis beliebig viele Girokonten.
- Jedes Girokonto hat 0 bis beliebig viele Zeichnungsberechtigte und jeder Bankkunde ist auf 0 bis beliebig vielen Girokonten zeichnungsberechtigt.

Die UML kennt folgende Arten von Kardinalitäten:

Kardinalität	Bedeutung
`1`	Kommunikation mit genau einem Element der verbunden Klasse
`0..1`	Kommunikation mit keinem oder einem Element der verbunden Klasse
`*`	Kommunikation mit keinem bis beliebig vielen Elementen der verbunden Klasse
`n..*`	Kommunikation mit `n` bis beliebig vielen Elementen der verbunden Klasse
`n`	Kommunikation mit genau `n` Elementen der verbunden Klasse

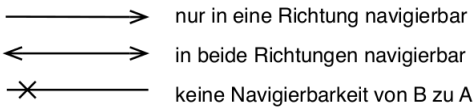
Navigierbarkeit

Die Assoziationslinien können auch mit Pfeilen versehen werden. Dabei bedeutet ein Pfeil, dass das assoziierte Objekt direkt über eine Referenz erreichbar sein muss. So zeigt z.B. das Diagramm



dass die 0 bis beliebig vielen Objekte vom Typ `B`, mit denen `A` kommuniziert, direkt über eine Referenz von `A` aus erreichbar sein müssen. Dies kann z.B. dadurch erreicht werden, das `A` eine Instanzvariable vom Typ `Collection<B>` besitzt.

Die folgende Grafik zeigt alle Möglichkeiten der Navigierbarkeit. Man spricht auch von unidirektionalen bzw. bidirektionalen Beziehungen.



Die einfache Linie trifft keine Aussagen über die Existenzabhängigkeit der Objekte dieser Klassen. Eine Pfeilspitze an einem oder beiden Enden der Assoziation zeigt an, dass die Objekte der anderen Klasse direkt über eine Referenz erreichbar sind. Befindet sich z.B. an der rechten Seite der Assoziation eine Pfeilspitze, kann von einem Objekt des Typs `A` ein Objekt des Typs `B` erreicht werden. Zur Navigierbarkeit vom Objekt desTyps `B` zum Objekt des Typs `A` wird keine Aussage getroffen. Möchte man ausdrücken, dass `A` von `B` aus nicht direkt durch eine Referenz erreicht werden darf, muss dies durch ein Kreuz kenntlich gemacht werden.

Aggregation und Komposition

- **Aggregation**
  Eine Aggregation drückt aus, dass Objekte der Klasse `B` Teil der Klasse `A` sind, `B`-Instanzen aber auch allein existieren können. Eine Aggregation wird durch eine ungefüllte Raute ausgedrückt.

- **Komposition**
  Eine Komposition drückt aus, dass Objekte der Klasse `B` Teil der Klasse `A` sind und nur dann existieren können, wenn die zugehörige Instanz der Klasse `A` existiert. Eine Komposition wird durch eine gefüllte Raute ausgedrückt.

Übung - Universität

1. **Beschreibung des Systems**
   Eine Universität besteht aus mehreren Fakultäten. Jede Fakultät besteht aus Departments. In jedem Department werden pro Semester Kurse angeboten. Jeder Kurs hat einen Dozenten und Studenten können sich über eine RegistrierungsBüro für die Kurse registrieren. Dieses System ist mit Hilfe eines UML-Diagramms abzubilden.

2. **Definieren der Entitis**
   Zur Bestimmung der Entitis kann man die Hauptworte in der Beschreibung heranziehen und dann für jedes Hauptwort prüfen, ob es sich als Entity eignet:

   - Fakultaet
   - Department
   - Kurs
   - Semester
   - Dozent
   - Student
   - RegistrierungsBuero

3. **Definition der Attribute und der Operationen** Im nächsten Schritt definiert man die Attribute und Operationen für jede Klasse. So hat z.B. ein Student Name, Geburtsdatum und ein Eintrittsjahr. Ein Kurs hat einen Namen und ECTS-Punkte.

   Diese Informationen sind in folgender Tabelle zusammengefasst:

Klassenname	Attribute	Verhalten
`Fakultaet`	`name:String`	`addDepartment(dep: Department) : void`
`Department`	`name:String`	`addKurs(kurs: Kurs) : void`
`Kurs`	`name:String` `ecds: Integer`	
`Semester`	`jahr: Integer` `saison: String`	`registriereKurs(kurs: Kurs,` ` dozent: Dozent) : void`

`Dozent`	`name: String`
	`geboren: Date`
	`raum: String`	

`Student`	`name: String`
	`geboren: Date`
	`regJahr: Integer`	

`RegBuero`		`startRegistrierung() : void`
		`stopRegistrierung()`
		`registriereStudent(student: Student,`
		` kurs: Kurs, semester: Semester) : void`

4. **Definition der Assoziationen**
   Im nächsten Schritt müssen die Beziehungen zwischen den Klassen definiert werden. So hat eine Fakultät ein oder mehrere Departments, ein Department hat keinen bis beliebig viele Kurse.

Definieren Sie diese Assoziationen und erstellen Sie ein UML-Diagramm mit Hilfe von ARGO-Uml.

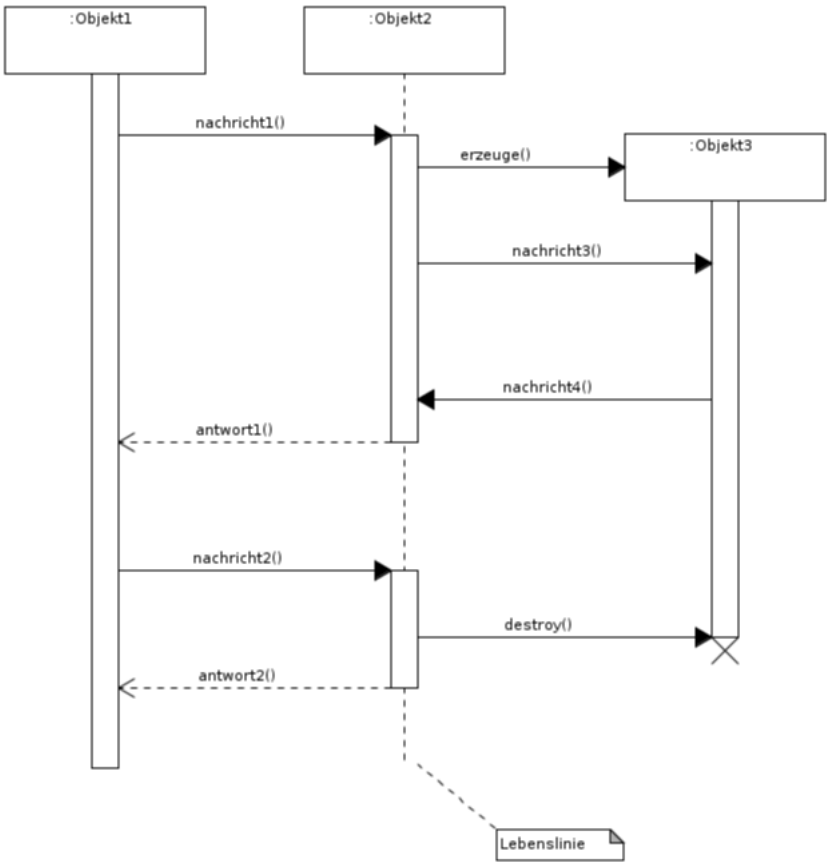
5.3 Verhaltensdiagramme

Verhaltensdiagramme repräsentieren dynamische Sachverhalte innerhalb eines objektorientierten Modells. In diesem Abschnitt werden die wichtigsten Verhaltensdiagramme vorgestellt.

5.3.1 Sequenzdiagramme

Sequenzdiagramme veranschaulichen Interaktionen zwischen Objekten im Zeitverlauf. Sie besitzen in der Vertikalen eine zeitliche Dimension, in der Horizontalen werden alle an der Interaktion beteiligten Objekte dargestellt.
Sequenzdiagramme dürfen also beteiligte Objekte und weitere Akteure beinhalten. Die Beteiligten kommunizieren über Nachrichten. Sie werden durch ein Rechteck mit Beschriftung, an dem unten eine vertikale, gestrichelte Linie angebracht ist, dargestellt. Die Gesamtheit aus Rechteck und gestrichelter Linie wird als Lebenslinie bezeichnet.

Die folgende Grafik zeigt ein einfaches Sequenzdiagramm:

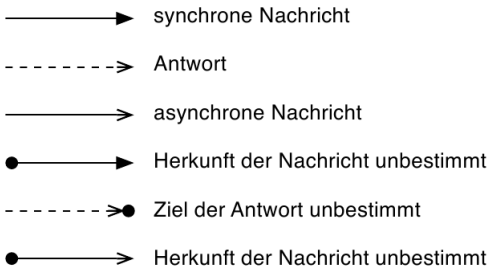


Aus der obigen Grafik lassen sich die folgenden Eigenschaften eines Sequenzdiagrammes erkennen:

- Objekte werden in Sequenzdiagrammen durch Rechtecke dargestellt.
- Zu jedem Objekt gehört eine senkrechte strichlierte Linie, die den Zeitablauf veranschaulicht. Diese Linie symbolisiert die Lebensdauer eines Objektes von der Erzeugung bis zur Zerstörung (Objektlebenszyklus).
- Senkrechte Balken, welche eine Lebenslinie überlagern, geben an, dass das Objekt gerade aktiv ist. Diese Rechtecke werden auch als Steuerungsfokus bezeichnet. In der Regel wird dadurch beschrieben, dass gerade eine Instanzmethode arbeitet, die von einem anderen Objekt aufgerufen wurde (Nachricht). Hat ein Objekt eine Methode beendet, so sendet es eine Nachricht (Returnwert) an das aufrufende Objekt.

- Es gibt auch Nachrichten, die ein Objekt erzeugen und solche, die ein Objekt zerstören und damit den Objektlebenszyklus beenden. Das Ende eines Objektlebenszyklus wird durch ein Kreuz veranschaulicht.

Im Folgenden werden die wichtigsten Pfeilarten dargestellt. Eine synchrone Nachricht besitzt eine schwarze, gefüllte Pfeilspitze und wird mit einer Antwort (ungefüllte Pfeilspitze, strichlierte Linie) quittiert. Synchron bedeutet in diesem Zusammenhang, dass die aufrufende Lebenslinie so lange blockiert, bis sie eine Antwort bekommt. Die Angabe des Antwortpfeils ist optional. Sie ist aber üblich, wenn die Methode einen Wert zurück liefert.
Eine asynchrone Nachricht wird durch eine ungefüllte Pfeilspitze an einer durchgehenden Linie dargestellt. Asynchron bedeutet, dass der Sender der Nachricht nicht auf eine Antwort wartet und nicht blockiert. Die unteren drei Pfeile sind von ihrer Bedeutung her identisch, symbolisieren aber, dass die Herkunft bzw. das Ziel der Nachricht unbestimmt ist.



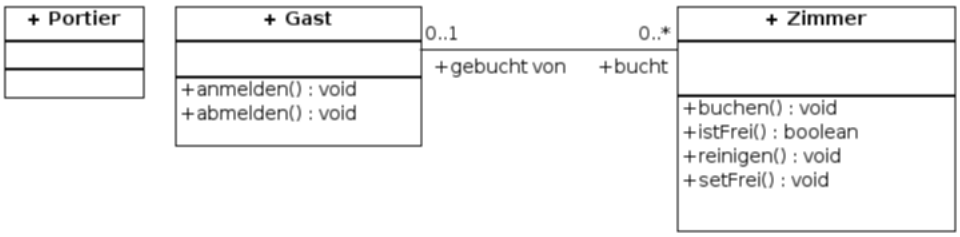
Beim Arbeiten mit Sequenzdiagrammen sind die folgenden Regeln zu beachten:

1. Die Reihenfolge der Objekte in einem Sequenzdiagramm ist unwesentlich und kann so gewählt werden, dass sich eine möglichst übersichtliche Darstellung ergibt.
2. Sequenzdiagramme zeigen in der Regel nicht die Kommunikation zwischen konkreten Objekten. Vielmehr handelt es sich um die Darstellung beliebiger Instanzen einer bestimmten Klasse bzw. um deren Rollen.
3. Sequenzdiagramme und Klassendiagramme eines Objektmodells müssen übereinstimmen. Ein Sequenzdiagramm darf daher nur solche Nachrichten enthalten, die auch in den Klassendiagrammen als Methoden definiert wurden.

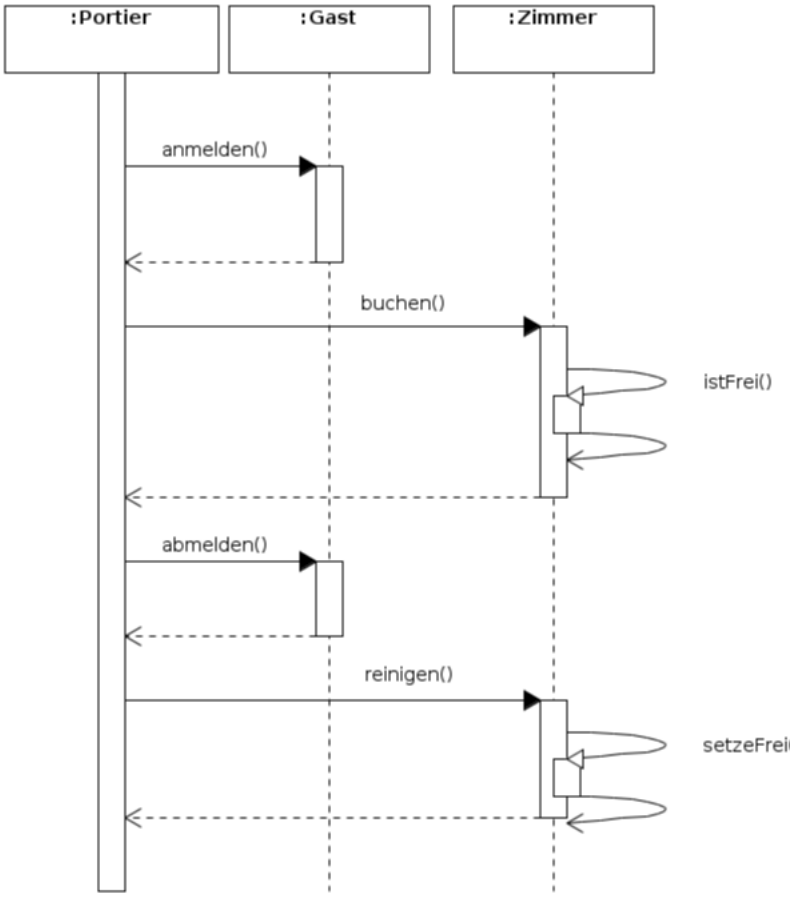
Beispiel

Das folgende Beispiel zeigt das Klassen- und Sequenzdiagramm zu einem einfachen Buchungsvorgang in einem Hotel.

Klassendiagramm:



Sequenzdiagramm:



Kapitel 6

Entwurfsmuster

6.1 Was sind Entwurfsmuster

Ein Entwurfsmuster (engl. design pattern) beschreibt eine bewährte Lösungsschablone für ein Entwurfsproblem. Es stellt damit eine wiederverwendbare Vorlage zur Problemlösung dar, die in einem spezifischen Kontext einsetzbar ist. Entstanden ist der Ausdruck in der Architektur, von der er für die Softwareentwicklung übernommen wurde.

6.1.1 Einteilung nach der GangOfFour

Ein Standardwerk zu Designpattern ist:

Design Patterns. Elements of Reusable Object-Oriented Software.
1994 von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides

Die 4 Autoren haben Designpattern kategorisiert und werden in diesem Zusammenhang nur als GroupOfFour (Viererbande) bezeichnet. Ihre Einteilung lautet:

- **Erzeugermuster**
  FactoryMethod, AbstractFactory, Builder, Prototype, Singleton
- **Strukturmuster**
  Adapter, Bridge, Composite, Decorator, Facade, Proxy, Flyweight
- **Verhaltensmuster**
  Observer, Interpreter, Template Method, Iterator, Strategy, ...

Eine gute Startseite zum Studium von Designpattern findet man unter Design Pattern

6.2 Erzeugermuster

6.2.1 Singleton

Das Design Pattern **Singleton** (Einzelstück) verhindert, dass von einer Klasse mehr als ein Objekt erzeugt werden kann. Es wird durch das folgende UML-Diagramm beschrieben:

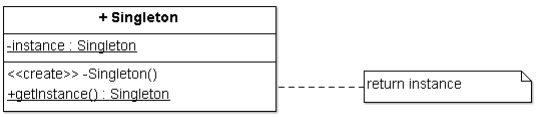


Abbildung 6.1: Singleton

Funktionsweise

Ein Singleton hat einen privaten Konstruktor. Anstatt selbst eine neue Instanz durch Aufruf des Konstruktors zu erzeugen, müssen sich Benutzer der Singleton-Klasse eine Referenz auf die eindeutige Instanz mit Hilfe der statischen Methode `getInstance()` besorgen. Diese Methode kann nun sicherstellen, dass bei jedem Aufruf eine Referenz auf dieselbe und einzige Instanz - gehalten in einer versteckten Klassenvariable - zurückgegeben wird. Entweder wird diese Instanz von `getInstance()` beim ersten Aufruf (lazy creation) oder bereits beim Laden der Klasse (eager creation) erzeugt.

Codebeispiel

```
1  public final class Singleton {
2
3    // Eine (versteckte) Klassenvariable vom Typ der eigenen Klasse
4    private static Singleton instance;
5    // Verhindere die Erzeugung des Objektes von aussen
6    private Singleton () {}
7    // Eine Zugriffsmethode auf Klassenebene, welches einmal ein
8    // konkretes Objekt erzeugt und dieses zurueckliefert
9    public static synchronized Singleton getInstance () {
10     if (Singleton.instance == null) {
11       Singleton.instance = new Singleton ();
12     }
13     return Singleton.instance;
14   }
15 }
```

Listing 6.1: Implementierung des Musters Singleton

In obigem Beispiel wird die Singleton-Instanz lazy erzeugt, erst wenn sie benötigt wird, d.h. beim ersten Aufruf der Methode `getInstance()`. In diesem Fall ist die Methode `getInstance()` auch zu synchronisieren, um Race-Conditions zu vermeiden.

6.2.2 Simple Factory

Bei einer einfachen Fabrik (Simple Factory) handelt es sich um kein GoF-Entwurfsmuster, sondern um ein Programmieridiom [1]. Es wird durch das folgende UML-Diagramm beschrieben:

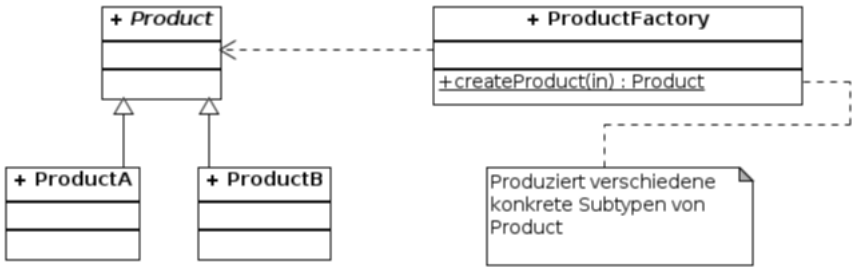


Abbildung 6.2: Simple Factory

Funktionsweise

Product ist eine (abstrakte) Basisklasse für die Klassen `ProductA` und `ProductB`. Die statische Methode `createProduct()` der Klasse `ProductFactory` entscheidet in Abhängigkeit der übergebenen Argumente, welches konkrete Produkt sie erzeugt und liefert.

[1] Softwaretechnik

Codebeispiel

```
1  public abstract class Product {
2    // Definition abstrakter Methoden
3    // Implementierung gemeinsamer Methoden
4  }
5
6  public class ProductA extends Product {
7    // Konkrete Implementierung von Produkt
8  }
9
10 public class ProductB extends Product {
11   // Konkrete Implementierung von Produkt
12 }
13
14 public class ProductFactory {
15   public static Product createProduct(char typ) {
16     switch(typ) {
17       case 'A': return new ProductA();
18       case 'B': return new ProductB();
19       default:  return null;
20     }
21   }
22 }
```

Listing 6.2: Implementierung des Idioms SimpleFactory

Bemerkung: Im obigen sowie in den folgenden Beispielen wird aus Gründen der Übersichtlichkeit für alle beteiligten Klassen jeweils nur ein Listing angegeben. Dies ist aus technischen Gründen gar nicht möglich, weil es in Java pro Quelldatei nur eine öffentliche Klasse geben darf. In konkreten Anwendungen sollte für jede Klasse / jedes Interface eine eigene Quelldatei geschrieben werden.

6.2.3 Fabriksmethode (Factory method)

Funktionsweise

Das Pattern **Factory Method** (Fabrikmethode) beschreibt, wie ein Objekt durch Aufruf einer Methode anstatt durch direkten Aufruf eines Konstruktors erzeugt wird. Dabei wird als Schnittstelle zur Erstellung eines Objektes eine abstrakte Methode einer Superklasse verwendet. Die konkrete Implementierung der Erzeugung neuer Objekte findet dabei nicht in der Superklasse statt, sondern in davon abgeleiteten Subklassen, welche die besagte abstrakte Methode implementieren.

Das Muster beschreibt somit die Erzeugung von Produktobjekten, deren konkreter Typ von Unterklassen einer Erzeugerklasse bestimmt wird. und der ein Untertyp einer abstrakten Produktklasse ist.

Das folgende Klassendiagramm zeigt die vier am Entwurfsmuster beteiligten Rollen: Konkreter Erzeuger erbt die Fabrikmethode von Erzeuger und implementiert sie so, dass sie KonkretesProdukt erzeugt, das wiederum Produkt implementiert.
UML-Diagramm:
`Produkt` ist der abstrakte Basistyp (Klasse oder Schnittstelle) für das zu erzeugende Produkt. Der Erzeuger deklariert die Fabrikmethode, um ein solches Produkt zu erzeugen und kann eine Default-Implementation beinhalten. Mitunter wird für die Fabrikmethode eine Implementierung vorgegeben, die ein ”Standard-Produkt” erzeugt.

`KonkretesProdukt` implementiert die Produkt-Schnittstelle. (Es ist also ein konkreter Subtyp von `Produkt`). `KonkreterErzeuger` überschreibt die Fabrikmethode, um die ihm entsprechenden konkreten Produkte zu erzeugen (z. B. indem er den Konstruktor einer konkreten Produkt-Klasse aufruft).

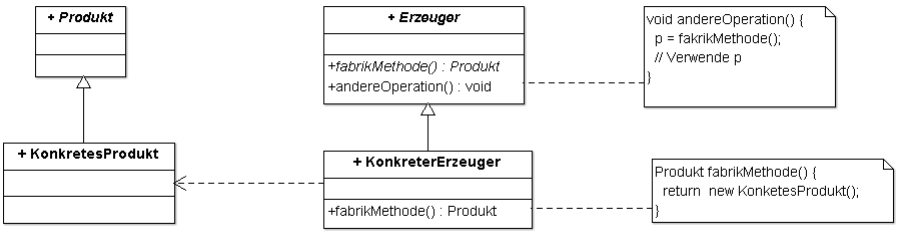


Abbildung 6.3: Fabrikmethode

Verwendung

Das Muster Fabrikmethode (in der GoF-Bedeutung) findet Anwendung, wenn eine Klasse die von ihr zu erzeugenden Objekte nicht kennen kann bzw. soll, oder wenn Unterklassen bestimmen sollen, welche Objekte erzeugt werden. Typische Anwendungsfälle sind Frameworks und Klassenbibliotheken.

Beispiel:
`java.sql.Connection.createStatement()` - das erzeugte Statement verweist auf die Connection und ”lebt in dieser”.

Beispiel

Das folgende Beispiel verdeutlicht die Verwendung des Musters in einem fiktiven Framework für eine Restaurant-Software.

Ein Restaurant (Erzeuger) liefert Mahlzeiten (Produkte). Das grundsätzliche Verfahren zum Liefern einer Mahlzeit ist immer dasselbe: Bestellung aufnehmen, Mahlzeit zubereiten, Mahlzeit servieren. Mit Ausnahme der Zubereitung lassen sich diese Funktionalitäten schon in der Klasse Restaurant implementieren. Das Zubereiten (Erzeugen) ist abhängig von der Art des Restaurants: Eine Pizzeria (konkreter Erzeuger) erzeugt Pizzen (konkrete Produkte), ein Schnitzelhaus erzeugt Schnitzel.
Die Klasse `Restaurant` implementiert hierzu eine Methode `liefereMahlzeit()`, die die Mahlzeit liefert, eingeschlossen des Bestell- und Serviervorganges. Sie benutzt eine abstrakte Fabriksmethode `bereiteMahlzeit()`, welche die für das konkrete Restaurant konkrete Mahlzeit zubereitet (erzeugt. `bereiteMahlzeit()` ist die Fabrik-Methode und muss für jedes konkrete Restaurant entsprechend implementiert werden.

UML-Diagramm:

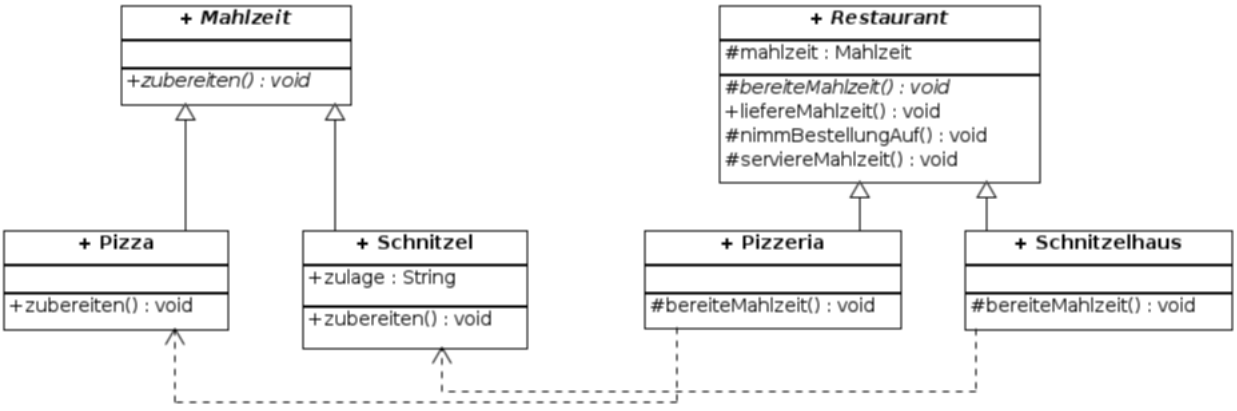


Abbildung 6.4: Fabriksmethode Beispiel

Es ist leicht einzusehen, wie dieses Framework für eine neue Art von Restaurant mit einem eigenen Mahlzeitangebot genutzt werden kann: Von Mahlzeit und von Restaurant muss jeweils eine neue Klasse

abgeleitet werden, wobei die von Restaurant abgeleitete Klasse in ihrer Methode `bereiteMahlzeit()` die in diesem Restaurant angebotene Mahlzeit zubereiten (erzeugen) muss.

```
public abstract class Mahlzeit {
  public abstract void zubereiten();
}

public class Pizza extends Mahlzeit {
  @Override
  public void zubereiten() {
    System.out.println("Pizza zubereitet");
  }
}

public class Schnitzel extends Mahlzeit {
  private String zulage;

  public Schnitzel(String zulage) {
    this.zulage = zulage;
  }

  @Override
  public void zubereiten() {
    System.out.format("Schnitzel mit %s  zubereitet\n", zulage);
  }
}
```

Listing 6.3: Produkte zu Fabrikmethode

```
public abstract class Restaurant {
  // Erzeuger
  protected Mahlzeit mahlzeit;

  protected void nimmBestellungAuf() {
    System.out.println("Ihre Bestellung bitte.");
  }

  protected abstract void bereiteMahlzeit();

  protected void serviereMahlzeit() {
    System.out.println("Hier Ihre Mahlzeit. Guten Appetit!");
  }

  public void liefereMahlzeit() {
    nimmBestellungAuf();
    bereiteMahlzeit(); // Aufruf der Factory-Methode
    serviereMahlzeit();
  }
}

// Konkreter Erzeuger fuer konkretes Produkt "Pizza"
public class Pizzeria extends Restaurant {
  protected void bereiteMahlzeit() {
    mahlzeit = new Pizza();
    mahlzeit.zubereiten();
  }
}

// Konkreter Erzeuger fuer konkretes Produkt "Schnitzel"
public class Schnitzelhaus extends Restaurant {
 protected void bereiteMahlzeit() {
    mahlzeit = new Schnitzel("Pommes und Ketchup");
    mahlzeit.zubereiten();
```

```
35   }
36 }
```

Listing 6.4: Erzeuger zu Fabrikmethode

```
1  public class Fabrikmethode {
2    public static void main(String[] args) {
3      Restaurant daToni = new Pizzeria();
4      daToni.liefereMahlzeit();
5
6      Restaurant brunosImbiss = new Schnitzelhaus();
7      brunosImbiss.liefereMahlzeit();
8    }
9  }
```

Listing 6.5: Applikation zu Fabrikmethode

Ausgabe:

```
Ihre Bestellung bitte.
Pizza zubereitet
Hier Ihre Mahlzeit. Guten Appetit!
Ihre Bestellung bitte.
Schnitzel mit Pommes und Ketchup  zubereitet
Hier Ihre Mahlzeit. Guten Appetit!
```

6.2.4 Abstrakte Fabrik (Abstract Factory)

Funktionsweise

Eine **Abstract Factory** hat die Aufgabe, Instanzen anderer Klassen zu erzeugen, wobei der Typ der zu instanziierenden Objekte erst zur Laufzeit festgelegt werden soll.

In der Klassenhierarchie existiert dabei eine abstrakte Superklasse für die Fabrik, welche die Methoden zur Objekterzeugung definiert. Je nach Art der zu erzeugenden Objekte werden die Produkte ebenfalls mittels abstrakter Superklassen definiert. Diese Typen sind dann die Rückgabetypen der Methoden der abstrakten Fabrik, eine abstrakte Fabrik bietet also Methoden zur Erzeugung von 1..n abstrakten Produkten an.

Der Klient kümmert sich dabei nicht um die Erzeugung der speziellen Objekte, er wählt lediglich eine spezielle Fabrik aus. Diese Fabrik übernimmt dann die Funktionalität der Instanziierung, der Client verwendet nur die im abstrakten Produkt definierten Methoden.

Das folgende UML-Diagramm zeigt die Akteure des Pattern Abstract Factory:

- **AbstrakteFabrik**
  definiert eine Schnittstelle zur Erzeugung abstrakter Produkte einer Produktfamilie
- **KonkreteFabrik**
  erzeugt konkrete Produkte einer Produktfamilie durch Implementierung der Schnittstelle
- **AbstraktesProdukt**
  definiert eine Schnittstelle für eine Produktart
- **KonkretesProdukt**
  definiert ein konkretes Produkt einer Produktart durch Implementierung der Schnittstelle, wird durch die korrespondierende konkrete Fabrik erzeugt
- **Klient**
  verwendet die Schnittstellen der abstrakten Fabrik und der abstrakten Produkte, kennt die konkret erzeugten Produkte nicht

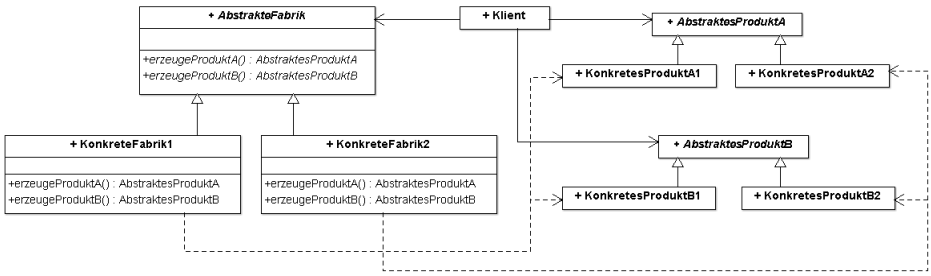


Abbildung 6.5: UML Abstract Factory

Beispiel

Das folgende Beispiel demonstriert das Pattern Abstract Factory. Es werden 2 Typen von Autos erzeugt: Normale Autos und Sportwagen. Für jeden Typ wird jeweils eine eigene Karosserie und ein eigener Motor erzeugt. Daher erzeugt die Fabrik `CarFactory` 3 Produkte: Autos (`Car`), Motoren (`Engine`) und Karosserien (`Coach`).

Die Vererbungshierarchie `CarFactory`:

```
// Abstrakte Fabrik CarFactory
public abstract class CarFactory {
    protected abstract Engine createEngine();
    protected abstract Coach createCoach();
    public abstract Car createCar();
}

// Konkrete Fabrik SportCarFactory
// Wird als Singleton realisiert
public class SportCarFactory extends CarFactory {

  private static SportCarFactory instance = new SportCarFactory();

  private SportCarFactory() {
  }

  public static SportCarFactory getInstance() {
    return instance;
  }

  @Override
  protected Engine createEngine() {
    return new SportEngine();
  }

  @Override
  protected Coach createCoach() {
    return new SportCoach();
  }

  @Override
  public Car createCar() {
    return new SportCar(createCoach(), createEngine());
  }
}

// Konkrete Fabrik NormalCarFactory
// Wird als Singleton realisiert
public class NormalCarFactory extends CarFactory {

  private static NormalCarFactory instance = new NormalCarFactory();
```

```
42
43    private NormalCarFactory() {
44    }
45
46    public static NormalCarFactory getInstance() {
47      return instance;
48    }
49
50    @Override
51    protected Engine createEngine() {
52      return new NormalEngine();
53    }
54
55    @Override
56    protected Coach createCoach() {
57      return new NormalCoach();
58    }
59
60    @Override
61    public Car createCar() {
62      return new NormalCar(createCoach(), createEngine());
63    }
64  }
```

Listing 6.6: Vererbungshierarchie `CarFactory`

Die Verebungshierarchie zum Produkt `Engine`:

```
1  // Das abstrakte Produkt Engine
2  public abstract class Engine {
3  }
4
5  // das konkrete Produkt SportEngine
6  public class SportEngine extends Engine {
7    @Override
8    public String toString() {
9      return "SportEngine";
10   }
11 }
12
13 //Das konkrete Produkt NormalEngine
14 public class NormalEngine extends Engine {
15   @Override
16   public String toString() {
17     return "NormalEngine";
18   }
19 }
```

Listing 6.7: Vererbungshierarchie `Engine`

Die Verebungshierarchie zum Produkt `Coach`:

```
1  // das abstrakte Produkt Coach
2  public abstract class Coach {
3  }
4
5  // Das konkrete Produkt SportCoach
6  public class SportCoach extends Coach {
7    @Override
8    public String toString() {
9      return "SportCoach";
10   }
11 }
12
13 // Das konkrete Produkt NormalCoach
14 public class NormalCoach extends Coach {
```

```
15   @Override
16   public String toString() {
17     return "NormalCoach";
18   }
19 }
```

Listing 6.8: Vererbungshierarchie `Coach`

Die Verebungshierarchie zum Produkt `Car`:

```
1  // das abstrakte Produkt Car
2  public abstract class Car {
3    private Coach coach;
4    private Engine engine;
5
6    public Car(Coach coach, Engine engine) {
7      this.coach = coach;
8      this.engine = engine;
9    }
10
11   @Override
12   public String toString() {
13     return "Car{" + "coach=" + coach + ", engine=" + engine + '}';
14   }
15 }
16
17 // Das konkrete Produkt SportCar
18 public class SportCar extends Car {
19   public SportCar(Coach coach, Engine engine) {
20     super(coach, engine);
21   }
22 }
23
24 // Das konkrete Produkt NormalCar
25 public class NormalCar extends Car {
26   public NormalCar(Coach coach, Engine engine) {
27     super(coach, engine);
28   }
29 }
```

Listing 6.9: Vererbungshierarchie `Car`

Die Anwendung `CarClient`:

```
1  public class CarClient {
2    public static void main(String[] args) {
3      CarFactory cf;
4      Car c;
5
6      cf = SportCarFactory.getInstance();
7      c = cf.createCar();
8      System.out.println(c);;
9      cf = NormalCarFactory.getInstance();
10     c = cf.createCar();
11     System.out.println(c);
12   }
13 }
```

Listing 6.10: Anwendung `CarClient`

Ausgabe:

```
Car{coach=SportCoach, engine=SportEngine}
Car{coach=NormalCoach, engine=NormalEngine}
```

Im Wesentlichen handelt sich beim Muster Abstrakte Fabrik um die mehrfache Anwendung des Patterns Fabriksmethode.

6.3 Verhaltensmuster

6.3.1 Observer

Dieses Pattern dient zur Weitergabe von Änderungen an einem Objekt an andere Objekte. Es ist auch unter dem Namen publish-subscribe bekannt, frei übersetzt ”veröffentlichen und abonnieren”. Es wird immer dann eingesetzt, wenn die Zustandsänderung eines Objektes (dem Subjekt `o`) andere Objekte interessiert, ohne dass das Subjekt `o` diese anderen Objekte kennen muss.

Das Observer-Pattern wird durch das folgende UML-Diagramm beschrieben:

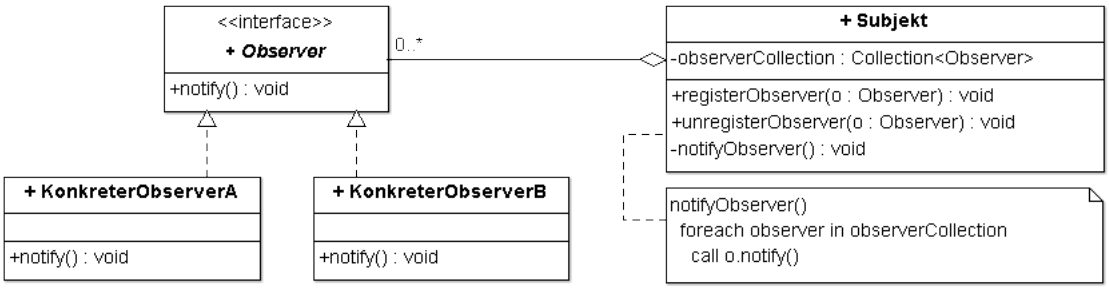


Abbildung 6.6: Observer

Ein Subjekt (beobachtbares Objekt, auch Publisher, also ”Veröffentlicher”, genannt) verwaltet eine Liste vom abstrakten Typ `Observer`, aber keine konkreten Observer. Es bietet Methoden zur An- und Abmeldung von Observern an. Das Interface `Observer` definiert z.B. die Methode `notify()`. Das beobachtbare Subjekt ruft bei einer Zustandsänderung für alle aktuell registrierten konkreten Observer diese Methode `notify()` auf.
Oft erhält die Methode `notify()` ein Event-Objekt, in dem wesentliche Information über die Zustandsänderung des Subjekts gespeichert sind.

Verwendung

Das Observer Pattern wird oft bei der Programmierung von grafischen Oberflächen verwendet. Es sorgt dafür, dass bestimmte Bereiche eines Programms, beispielsweise Elemente einer grafischen Oberfläche, die eine identische Datenquelle verwenden sich aktualisieren, sobald sich die Daten ändern (JavaFX Properties und Binding).

Ein bekanntes Beispiel ist auch das Eventhandling: So ist z.B. ein Button ein Subjekt, bei dem sich Observer registrieren können, die von einem Click auf den Button benachrichtigt werden wollen. Unter AWT und Swing heißt das Observer-Interface `ActionListener`, die Methoden zum Registrieren bzw. Abmelden sind

```
public void addActionListener(ActionListener l)
```

und

```
public void removeActionListener(ActionListener l)
```

und die Methode `notify()` entspricht der Methode `void actionPerformed(ActionEvent e)`.

Beispiel

Im folgenden Beispiel ist das Subjekt ein einfacher Zähler, der durch Aufruf der Methode `incValue()` hochgezählt wird. Der konkrete Observer `LogObserver` loggt alle Änderungen auf `System.err` und in eine Datei `counter.log`.

```java
package observer;

import java.util.ArrayList;
import java.util.List;

public class Counter {

  // Liste aller registrierten Observer
  private List<Observer> observer;

  // Die zu beobachtenden Daten
  private int curValue;

  // Konstruktor
  public Counter() {
    observer = new ArrayList<Observer>();
    curValue = 0;
  }

  // Registriert einen neuen Observer
  public void addObserver(Observer o) {
    observer.add(o);
  }

  // Entfernt den Observer o
  public void removeObserver(Observer o) {
    observer.remove(o);
  }

  // Benachrichtigt alle Observer
  public void notifyObserver() {
    for (Observer o : observer) {
      o.refresh();
    }
  }

  // Veraendert die Daten und benachrichtigt alle Observer
  public void incValue() {
    curValue++;
    notifyObserver();
  }

  // Liefert die aktuellen Daten
  public int getValue() {
    return curValue;
  }
}
```

Listing 6.11: Der Publisher Counter

```java
package observer;

/*
 * Dieses Interface stellt den anderen Observern die Methode zur Verfuegung.
 * Jeder Observer muss diese Methode implementieren. In der Implementierung
 * der refresh() - Methode kann jeder Observer mit Daten des Subjekts arbeiten.
 * Immer wenn diese sich aendern sollte refresh() aufgerufen werden.
 */

public interface Observer {
    void refresh();
}

```

```
14 //-------------------------------------------------------
15 // Ein konkreter Observer, die jede Aenderung des Zaehlers
16 // loggt
17
18 class LogObserver implements Observer {
19
20   private Counter subject;     // das zu beobachtende Objekt
21   private Logger logger;
22
23   public LogObserver(Counter subject) {
24     this.subject = subject;
25     try {
26       FileHandler handler = new FileHandler("counter.log", true);
27       handler.setFormatter(new SimpleFormatter());
28       // Ein Logger wird mit einem Namen erzeugt
29       logger = Logger.getLogger(LogObserver.class.getName());
30       logger.addHandler(handler);
31       //logger.setUseParentHandlers(false);
32     } catch (IOException | SecurityException ex) {
33       System.err.println(ex.getMessage());
34     }
35   }
36
37   @Override
38   public void refresh() {
39     String message = String.format("Neuer Wert von Counter: %d", subject.getValue())
         ;
40     logger.log(Level.INFO, message);
41   }
42 }
```

Listing 6.12: Subscriber

```
1  package observer;
2
3  public class ObserverTest {
4    private static Random rd = new Random();
5
6    public static void main(String []args) {
7      final Counter counter = new Counter();
8      final LogObserver ob1 = new LogObserver(counter);
9
10     counter.addObserver(ob1);
11
12     new Thread() {
13       @Override
14       public void run() {
15         for(int i = 1; i < 10; i++) {
16           try {
17             Thread.sleep(rd.nextInt(5000));
18           } catch (InterruptedException ex) {
19             System.out.println(ex.getMessage());
20           }
21           counter.incValue();
22         }
23       }
24     }.start();
25   }
26 }
```

Listing 6.13: Testprogramm

Ausgabe:

```
Dez 09, 2012 5:51:13 PM observer.LogObserver refresh
```

```
INFO: Neuer Wert von Counter: 1
...
Dez 09, 2012 5:51:32 PM observer.LogObserver refresh
INFO: Neuer Wert von Counter: 9
```

`java.util.Observable` und `java.util.Observer`

Die Java API bietet auch einen vorgefertigten Observer durch die Klasse `java.util.Observable`.
Diese repräsentiert den Publisher. Um einen Observer hinzuzufügen bzw. zu entfernen gibt es die Methoden `addObserver()` bzw. `deleteObserver()`. Um die Observer zu benachrichtigen benutzt man die Methode `notifyObservers()`. Die eigentlichen Observer zeichnen sich dadurch aus, dass sie das Interface `java.util.Observer` implementieren. Dazu ist es nötig, die Methode `update()` zu überschreiben. Diese entspricht der `refresh()` Methode aus dem obigen Beispiel.

Übungen

1. Ergänzen Sie das obige Beispiel um einen zweiten konkreten Observer vom Typ `JFrame`. Bei jeder Änderung des Zählerstandes soll der neue Wert auf einem JLabel dargestellt werden.
2. Ändern Sie das Musterbeispiel inklusive dem in Übung 1 erstellten Observer so ab, dass `java.util.Observable` und `java.util.Observer` Verwendung finden.

6.3.2 Strategy

Das Muster **Strategy** (Strategie) wird verwendet, um zur Laufzeit eines Programms einen bestimmten Algorithmus ändern zu können.
Das Strategy-Pattern wird durch das folgende UML-Diagramm beschrieben:

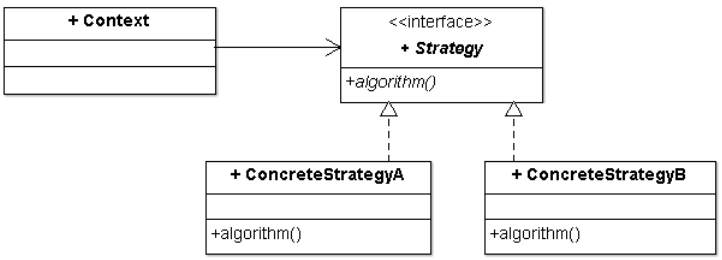


Abbildung 6.7: Strategy

Das Interface (die abstrakte Klasse) `Strategy` definiert nur eine Schnittstelle für alle unterstützten Algorithmen. Die Implementierung der eigentlichen Algorithmen finden sich erst in den Ableitungen wieder (z.B. `ConcreteStrategyA`).

Die Klasse `Context` verwaltet eine Variable zum Typ `Strategy`, die mit einer Referenz auf das gewünschte Strategieobjekt belegt ist. Auf diese Weise wird der konkrete Algorithmus über die Schnittstelle eingebunden und kann bei Bedarf selbst zur Laufzeit noch dynamisch gegen eine andere Implementierung ausgetauscht werden.

Verwendung

Strategie-Objekte werden ähnlich wie Klassenbibliotheken verwendet. Im Gegensatz dazu handelt es sich jedoch nicht um externe Programmteile, die als ein Toolkit genutzt werden können, sondern um integrale Bestandteile des eigentlichen Programms, die deshalb als eigene Objekte definiert wurden, damit sie durch andere Algorithmen ausgetauscht werden können.

Die Verwendung von Strategien bietet sich an, wenn

- viele verwandte Klassen sich nur in ihrem Verhalten (also nicht in ihrem Zustand) unterscheiden
- unterschiedliche (austauschbare) Varianten eines Algorithmus benötigt werden

Ein Beispiel ist etwa ein Packer, der verschiedene Kompressionsalgorithmen unterstützt. Dieser kann mit Hilfe des Musters Stratetgy implementiert sein.
In Java wird das Entwurfsmuster zum Beispiel zur Delegierung des Layouts von AWT-Komponenten an entsprechende LayoutManager (BorderLayout, FlowLayout etc.) verwendet.

Beispiel

Im folgenden Beispiel wird das Pattern Strategy verwendet um in einem Array mit verschiedenen Algorithmen den kleinsten Wert zu suchen.

```
1  package strategy;
2
3  public interface FindMinimum {
4    // Liefert den kleinsten Wert des Arrays array
5    double minimum(double[] array);
6  }
7
8  public class LinearSearch implements FindMinimum {
9    // Konkreter Algorithmus (lineare Suche)
10   public double minimum(double[] array) {
11     double min = array[0];
12     for(double d : array) {
13       if(d < min) min = d;
14     }
15     return min;
16   }
17 }
18
19 public class SortArray implements FindMinimum {
20   // Konkreter Algorithmus (Array sortieren)
21   public double minimum(double[] array) {
22     double [] copy = java.util.Arrays.copyOf(array, array.length);
23     java.util.Arrays.sort(copy);
24     return copy[0];
25   }
26 }
27
28 // Der "Context" dieses Strategy - Musters
29 public class MinimumContext implements FindMinimum {
30
31   private FindMinimum strategy;
32
33   public MinimumContext(FindMinimum strategy) {
34     this.strategy = strategy;
35   }
36
37   public double minimum(double[] array) {
38     return strategy.minimum(array);
39   }
40
41   public void changeAlgorithm(FindMinimum newAlgorithm) {
42     this.strategy = newAlgorithm;
43   }
44 }
45
46 public class StrategyPattern {
47   public static void main(String[] args) {
48     MinimumContext minFinder = new MinimumContext(new LinearSearch());
49
50     double[] x = {1.0, 2.0, 1.0, 2.0, -1.0, 3.0, 4.0, 5.0, 4.0};
```

```
51      System.out.println(minFinder.minimum(x));
52      // Aendern der Strategie
53      minFinder.changeAlgorithm(new SortArray());
54      System.out.println(minFinder.minimum(x));
55    }
56  }
```

Listing 6.14: Das Pattern Strategy

6.4 Strukturmuster

6.4.1 Adapter

Das Muster dient zur Übersetzung einer Schnittstelle in eine andere. Dadurch wird die Kommunikation von Klassen mit zueinander inkompatiblen Schnittstellen ermöglicht. Das Muster konvertiert also die Schnittstelle einer Klasse in eine Schnittstelle, die der Client erwartet. Adapter ermöglichen die Zusammenarbeit von Klassen, welche ohne diese nicht zusammenarbeiten könnten, weil sie inkompatible Schnittstellen haben.

Es gibt zwei Arten des Adapter Musters

- **Objekt Adapter**
  Dieser erbt von der Klasse (bzw. implementiert das Interface, die (das) der Client erwartet und wandelt ein Objekt mit einer inkompatiblen Schnittstelle in ein Objekt der geerbten Klasse um.
- **Klassen Adapter**
  Der Klassen Adapter hingegen erbt neben der Klasse, die der Client erwartet, auch von der Klasse des Objekts mit der inkompatiblen Schnittstelle und adaptiert diese. Das dieser Adapter i.A. Mehrfachvererbung benötigt, wird er hier nicht weiter beschrieben.

Das UML-Diagramm des Objekt Adapters sieht wie folgt aus:

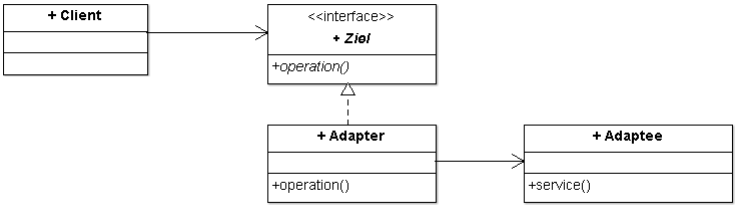


Abbildung 6.8: Object Adapter

Dabei besitzt der `Client` eine Referenz auf das Interface `Ziel`, welches die geforderte Schnittstelle definiert. Die Klasse Adapter setzt die in `Ziel` definierte Schnittstelle um, indem Sie dafür ein Objekt von `Adaptee` verwendet. `Adaptee` stellt die geforderte Funktionalität über eine andere als in Ziel definierte Schnittstelle zur Verfügung.

Beispiel

Im folgenden Beispiel wird ein Adapter vorgestellt, der die durch folgende abstrakte Klasse `MyList` definierte Schnittstelle für eine verkettete Liste erfüllt. Die Schnittstelle wird dabei als abstrakte Klasse und nicht als Interface definiert, weil einige Methoden (`isEmpty()`, `head()`) bereits eine Standardimplementierung haben.

```
1  package adapter;
2
3  // Definiert das Interface, welches der Client benoetigt
4
```

```
5  public abstract class MyList<T> {
6    // Liefert die Laenge der Liste
7    public abstract int len();
8
9    // Liefert true, wenn die Liste leer ist
10   public boolean isEmpty() {
11     return len() == 0;
12   }
13
14   // Liefert das erste Element der Liste
15   public T head() {
16     return this.at(0);
17   }
18
19   // Liefert das Element an der Position index
20   public abstract T at(int index);
21
22   // Loescht das erste Element der Liste
23   public abstract T delHead();
24
25   // Element anhaengen
26   public abstract boolean addTail(T e);
27 }
```

Listing 6.15: Adapter Schnittstelle

Als Adapter wird die folgende Klasse `ListAdapter<T>` implementiert:

```
1  package adapter;
2
3  import java.util.LinkedList;
4  import java.util.List;
5
6  // Der Adapter
7  // Erfuellt die in MyList definierte Schnittstelle und
8  // verwendet als Adaptee eine java.util.LinkedList
9
10 public class ListAdapter<T> extends MyList<T> {
11   private List<T> l = new LinkedList<>();
12
13   @Override
14   public int len() {
15     return l.size();
16   }
17
18   @Override
19   public T at(int i) {
20     return l.get(i);
21   }
22
23   @Override
24   public T delHead() {
25     T el = l.get(0);
26     return l.remove(0) ? el : null;
27   }
28
29   @Override
30   public boolean addTail(T e) {
31     return l.add(e);
32   }
33
34   @Override
35   public String toString() {
36     return l.toString();
```

```
37    }
38 }
```

Listing 6.16: Adapter Implementierung

Die Implementierung kann z.B. mit folgender Applikation getestet werden:

```
1  package adapter;
2
3  public class AdapterTest {
4
5    public static void main(String[] args) {
6      MyList<String> lst = new ListAdapter<String>();
7      lst.addTail("Eins");
8      lst.addTail("Zwei");
9      lst.addTail("Drei");
10     System.out.println(lst);
11     lst.delHead();
12     System.out.println(lst);
13   }
14 }
```

Listing 6.17: Adaptertest

Ausgabe:

```
[Eins, Zwei, Drei]
[Zwei, Drei]
```

6.4.2 Proxy

Das Muster dient zum Verschieben der Kontrolle über ein Objekt auf ein vorgelagertes Stellvertreterobjekt.

Das Proxy-Pattern wird durch das folgende UML-Diagramm beschrieben:

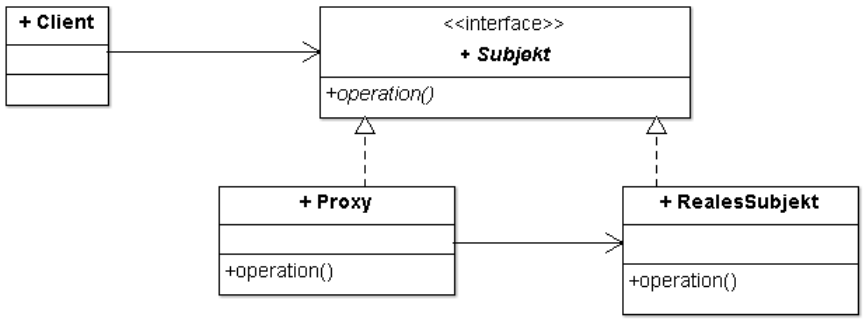


Abbildung 6.9: Proxy

Der Client stellt das Objekt dar, welches durch den Stellvertreter (Proxy) auf das reale Subjekt zugreift. Der Stellvertreter bietet nach außen hin eine zum realen Subjekt identische Schnittstelle. Er verwaltet eine Referenz auf dieses und ist eventuell auch verantwortlich für dessen Erzeugung und Löschung. Weitere Verantwortlichkeiten ergeben sich aus der Art des Stellvertreters. Das Subjekt definiert die gemeinsame Schnittstelle von Stellvertreter und realem Subjekt. Dadurch wird die Verwendung von Stellvertretern anstatt realer Subjekte möglich. Das reale Subjekt ist das durch den Stellvertreter repräsentierte Objekt.

Beispiel

In folgendem Beispiel wird ein Generator zur Berechnung von Sinuswerten implementiert. Da die Berechnung Zeit kostet, wird dem Generator ein Proxy vorgelagert, der alle bereits berechneten Sinuswerte cacht und eine Anfrage nur dann an den eigentlichen Generator weiterleitet, wenn es den Sinuswert noch nicht im Cache hat.

```
1  package proxy;
2
3  import java.util.Map;
4  import java.util.concurrent.ConcurrentHashMap;
5
6  // Das Interface
7  public interface GeneratorInterface {
8    double berechneSinus(double zahl);
9  }
10
11 // Der eigentliche Sinusgenerator
12 // Das reale Objekt
13 public class SinusGenerator implements GeneratorInterface {
14
15   public double berechneSinus(double zahl) {
16     System.out.println("SinusGenerator.BerechneSinus(" + zahl + ")");
17     return Math.sin(zahl);
18   }
19 }
20
21 // Der Stellvertreter
22 public class GeneratorProxy implements GeneratorInterface {
23
24   private SinusGenerator genobj = null;
25   private static Map<Double, Double> mem =
26           new ConcurrentHashMap<Double, Double>();
27   double speicher, zugriff;
28
29   public GeneratorProxy() {
30     System.out.println("GeneratorProxy.GeneratorProxy()");
31     genobj = new SinusGenerator();
32
33   }
34
35   public double berechneSinus(double zahl) {
36
37     System.out.println("GeneratorProxy.BerechneSinus(" + zahl + ")");
38
39     if (mem.containsKey(zahl)) {
40       System.out.println("Sinus von " + zahl + " aus Cache geholt" + "!");
41       return mem.get(zahl);
42     } else {
43       double result = genobj.berechneSinus(zahl);
44       mem.put(zahl, result);
45       System.out.println("Sinus von " + zahl + " jetzt im Cache abgelegt!");
46       return result;
47     }
48   }
49 }
50
51 // Der Client
52 public class Client {
53   public static void main(String args[]) {
54     GeneratorInterface anfrage = new GeneratorProxy();
55     double[] data = {3.01, 3.01, 4.2, 3.01, 4.2, 4.201};
56     for (double d : data) {
```

```
57        System.out.format("Sinus von %.3f: %.5f\n", d,
58                anfrage.berechneSinus(d));
59        System.out.println("---------------------------------------");
60      }
61    }
62  }
```

Listing 6.18: Adapter Implementierung

Ausgabe:

```
GeneratorProxy.GeneratorProxy()
GeneratorProxy.BerechneSinus(3.01)
SinusGenerator.BerechneSinus(3.01)
Sinus von 3.01 jetzt im Cache abgelegt!
Sinus von 3,010: 0,13121
---------------------------------------
GeneratorProxy.BerechneSinus(3.01)
Sinus von 3.01 aus Cache geholt!
Sinus von 3,010: 0,13121
---------------------------------------
GeneratorProxy.BerechneSinus(4.2)
SinusGenerator.BerechneSinus(4.2)
Sinus von 4.2 jetzt im Cache abgelegt!
Sinus von 4,200: -0,87158
---------------------------------------
GeneratorProxy.BerechneSinus(3.01)
Sinus von 3.01 aus Cache geholt!
Sinus von 3,010: 0,13121
---------------------------------------
GeneratorProxy.BerechneSinus(4.2)
Sinus von 4.2 aus Cache geholt!
Sinus von 4,200: -0,87158
---------------------------------------
GeneratorProxy.BerechneSinus(4.201)
SinusGenerator.BerechneSinus(4.201)
Sinus von 4.201 jetzt im Cache abgelegt!
Sinus von 4,201: -0,87207
---------------------------------------
```

6.4.3 Dekorierer

Das Muster ist eine flexible Alternative zur Unterklassenbildung, um eine Klasse um zusätzliche Funktionalitäten zu erweitern.
Das Dekorierer-Pattern wird durch das folgende UML-Diagramm beschrieben:
Die abstrakte Komponente definiert die öffentliche Schnittstelle für zu dekorierende Objekte. Die konkrete Komponente definiert Objekte, die dekoriert werden können.
Der abstrakte Dekorierer hält eine Referenz auf eine konkrete Komponente und bietet dieselbe Schnittstelle wie die abstrakte Komponente. Der konkrete Dekorierer definiert und implementiert eine oder mehrere spezielle Dekorationen.

Vor- und Nachteile

- **Vorteile**
  Die Vorteile bestehen darin, dass mehrere Dekorierer hintereinandergeschaltet werden können. Durch Einsatz eines Dekorierers können lange und unübersichtliche Vererbungshierarchien vermieden werden.

- **Nachteile**
  Da eine dekorierte Komponente nicht identisch mit der Komponente selbst ist (als Objekt), muss

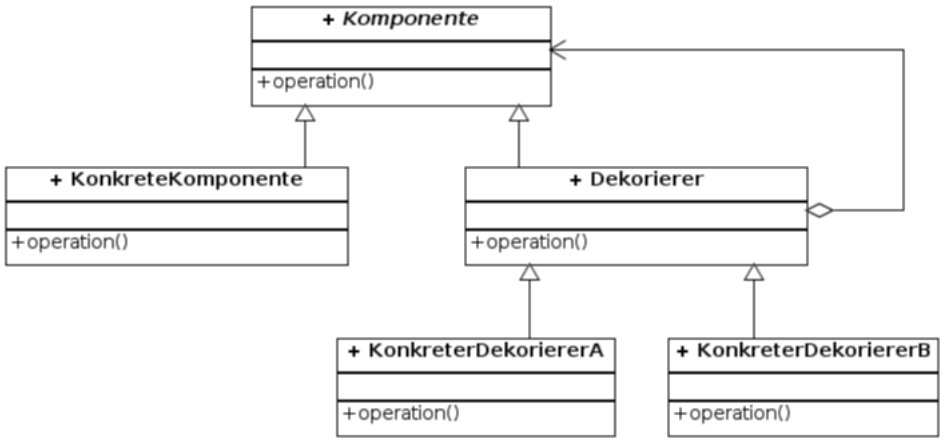


Abbildung 6.10: Dekorierer

man beim Testen auf Objekt-Identität vorsichtig sein. (Ein Vergleich kann falsch ausgehen, obwohl dieselbe Komponente gemeint ist.) Zudem müssen bei der Verwendung von dekorierten Komponenten die Nachrichten vom Dekorator an das dekorierte Objekt weitergeleitet werden.

Ein Beispiel für einen Dekorierer ist die Streamverkettung im Paket `java.io`. Die abstrakte Klasse `InputStream` entspricht hier der abstrakten Komponente, konkrete Komponenten sind z.B. alle zugehörigen LowLevelStreams wie z.B. `FileInputStream`, `ByteArrayInputstream` und andere. Der abstrakte Dekorierer entspricht dem Typ `FilterInputStream` und alle High-Level Streams wie z.B. `DataInputStream`, `BufferedInputStream` etc. sind konkrete `Dekorierer`.
Ein weiteres Beispiel ist das verschöneren von GUI-Komponenten. Ein Button soll z.B. mit einer Umrahmung ”dekoriert” werden. Zwischen dem Aufrufer und dem Buttonobjekt wird das entsprechende Dekoriererobjekt eingefügt. Das Dekoriererobjekt erzeugt die Umrahmung und übergibt den Kontrollfluss an das Textfeld. Da der Dekorierer dieselbe Schnittstelle hat, ändert sich aus der Sicht des Aufrufers nichts.

Beispiel

In einem Ritterspiel gibt es Spielfiguren. Jede davon kann verschiedene Tätigkeiten durchführen. Das folgende Programmbeispiel zeigt, wie das Dekorierer-Muster benutzt wird, um einen Ritter neben seinem Grundverhalten Reiten das Singen und Schwertkämpfen beizubringen.

```
1  // Die abstrakte Komponente
2  public abstract class Spielfigur {
3    public abstract void spielen();
4  }
5
6  // Eine konkrete zu dekorierende Komponente
7  public class Ritter extends Spielfigur {
8    public void spielen() {
9      System.out.println("Reiten");
10   }
11 }
12
13 // Der abstrakte Dekorierer
14 public abstract class Dekorierer extends Spielfigur {
15   // Die zu dekorierende Komponente
16   private Spielfigur figur;
17
18   protected Dekorierer(Spielfigur figur) {
19     this.figur = figur;
20   }
21   // Grundfunktionalitaet
22   public void spielen() {
```

```
23      figur.spielen();
24    }
25  }
26
27  // Ein erster konkreter Dekorierer
28  public class Minnesaenger extends Dekorierer {
29    public Minnesaenger(Spielfigur figur) {
30      super(figur);
31    }
32
33    @Override
34    public void spielen() {
35      System.out.print("Singen, ");
36      super.spielen();
37    }
38  }
39
40  // Noch ein konkreter Dekorierer
41  public class Schwertkaempfer extends Dekorierer {
42
43    public Schwertkaempfer(Spielfigur figur) {
44      super(figur);
45    }
46
47    @Override
48    public void spielen() {
49      System.out.print("Kaempfen, ");
50      super.spielen();
51    }
52  }
53
54  public class DekoriererDemo {
55    public static void main(String []args) {
56      Spielfigur ritter = new Ritter();
57      ritter.spielen();
58      ritter = new Minnesaenger(ritter);
59      ritter.spielen();
60      ritter = new Schwertkaempfer(ritter);
61      ritter.spielen();
62    }
63  }
```

Listing 6.19: Dekorierer Implementierung

Ausgabe:

```
Reiten
Singen, Reiten
Kaempfen, Singen, Reiten
```

Kapitel 7

Ein- und Ausgabe

Dieser Abschnitt behandelt die Kommunikation mit externen Datenquellen. Es kann sich dabei beispielsweise um Dateien, um Speicherbereiche im RAM[1] oder aber auch um Netzwerkverbindungen handeln. Alle diese Datenquellen werden über das gleiche Kommunikationsmodell, sogenannte *Streams*, angesprochen. Vor der Behandlung von Streams werden zwei Hilfsklassen aus dem Paket `java.io`, die Klasse `File` und die Klasse `RandomAccessFile` vorgestellt.

7.1 Die Klasse `java.io.File`

Ein Objekt der Klasse `java.io.File` kapselt eine abstrakte (d.h. betriebssystemunabhängige) Repräsentation einer Datei oder eines Verzeichnisses auf dem lokalen Dateisystem. Die wichtigsten Konstruktoren lauten:

`public File(String pathname)`
Instanziiert ein Fileobjekt mit dem gegebenen Pfadnamen.

`public File(File parent, String child)`
`public File(String parent, String child)`
Instanziiert ein Fileobjekt mit Namen `child` im Verzeichnis `parent`.

`public File(URI uri)`
Instanziiert ein Fileobjekt zum gegebenen `URI`[2]-Objekt.

Bemerkung:
Das Instanziieren eines `File`-Objektes legt keine Datei und kein Verzeichnis an. Eine Instanz der Klasse `File` kapselt lediglich die übergebenen Informationen, wobei beim Anlegen des `File`-Objektes keine Überprüfung auf Gültigkeit vorgenommen wird.
Nachdem ein Objekt der Klasse `File` instanziiert wurde können verschiedene Funktionen verwendet werden. Im Folgenden werden die wichtigsten kurz vorgestellt:

`public boolean exists()`
Liefert `true`, wenn die Datei oder das Verzeichnis existiert, sonst `false`.

`public String getAbsolutePath()`
Liefert den absoluten Pfad der Datei oder des Verzeichnisses. Der absolute Pfad ist betriebssystemabhängig und beginnt unter Linux immer mit dem Zeichen `/`, unter Windows mit dem Laufwerksbuchstaben, gefolgt von `\\`.

`public String getCanonicalPath()`
Liefert jenen eindeutigen absoluten Pfad, in dem die Symbole `.` und `..` aufgelöst sind.

[1] Random Access Memory
[2] Klasse aus `java.net`

`public String getName()`
Liefert den Namen der Datei oder des Verzeichnisses (also den letzten Bestandteil im Pfad).

`public String getParent()`
Liefert den Namen des Verzeichnisses, welches das Fileobjekt enthält.

`public boolean isDirectory()`
Liefert `true`, wenn das Fileobjekt ein existierendes Verzeichnis beschreibt.

`public boolean isFile()`
Liefert `true`, wenn das Fileobjekt eine existierende Datei beschreibt.

`public String[] list()`
Liefert ein Stringfeld, das die Dateien und Verzeichnisse im aktuellen Fileobjekt enthält. Das Fileobjekt muss ein Verzeichnis repräsentieren, sonst wird `null` geliefert.

`public boolean canRead()`
Liefert `true` wenn das Fileobjekt eine Datei mit Leserecht beschreibt.

`public boolean canWrite()`
Liefert `true` wenn das Fileobjekt eine Datei mit Schreibrecht beschreibt.

`public boolean delete()`
Versucht den durch das Fileobjekt beschriebenen Eintrag zu löschen (Verzeichnisse müssen leer sein) und liefert bei Erfolg `true`.

`public long length()`
Liefert die Größe der durch das Fileobjekt beschriebene Datei in Byte. Bei einem Verzeichnis ist das Ergebnis unbestimmt.

`public boolean mkdir()`
Versucht jenes Verzeichnis zu erzeugen, das durch das Fileobjekt beschrieben wird und liefert bei Erfolg `true`.

`public boolean renameTo(File newname)`
Versucht den durch das aktuelle Fileobjekt beschriebenen Eintrag umzubenennen und liefert bei Erfolg `true`.

`public boolean createNewFile() throws IOException`
Erzeugt eine neue leere Datei mit dem zugeordneten absoluten Pfad, wenn eine solche Datei noch nicht existiert. Liefert bei Erfolg `true`, sonst `false`.

Die Klasse `java.io.File` stellt auch 4 statische Datenmitglieder für systemunabhängige Pfadseperatoren zur Verfügung. Das folgende Beispiel demonstriert die Instanziierung einer Fileinstanz, die Unterschiede zwischen absolutem und kanonischem Pfad sowie die Werte der Pfadseperatoren auf einem Windowssystem.

```
1  package ein_aus;
2  import java.io.*;
3
4  public class File1 {
5
6    public static void main(String[] args) {
7      File f1 = new File(".");  // aktuelles Arbeitsverzeichnis
8      File f2 = new File(f1, "File1.java");
9      System.out.println("f1: " + f1.getAbsolutePath());
10     System.out.println("f2: " + f2.getAbsolutePath());
11     try {
12       System.out.println("f1: " + f1.getCanonicalPath());
13       System.out.println("f2: " + f2.getCanonicalPath());
```

```
14        String name = f1.getCanonicalPath() + File.separatorChar + "File2.java";
15        System.out.println("name: " + name);
16        System.out.println(File.pathSeparator + " " + File.pathSeparatorChar);
17        System.out.println(File.separator + " " + File.separatorChar);
18      }
19      catch(IOException e) {
20        System.err.println("Fehler 1: " + e);
21      }
22    }
23  }
```

Listing 7.1: Die Klasse File

Dieses Programm erzeugt die folgende Ausgabe:

```
f1: F:\working\Skripten\Javaskriptum\.
f2: F:\working\Skripten\Javaskriptum\.\File1.java
f1: F:\working\Skripten\Javaskriptum
f2: F:\working\Skripten\Javaskriptum\File1.java
name: F:\working\Skripten\Javaskriptum\File2.java
; ;
\ \
```

Das folgende Beispiel arbeitet von einem gegebenen Startpfad alle Verzeichnisse rekursiv durch, zählt alle Unterverzeichnisse und Dateien sowie die Gesamtgröße aller Dateien. Der Name des Startverzeichnisses kann über die Kommandozeile angegeben werden. Ohne Angabe wird das aktuelle Arbeitsverzeichnis verwendet.

```
1  import java.io.File;
2
3  public class Rekursion {
4    private int dirs = 0;
5    private int files = 0;
6    private long bytes = 0;
7
8    public static void main(String []args) {
9      String path = ".";
10     if(args.length >=1) path = args[0];
11     File f = new File(path);
12     if(!f.isDirectory()) {
13       System.err.println(path + " doesn't exist or not dir");
14       System.exit(1);
15     }
16     Rekursion r = new Rekursion();
17     r.rek(f);
18     System.out.println("Startpfad: " + f.getAbsolutePath());
19     System.out.println("Unterverzeichnisse: " + r.dirs);
20     System.out.println("Dateien: " + r.files);
21     System.out.println("Insgesamt " + r.bytes + " Bytes");
22   }
23
24   private void rek(File startdir) {
25     String []contents = startdir.list();
26     if(contents != null) {
27       for(String f : contents) {
28         File child = new File(startdir, f);
29         if(child.isDirectory()) {
30           this.dirs++;
31           this.rek(child);
32         }
33         else {
34           this.files++;
35           this.bytes += child.length();
36         }
```

```
37        }
38      }
39    }
40  }
```

Listing 7.2: Rekursives Abarbeiten des Verzeichnisbaumes

Dieses Programm erzeugt z.B. die folgende Ausgabe:

```
Startpfad: F:\working\Skripten\Javaskriptum\.
Unterverzeichnisse: 12
Dateien: 38
Insgesamt 63428 Bytes
```

7.2 Die Klassen `Path` und `Files` von NIO2

7.2.1 Einführung

Die Klasse `java.io.File` hat einige Designschwächen. Einge davon sind:

- Viele Methoden zur Dateimanipulation werfen keine Exceptions, sondern liefert nur `true` bzw. `false`. Es besteht keine Möglichkeit den Grund des Fehlers festzustellen.
- Die Methode `rename()` arbeitet nicht konsistent über verschieden Plattformen.
- Es gibt keine echte Unterstützung für symbolische Links.
- Es gibt nur eine eingeschränkte Unterstützung für Dateiattribute wie Zugriffsrechte und Eigentümer. Die verschiedenen Möglichkeiten der einzelnen Plattfomen werden nicht unterstützt.
- Die Funktionalität für Verzeichnisse von `java.io.File` lässt sich nicht auf große Verzeichnisse skalieren.

Aus diesem Grund wurde mit Java 7 eine neues API zur Verarbeitung von Dateien und Verzeichnissen eingeführt, welches die oben beschriebenen Nachteile nicht mehr aufweist. Darüber hinaus werden Möglichkeiten angeboten, Dateien zu kopieren und zu verschieben, um den Verzeichnisbaum rekursiv abzuarbeiten bzw. um den Zustand von Dateien und Verzeichnissen überwachen.

7.2.2 Arbeiten mit Pfaden - das Interface `java.nio.file.Path`

Ein `Path` ist eine Schnittstelle zur Darstellung eines Pfades (absolut oder relativ) im Dateisystem. Eine `Path`-Instanz enthält den Dateinamen und die Verzeichnisliste zur Erstellung des Pfades. In folgendem Listing wird der Zugriff auf eine einfache Datei demonstriert.

```
1 // Vor Java 7
2 File f = new File("test.txt");
3
4 // Mit Java 7
5 Path p = Paths.get("test.txt");
```

Listing 7.3: Zugriff auf eine einfache Datei

Mit Hilfe der Methoden
`File toFile()`
der Schnittstelle `Path` und
`Path toPath()`
der Klasse `File`
lassen sich Objekte vom Typ `Path` in Objekte vom Typ `File` und umgekehrt konvertieren:

```
Path p = new File("test.txt").toPath();
File f = Paths.get("test.txt").toFile();
```

Listing 7.4: Konvertierung von File und Path

Mit einem `Path`-Objekt können 2 grundlegende Arbeiten durchgeführt werden:

- Man kann Pfade bearbeiten, Komponenten extrahieren bzw. über einen Pfad iterieren.
- Man kann Dateien bzw. Verzeichnisse lokalisieren und mit Hilfe der Klasse `java.nio.file.Files` Dateioperationen (Öffnen, Löschen, Attribute setzen usw.) durchführen.

Erstellen eines Pfades

Die Hilfsklasse `java.nio.file.Paths` bietet `get()`-Methoden, mit denen ein Pfad erstellt werden kann. Diese Klasse `Paths` hat nur die beiden folgenden statische Methoden:

`public static Path get(String first, String... more)`
Konvertiert den String `first` bzw. eine weitere über `more` definierte Folge von Strings in einen `Path`. Wirft die unchecked `InvalidPathException` bei einem Fehler.

`public static Path get(URI uri)`
Konvertiert die gegebene URI `uri` in einen Path. Wirft eine `IllegalArgumentException` bei einem Fehler.

Beispiel:

```
Path p1 = Paths.get("/home/user1/data/test.txt");
Path p2 = Paths.get("/home", "user1", "data", "test.txt");
Path p3 = Paths.get(URI.create("file:///home/user1/data/test.txt"));
// Alternative Art der Erzeugung
Path p4 = FileSystems.getDefault().getPath("/home/user1/data/test.txt");
```

Listing 7.5: Erstellen einer Path-Instanz

Bemerkung:
Die Methode `Paths.get()` ist eine Kurzform für `FileSystems.getDefault().getPath()`.

Ein Pfad kann absolut (der erste Teil ist oder enthält eine Wurzel des Dateisystems) oder relativ definiert werden. Er kann auch aus einem einzelnen Dateinamen oder einem Verzeichnisnamen bestehen.

Abrufen von Pfadinformationen

Intern speichert eine `Path`-Instanz die einzelnen Namenselemente als Liste (indiziert mit `0` bis `n-1`). Die wichtigsten Methoden zum Abrufen von Pfadinformationen sind:

`Path getFileName()`
Liefert den Datei- oder Verzeichnisnamen dieses Pfades, also jenen Teil, der von der Wurzel des absoluten Pfades am weitesten entfernt ist.

`int getNameCount()`
Liefert die Anzahl der Namenselemente in diesem Pfad bzw. 0, wenn der Pfad nur eine Wurzelkomponenete enthält.

`Path getName(int index)`
Liefert den durch `index` spezifizierten Namen des Elements dieses Pfades. Wirft bei ungültigem Index eine `IllegalArgumentException`.

`Path subpath(int beginIndex, int endIndex)`
Liefert einen relativen Pfad von `beginIndex` bis `endIndex-1`, der ein Teil dieses Pfades ist. Wirft bei

ungültigem Index eine IllegalArgumentException.

Path getParent()
Liefert diesen Pfad mit Ausnahme des letzten Elements, also im Wesentlichen einen Pfad auf das übergeordnete Verzeichnis.

Path getRoot()
Liefert die Wurzelkomponente dieses Pfades bzw. null, wenn keine Wurzelkomponente existiert.

Der folgende Codeausschnitt demonstriert das Arbeiten mit diesen Methoden:

```
Path p1 = Paths.get("/home/user1/data/test.txt");

System.out.println(p1.getFileName());      // test.txt
System.out.println(p1.getName(1));         // user1
System.out.println(p1.getNameCount());     // 4
System.out.println(p1.subpath(0, 2));      // home/user1
System.out.println(p1.getRoot());          // /
System.out.println(p1.getParent());        // /home/user1/data
```

Listing 7.6: Abrufen von Pfadinformationen

Verknüpfen zweier Pfade

Zum Verknüpfen zweier Pfade stehen die Methoden resolve() und relativize() zur Verfügung:

Path resolve(Path other)
Hängt den Pfad other an diesen Pfad an und liefert den so erstellten Pfad zurück. Ist other ein absoluter Pfad, so wird other retourniert.

Path relativize(Path other)
Diese Methode erstellt einen Pfad von diesem Pfad zum Pfad other. Es wird also ein Pfad erstellt, der in diesem Pfad beginnt und in other endet.

Beispiel:

```
Path p1 = Paths.get("/home/user1/data");
Path p2 = Paths.get("a/b");

Path p3 = p1.resolve(p2);
System.out.println(p3);
p3 = p2.resolve(p1);
System.out.println(p3);

p1 = Paths.get("/home/user1/a/b/test.txt");
p2 = Paths.get("/home/user1/c/test1.txt");
p3 = p1.relativize(p2);
System.out.println(p3);
Path p4 = p1.resolve(p3);
System.out.println(p4);
p4 = p4.normalize();
System.out.println(p4);
```

Listing 7.7: verknüpfen zweier Pfade

Ausgabe:

```
/home/user1/data/a/b
/home/user1/data
../../../c/test1.txt
/home/user1/a/b/test.txt/../../../c/test1.txt
/home/user1/c/test1.txt
```

Bemerkung:
Die Methode `Path normalize()` entfernt aus diesem Pfad alle reduntanten Verzeichnisinformation wie `'.'` und `'..'`.

7.2.3 Die Klasse `java.nio.file.Files`

Diese Klasse beinhaltet viele statische Methoden zum Lesen, Schreiben und Bearbeiten von Dateien und Verzeichnissen. Die Methoden dieser Klasse arbeiten mit `Path`-Objekten und müssen in der Regel auf das Dateisystem zugreifen.

Die Methoden dieser Klasse arbeiten auch mit folgenden Enums aus dem Paket `java.nio.file`:

```
public enum LinkOption implements CopyOption {
  NOFOLLOW_LINKS;    // symbolische Links werden nicht verfolgt
}

public enum StandardCopyOption implements CopyOption {
  REPLACE_EXISTING,   // ersetzt ein existierendes Ziel
  COPY_ATTRIBUTES,    // kopiert auch die Dateiattribute
  ATOMIC_MOVE;        // verschiebt die Datei in einer
                      // atomaren File-System-Operation
}
```

Prüfen von Dateien und Verzeichnissen

Mit Hilfe der Klasse `Files` können unter anderem folgende Prüfungen durchgeführt werden:

- **Prüfen auf Existenz**

  `public static boolean exists(Path path, LinkOption... options)`
  Prüft den Pfad `path` auf Existenz. Wird der optionale Parameter `LinkOption.NOFOLLOW_LINKS` angegeben, so werden symbolische Links nicht verfolgt, standardmäßig werden sie verfolgt.

  `public static boolean notExists(Path path, LinkOption... options)`
  Prüft ob der Pfad `path` nicht existiert. Wird der optionale Parameter `LinkOption.NOFOLLOW_LINKS` angegeben, so werden symbolische Links nicht verfolgt, standardmäßig werden sie verfolgt.

- **Berechtigungen prüfen**
  Die folgenden Methoden prüfen, ob das durch `path` angegebene Verzeichnis bzw. die durch `path` angegebene Datei gelesen, geschrieben bzw. ausgeführt werden kann:

  ```
  public static boolean isReadable(Path path)
  public static boolean isWritable(Path path)
  public static boolean isExecutable(Path path)
  ```

- **Auf Gleichheit prüfen**

  `public static boolean isSameFile(Path p1, Path p2) throws IOException`
  Prüft, ob die Pfade `p1` und `p2` die selbe Datei beschreiben. Wenn die beiden Pfade identisch sind, so wird dabei keine Prüfung auf Existenz vorgenommen.

Löschen von Dateien und Verzeichnissen

Zum Löschen von Dateien und Verzeichnissen stehen die folgenden Methoden zur Verfügung:

`public static void delete(Path path) throws IOException`
Löscht den durch `path` angegebenen Eintrag. Wirft eine

- `NoSuchFileException`, wenn der Eintrag nicht existiert.

- `DirectoryNotEmptyException`, wenn der Eintrag ein nicht leeres Verzeichnis ist.
- `IOException`, bei einem allgemeinen IO-Fehler.

```
public static boolean deleteIfExists(Path path) throws IOException
```
Wie `delete()`, bei nicht existierendem Eintrag wird aber keine `NoSuchFileException` geworfen.

Kopieren von Dateien und Verzeichnissen

Zum Kopieren von Dateien und Verzeichnissen stehen die folgenden Methoden zur Verfügung:

```
public static Path copy(Path source, Path target, CopyOption... options)
                  throws IOException
```
Kopiert den Eintrag mit Pfad `source` auf den Eintrag mit Pfad `target`. Standardmäßig schlägt die Operation fehl, wenn `target` existiert. Verzeichnisse können kopiert werden, das Zielverzeichnis ist aber immer leer, selbst wenn das Quellverzeichnis Dateien enthält. Beim Kopieren eines symbolischen Links wird dessen Ziel kopiert. Dieses Verhalten kann durch Angabe zusätzlicher Optionen beeinflusst werden. Mögliche Optionen sind:

- `REPLACE_EXISTING`
  Es wird auch kopiert, wenn das Ziel existiert. Ist das Ziel ein symbolischer Link, so wird ein Link erzeugt und nicht die Datei selbst kopiert. Ist das Ziel ein nicht leeres Verzeichnis, wird eine `FileAlreadyExistsException` geworfen.
- `COPY_ATTRIBUTES`
  Es werden auch die Dateiattribute kopiert. Welche Attribute übertragen werden, ist system- und plattformabhängig. Sicher übertragen wird der Zeitpunkt der letzten Änderung.
- `NOFOLLOW_LINKS`
  Wenn der zu kopierende Eintrag ein symbolischer Link ist, so wird der Link und nicht dessen Ziel kopiert.

Verschieben von Dateien und Verzeichnissen

Zum Verschieben von Dateien und Verzeichnissen stehen die folgenden Methoden zur Verfügung:

```
public static Path move(Path source, Path target, CopyOption... options)
                  throws IOException
```
Bei vorhandenem Ziel ist die Operation standardmäßig nicht erfolgreich. Beim Verschieben eines Verzeichnisses wird dessen Inhalt nicht mitverschoben. Dieses Verhalten kann durch Angabe zusätzlicher Optionen beeinflusst werden. Mögliche Optionen sind:

- `REPLACE_EXISTING`
  Es wird auch verschoben, wenn das Ziel existiert. Ist das Ziel ein symbolischer Link, so wird der Link ersetzt.
- `ATOMIC_MOVE`
  Der Verschiebevorgang wird als atomare Dateioperation ausgeführt. Wird dies vom betriebssystem nicht unterstützt, so wird eine Exception geworfen. Bei Verwendung dieser Option ist sichergestellt, dass Prozesse, die eine Datei beobachten, erst auf die vollständig verschobene Datei zugreifen.

7.2.4 Suchen von Dateien

Globs

Ein glob-Pattern ist ein String, der speziell verwendet wird, um Muster gegen Pfade zu prüfen. Sie sind einfacher als reguläre Ausdrücke zu verwenden und gehorchen den folgenden einfachen Regeln:

- Der Stern `*` passt auf jede beliebige Zeichengruppe (auch auf den Leerstring).
- Zwei Sterne `**` arbeiten wie `*`, überschreiten aber auch Verzeichnisgrenzen. Sie dienen also als Pattern für ganze Pfade.

- Das Fragezeichen ? passt auf jedes einzelne Zeichen.
- Geschwungene Klammern geben eine Aufzählung von Untermustern an.
- In eckigen Klammern werden Zeichenmengen angegeben, von denen genau ein Zeichen übereinstimmen muss. Mit Hilfe von '-' können auch Bereiche angegeben werden (vgl. reguläre Ausdrücke). In eckigen Klammern haben die Zeichen *, ? und \ keine Sonderstellung.
- Die Zeichen *, ? und \ können mit dem Backslash ausmaskiert werden.

  Die folgende Tabelle zeigt einige Beispiele:

Glob	Beschreibung
*.html	Passt auf alle Strings, die auf *.html enden.
???	Passt auf alle Strings, die exakt 3 Zeichen lang sind.
[0-9]	Passt auf alle Strings, die ein Ziffernzeichen beinhalten.
*.{htm,html,pdf}	Passt auf alle Strings, die mit .htm, .html oder .pdf enden.
A[a-z]*.java	Passt auf alle Strings, die mit A beginnen, gefolgt von einem Kleinbuchstaben und die auf .java enden.
{foo*,*[0-9]*}	Passt auf jeden String, der mit foo beginnt oder ein Ziffernzeichen beinhaltet.

Das Interface PathMatcher

Mit Hilfe des Interfaces java.nio.file.PathMatcher können Verzeichnisse und Dateien gesucht werden. Mit Hilfe der Methode getPathMatcher() der Klasse FileSystem erhält man den PathMatcher des Betriebssystems.

```
public abstract PathMatcher getPathMatcher(String syntaxAndPattern)
```
Liefert einen PathMatcher, mit dem Pfade gesucht werden können die zum übergebenen String passen. Dieser String syntaxAndPattern hat den Aufbau syntax:pattern, wobei syntax entweder regex (fur regulären Ausdruck) oder glob sein darf.

Der folgende Codeausschnitt erstellt einen PathMatcher, mit dem nach .java und .class - Dateien gesucht werden kann.

```
1 PathMatcher m = FileSystems.getDefault().getPathMatcher("glob:*.{java,class}");
```

Listing 7.8: Erstellen eines PathMatcher

PathMatcher definiert eine einzige Methode matches():

```
boolean matches(Path path)
```
Liefert true, wenn der übergebene Pfad auf diesen PathMatcher passt.

7.2.5 Abarbeiten des Verzeichnisbaumes

Zum rekursiven Abarbeiten des Verzeichnisbaumes stellt die Klasse java.nio.Files die folgenden beiden Methoden zur Verfügung:

```
public static Path walkFileTree(Path start, FileVisitor<? super Path> visitor)
                    throws IOException
```
Diese Methode arbeitet ab dem Startpfad start den Verzeichnisbaum rekursiv durch und verwendet dabei die in visitor definierte Funktionalität.

```
public static Path walkFileTree(Path start, Set<FileVisitOption> options,
                                int maxDepth, FileVisitor<? super Path> visitor)
                    throws IOException
```
Wie oben, es können aber zusätzliche Optionen (wie z.B. NOFOLLOW_LINKS) sowie eine maximale Rekursionstiefe angegeben werden.

Das Interface java.nio.file.FileVisitor

Eine Implementierung dieses Interfaces legt die Funktionalität beim Abarbeiten des Verezichnisbaumes fest:

```
public interface FileVisitor<T>
```

Der Typstellvertreter T wird in den aktuellen Implementierungen mit ? super Path ersetzt.

Das Interface definiert 4 Methoden, welche alle eine Enum-Instanz vom Typ FileVisitResult retournieren:

```
public enum FileVisitResult {
  CONTINUE,        // die Abarbeitung wird fortgesetzt
  SKIP_SIBLINGS,   // Die Geschwister dieses Pfades werden nicht abgearbeitet
  SKIP_SUBTREE,    // Die Eintraege dieses Verzeichnisses werden nicht abgearbeitet
  TERMINATE;       // Beendet die Abarbeitung des Verzeichnisbaumes
};
```

Die 4 Methoden im Detail:

```
FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)
                throws IOException
```

Wird beim Betreten eines Verzeichnisses aufgerufen, bevor der Inhalt des Verzeichnisses abgearbeitet wird. Liefert die Methode CONTINUE, dann wird der Inhalt des Verzeichnisses abgearbeitet. Bei SKIP_SUBTREE oder SKIP_SILBINGS wird der Inhalt übersprungen.

```
FileVisitResult postVisitDirectory(T dir, IOException exc)
                throws IOException
```

Wird beim Verlassen eines Verzeichnisses aufgerufen, nachdem der Inhalt des Verzeichnisses abgearbeitet wird. Die Methode wird auch aufgerufen, wenn während der Abarbeitung des Verzeichnisses eine IOException auftritt (diese wird in exc übergeben, sonst steht in exc null) bzw. wenn während der Abarbeitung SKIP_SIBLINGS retourniert wurde.

```
FileVisitResult visitFile(T file, BasicFileAttributes attrs)
                throws IOException
```

Wird für jede Datei innerhalb eines Verzeichnisses aufgerufen.

```
FileVisitResult visitFileFailed(T file, IOException exc)
                throws IOException
```

Wird für jede Datei aufgerufen, die nicht abgearbeitet werden konnte. In exc steht der Grund des Fehlers.
Möchte bzw. muss man nicht alle 4 Methoden implementieren, so kann man von der Klasse SimpleFileVisitor erben. Diese Klasse implementiert alle 4 Methoden mit dem Returnwert CONTINUE.

Beispiel

Das folgende Beispiel durchsucht einen Verzeichnisbaum und gibt die Pfade zu allen .java, .c und .cpp - Dateien auf der Konsole aus.

```
1  package nio;
2
3  import java.io.IOException;
4  import java.nio.file.FileSystems;
5  import java.nio.file.FileVisitResult;
6  import java.nio.file.Files;
7  import java.nio.file.Path;
8  import java.nio.file.PathMatcher;
9  import java.nio.file.Paths;
10 import java.nio.file.SimpleFileVisitor;
11 import java.nio.file.attribute.BasicFileAttributes;
```

```
12
13 public class Find {
14   public static class Finder extends SimpleFileVisitor<Path> {
15     private final PathMatcher matcher;
16     private int numMatches = 0;
17
18     public Finder(String pattern) {
19       matcher = FileSystems.getDefault().getPathMatcher("glob:" + pattern);
20     }
21
22     // Vergleicht das glob - Pattern mit dem Namen des Pfades
23     public void find(Path file) {
24       Path name = file.getFileName();
25       if (name != null && matcher.matches(name)) {
26          numMatches++;
27          System.out.println(file);
28       }
29     }
30
31     // Gibt die Anzahl der Uebereinstimmungen auf stdout aus.
32     public void done() {
33       System.out.println("Matched: " + numMatches);
34     }
35
36     @Override
37     public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {
38       find(file);
39       return FileVisitResult.CONTINUE;
40     }
41
42     @Override
43     public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) {
44       find(dir);
45       return FileVisitResult.CONTINUE;
46     }
47
48     @Override
49     public FileVisitResult visitFileFailed(Path file, IOException exc) {
50       System.err.println(exc);
51       return FileVisitResult.CONTINUE;
52     }
53   }
54
55   public static void usage() {
56     System.err.println("java Find <path> -name \"<glob_pattern>\"");
57     System.exit(1);
58   }
59
60   public static void main(String[] args) {
61     if (args.length < 3 || !args[1].equals("-name")) {
62       usage();
63     }
64
65     Path startingDir = Paths.get(args[0]);
66     String pattern = args[2];
67
68     try {
69       Finder finder = new Finder(pattern);
70       Files.walkFileTree(startingDir, finder);
71       finder.done();
72     }
73     catch(IOException ex) {
74       System.err.println(ex);
```

```
75      }
76    }
77  }
```

Listing 7.9: Rekursives Abarbeiten des Verzeichnisbaumes

Ausgabe:

```
/home/reio/DatenNeu/Skripten/xml/source/src/xml/MyHandler.java
/home/reio/DatenNeu/Skripten/xml/source/src/xml/StartParser.java
/home/reio/DatenNeu/Skripten/xml/source/src/dom/Example_3.java
...
/home/reio/DatenNeu/Skripten/CPP_Skriptum/sources/kap_25/DayTimeClient.cpp
/home/reio/DatenNeu/Skripten/CPP_Skriptum/sources/kap_25/udp_1.cpp
Matched: 569
```

7.3 Die Klasse `java.io.RandomAccessFile`

Die Klasse `RandomAccessFile` bietet ein Modell zum Lesen/Schreiben von Dateien, das nicht kompatibel zu dem universelleren Stream-Modell ist. Dieses Stream-Modell wurde fÜr generelle I/O Operationen (Streams, Sockets) entwickelt, während die Klasse `RandomAccessFile` Vorteile beim Umgang mit Dateien zur lokalen Dateispeicherung bietet. So kann man z.B. einen Schreib-/Lesezeiger innerhalb der Datei positionieren und Dateien zum gleichzeitigen Lesen und Schreiben öffnen.

Die Konstruktoren der Klasse `RandomAccessFile` sind:

```
public RandomAccessFile(File file, String mode)
throws FileNotFoundException
public RandomAccessFile(String file, String mode)
throws FileNotFoundException
```

Der String `mode` kann dabei die Werte `"r"` (nur lesen) oder `"rw"` (lesen/schreiben) annehmen. Wenn eine nicht existierende Datei im read-only Modus geöffnet wird, dann werfen die Konstruktoren eine `FileNotFoundException`, im read-write Modus wird eine neue Datei mit Länge Null erzeugt.

Innerhalb einer RandomAccessDatei gibt es zwei logische Marken:

- **Stream-Position-Indicator (SPI)**
  Der SPI markiert die momentane Schreib-Lese-Position. Jeder Schreib-Lesevorgang beginnt an der aktuellen SPI-Position und bewegt den SPI um die Anzahl der gelesenen/geschriebenen Bytes in Richtung Dateiende.
- **EOF-Marker (EOF)**
  Der EOF-Marker markiert das logische Ende der Datei. Jeder Lesevorgang, bei dem der SPI an oder hinter der Position des EOF-Markers steht, schlägt fehl.

Positionieren innerhalb der Datei

Zur Manipulation des internen Schreib/Lesezeigers (SPI) stehen folgende Methoden zur Verfügung:

`long getFilePointer() throws IOException`
Liefert die momentane Position des SPI in Byte. Folgende Schreib/Lesezugriffe starten bei dieser Position.

`long length() throws IOException`
Liefert die Länge der Datei in Byte (also die Position des EOF-Markes).

`void seek(long position) throws IOException`
Setzt die aktuelle Position des SPI in Byte vom Dateianfang an gerechnet (nullbasiert). Folgende Schreib/Lesefunktionen starten bei dieser Position. Der SPI kann auch hinter den EOF-Marker verschoben werden.

Der EOF-Marker wird erst dann nachgezogen, wenn an dieser Position Daten geschrieben werden.

`public void setLength(long newLength) throws IOException`
Setzt die logische Länge der Datei. Der EOF-Marker wird also auf `newLength` Bytes hinter den Dateianfang gesetzt. Mit `setLength(0L)` wird der Inhalt der Datei also logisch gelöscht.

Schreibe- und Lesezugriffe

Im Folgenden werden die grundlegenden Schreib-/Lesefunktionen vorgestellt. Man kann mit ihnen einzelne Bytes (Integerwerte im Bereich von 0-255), Bytefelder bzw. Teile von Bytefeldern lesen und schreiben. Diese Funktionen sind für die Ein-/Ausgabe in Javaprogrammen grundlegend und stehen in dieser Form auch für Streams zur Verfügung:

`int read() throws IOException`
Liefert den Wert des nächsten Bytes (0-255) bzw. -1 bei Dateiende.

`int read(byte[] b) throws IOException`
Versucht `b.length` Bytes aus der Datei zu lesen um damit das Array `b` zu füllen. Liefert die Anzahl der tatsächlich gelesenen Bytes bzw. -1 bei Dateiende.

`int read(byte[] b, int offset, int len) throws IOException`
Versucht `len` viele Bytes aus der Datei zu lesen und in das Feld `b` ab der Position `offset` zu schreiben. Liefert die Anzahl der tatsächlich gelesenen Bytes bzw. -1 bei Dateiende.

`void write(int b) throws IOException`
Schreibt das niederwertigste Byte von `b` in die Datei.

`void write(byte[] b) throws IOException`
Schreibt alle Bytes des Feldes `b` in die Datei.

`void write(byte[] b, int offset, int len) throws IOException`
Schreibt `len` Bytes, die im Feld `b` ab der Position `offset` stehen, in die Datei.

Neben diesen grundlegenden Schreib-/Leseoperationen wird auch das Lesen/Schreiben aller primitiven Datentypen und einiger nicht primitiver Typen unterstützt. Alle Methoden werfen gegebenenfalls eine `IOException`.
Die Klasse `RandomAccessFile` implementiert die Interfaces `java.io.DataInput` und `java.io.DataOutput`. Diese Interfaces definieren eine Reihe von Methoden zum Lesen und Schreiben primitiver Typen:

DataInput	DataOutput
boolean readBoolean()	void writeBoolean(boolean b)
byte readByte()	void writeByte(int b)
short readShort()	void writeShort(int s)
char readChar()	void writeChar(int c)
int readInt()	void writeInt(int i)
long readLong()	void writeLong(long l)
float readFloat()	void writeFloat(float f)
double readDouble()	void writeDouble(double d)
String readLine()	void writeBytes(String s)
	void writeChars(String s)
String readUTF()	void writeUTF(String s)

Eine Sonderstellung nehmen die Methoden `readUTF()` und `writeUTF()` ein. Sie dienen dazu, die 2 Byte langen UNICODE-Zeichen, mit denen Java intern arbeitet, in definierter Weise in 1, 2 oder 3 Byte lange Einzelzeichen zu verwandeln. Hat das Zeichen einen Wert zwischen `\u0001` und `\u007F`, wird es als Einzelbyte gespeichert. Hat es einen Wert zwischen `u0080` und `\u07FF`, belegt es zwei Byte, und in allen anderen Fällen werden drei Byte verwendet. Nähere Informationen findet man in der Dokumentation zu `java.io.DataInput`.

Schließen der Datei

Wird eine RandomAccessDatei nicht mehr benötigt, so ist sie mit Hilfe der Methode

```
void close() throws IOException
```

zu schließen.

Seit Java 7 gibt es bei der Fehlerbehandlung die Möglichkeit die Technik ***try with ressources*** zu verwenden. Dabei wird in runden Klammern nach dem Schlüsselwort `try` eine Autoclosable Ressource erzeugt. Diese wird beim Verlassen des `try`-Blocks automatisch geschlossen, man benötigt also keinen `finally`-Block mehr. Voraussetzung ist, dass die Ressource das Interface `java.lang.AutoCloseable` (neu seit Java 7) implementiert:

```
public interface AutoCloseable {
  void close();
}
```

Die Klasse `RandomAccessFile` und alle Datenstromklassen im Paket `java.io` implementieren seit Java 7 dieses Interface.
Verwendung von `RandomAccessFile` ohne *try with ressources*:

```
RandomAccessFile file = null;
try {
  file = RandomAccessFile("test.dat", "rw");
  // DateiOperationen
} catch(IOException ex) {
  // angemessene Fehlerbehandlung
} finally {
  try { file.close(); }
  catch(IOException ex) { /*.... */ }
}
```

Verwendung von `RandomAccessFile` mit *try with ressources*:

```
try(RandomAccess file = RandomAccessFile("test.dat", "rw")) {
  // DateiOperationen
} catch(IOException ex) {
  // angemessene Fehlerbehandlung
}
```

Beispiel 1

Im folgenden Beispiel wird zuerst ein Byte, dann ein Integer und zuletzt ein UTF-String in eine Datei geschrieben. Danach wird der interne Dateizeiger auf den Beginn des UTF-Strings gesetzt und dieser wird wieder gelesen. Zum sauberen Schließen der datei wird die Technik *try with ressources* verwendet.

```
import java.io.*;

public class RandomAccess_1 {
  public static void main(String[] args) {
    try (RandomAccessFile rf = new RandomAccessFile("test.dat", "rw")) {
      file.write(-1);       // ein Byte schreiben
      file.writeInt(0);     // Integer - 4 Byte schreiben
      file.writeUTF("aäuüoösß");
      file.seek(5);
      String s = rf.readUTF();
      System.out.println(s);
```

```
12      }
13      catch(FileNotFoundException e) {
14        System.err.println("Fehler beim Öffnen der Datei: " + e);
15      }
16      catch(IOException e) {
17        System.err.println("Allgemeiner Dateizugriffsfehler: " + e);
18      }
19    }
20  }
```

Listing 7.10: RandomAccessFile 1

Die Datei hat den folgenden Inhalt:

```
000000 FF 00 00 00 00 00 0C 61  C3 A4 75 C3 BC 6F C3 B4
        .  .  .  .  .  .  .  a   .  .  u  .  .  o  .  .
000010 73 C3 9F
        s  .  .
```

Beispiel 2

Im folgenden Beispiel werden die Elemente eines Integerarrays in ein RandomAccessFile geschrieben, danach wird der SPI 12 Bytes hinter das Dateiende gesetzt und die Elemente das Arrays werden noch einmal geschrieben. Zwischendurch wird wiederholt die logische Dateilänge ausgegeben. Zum Schluss werden alle in der Datei gespeicherten Integerwerte ausgegeben.

```
1  package inout;
2
3  import java.io.IOException;
4  import java.io.RandomAccessFile;
5
6  public class RandomAccessDemo {
7
8    public static void main(String[] args) {
9      int []a = {1, 2, 3, 4, 5};
10
11     try(RandomAccessFile file = new RandomAccessFile("daten.dat", "rw")) {
12       file.setLength(0L);                    // EOF-Marker auf Position 0
13       for(int x : a) {                       // Alle Elemente des Arrays a schreiben
14         raf.writeInt(x);
15       }
16       System.out.println("Dateilaenge: " + raf.length());
17       file.seek(file.getFilePointer() + 12); // SPI 12 Bytes nach hinten verschieben
18       System.out.println("Dateilaenge: " + raf.length());
19       for(int x : a) {                       // Alle Elemente des Arrays a schreiben
20         raf.writeInt(x);
21       }
22       System.out.println("Dateilaenge: " + raf.length());
23       // Dateinhalt ausgeben
24       file.seek(0);
25       int n = (int) file.length() / 4;
26       for(int i = 0; i < n; i++) {
27         System.out.format("%4d", file.readInt());
28       }
29       System.out.println();
30     }
31     catch(IOException ex) {
32       System.out.println("Fehler: " +  ex);
33     }
34   }
35 }
```

Listing 7.11: Arbeiten mit RandomAccessFile

Ausgabe:

```
Dateilaenge: 20
Dateilaenge: 20
Dateilaenge: 52
   1   2   3   4   5   0   0   0   1   2   3   4   5
```

Beispiel 3

Schreibt man in ein `RandomAccessFile` Datensätze gleicher Länge, so kann man den SPI durch richtige Positionierung auf den Beginn eines bestimmten Datensatzes setzen und so genau diesen Datensatz lesen bzw. schreiben. Auf Grund dieser Eigenschaft hat die Klasse `RandomAccessFile` auch ihren Namen (Datei mit wahlfreiem Zugriff).
Im folgenden Beispiel wird demonstriert, wie ein Objekt vom Typ `Schueler` (vgl. nachfolgendes Listing) als Datensatz fixer Länge in eine RandomAccessDatei geschrieben wird. Dabei hat ein Datensatz folgenden Aufbau:

```
Katalognummer:  4 Byte-Integer
Zuname:        30 Byte char-Array
Vorname:       30 Byte char-Array
Geschlecht:     1 Byte (W/M)
--------------------------------
Gesamlänge:    65 Byte
```

```java
package schueler;

import java.util.ArrayList;
import java.util.List;

public class Schueler {

  private int nr;
  private String nachname;
  private String vorname;
  private char geschlecht;

  private static final List<Schueler> klasse;

  static {
    klasse = new ArrayList<Schueler>();

    klasse.add(new Schueler(1, "Ehn", "Wilhelm", 'M'));
    // ...
    klasse.add(new Schueler(18, "Tröscher", "Dominik",'M'));
  }

  public static List<Schueler> getKlasse() {
    return klasse;
  }

  public Schueler(int nr, String nachname, String vorname, char geschlecht) {
    this.nr = nr;
    this.nachname = nachname;
    this.vorname = vorname;
    this.geschlecht = geschlecht;
  }

  // ab hier nur mehr Getter, toString

}
```

Listing 7.12: Geschäftsklasse `Schueler.java`

Die Dateifunktionen werden in einer Klasse `FileUtilities` als statische Methoden implementiert. Alle Methoden erhalten ein bereits geöffnetes `RandomAccessFile` und werfen allfällige Exceptions, damit diese im Hauptprogramm behandelt werden können.

```
1  package schueler;
2
3  import java.io.EOFException;
4  import java.io.IOException;
5  import java.io.RandomAccessFile;
6  import java.util.Arrays;
7
8
9  public class FileUtilities {
10
11   public static final int LENGTH = 65;
12
13   /**
14    * Hängt den Schüler schueler an die Datei file an
15    * Geschrieben werden Datensätze der fixen Länge 65 Byte (4/30/30/1)
16    * @param file Datei in die der datensatz geschrieben wird
17    * @param schueler
18    * @throws IOException bei einem Dateizugriffsfehler
19    */
20   public static void appendToFile(RandomAccessFile file, Schueler schueler) throws
         IOException {
21     long aktpos = file.getFilePointer();
22     file.seek(file.length());
23
24     file.writeInt(schueler.getNr());
25     byte []nn = Arrays.copyOf(schueler.getNachname().getBytes(), 30);
26     file.write(nn);
27     byte []vn = Arrays.copyOf(schueler.getVorname().getBytes(), 30);
28     file.write(vn);
29     file.writeByte(schueler.getGeschlecht());
30     file.seek(aktpos);
31   }
32
33   /**
34    * Liest den Schüler mit der Position pos aus der Datei file
35    * @param file
36    * @param pos Nullbasierte Nummer des zu lesenden Datensatzes
37    * @return der erfolgreich gelesenen Schüler
38    * @throws IOException bei einem Dateizugriffsfehler
39    * @throws EOFException wenn es den Datensatz mit der Nummer pos nicht gibt
40    */
41   public static Schueler readFromFile(RandomAccessFile file, int pos) throws
         IOException {
42     long aktpos = file.getFilePointer();
43     if(pos < 0 || (pos + 1) * LENGTH > file.length()) {
44       throw new EOFException("Wrong position: " + pos);
45     }
46     file.seek(pos * LENGTH);
47     int nr = file.readInt();
48     byte []nn = new byte[30];
49     file.read(nn);
50     String nachname = new String(nn, 0, 30).trim();
51     byte []vn = new byte[30];
52     file.read(vn);
53     String vorname = new String(vn, 0, 30).trim();
54     char geschlecht = (char) file.read();
55     return new Schueler(nr, nachname, vorname, geschlecht);
56   }
57 }
```

Listing 7.13: Schreiben und Lesen von Datensätzen fixer Länge

Die zugehörige Anwendung schreibt alle Schüler, die von der Methoden Schueler.getKlasse() geliefert werden als Datensätze fixer Länge in eine Datei, liest dann 10 Datensätze mit zufälligen Positionen aus der Datei und gibt deren Stringdarstellungen auf der Konsole aus:

```
1  package schueler;
2
3  import java.io.IOException;
4  import java.io.RandomAccessFile;
5  import java.util.Random;
6
7  public class SchuelerMain {
8
9    private final static Random RD = new Random();
10
11   public static void main(String[] args) {
12     // Anzahl der Schüler bestimmen
13     int anz = Schueler.getKlasse().size();
14     try(RandomAccessFile f = new RandomAccessFile("schueler.dat", "rw")) {
15       // Alle Schüler in die Datei schreiben
16       for(Schueler sch : Schueler.getKlasse()) {
17         FileUtilities.appendToFile(f, sch);
18       }
19       // 10 zufällige Schüler aus der Datei lesen und auf der Konsole ausgeben
20       for(int i = 0; i < 10; i++) {
21         int pos = RD.nextInt(anz);
22         Schueler akt = FileUtilities.readFromFile(f, pos);
23         System.out.println(akt);
24       }
25     } catch(IOException ex) {
26       System.out.println("Fehler: " + ex.getMessage());
27     }
28   }
29 }
```

Listing 7.14: Testprogramm zu FileUtilities

Mögliche Ausgabe:

```
15 Simmer          Patrick         M
18 Tröscher        Dominik         M
 2 Gruber          Sarah           W
13 Schirmer        Kurt            M
14 Schneider       Florens         M
 1 Ehn             Wilhelm         M
12 Sattler         Benedikt        M
 4 Hamberger       Sebastian       M
 4 Hamberger       Sebastian       M
 9 Purker          Angela          W
```

7.4 Bytestreams

Ein Stream transportiert Daten von einer Quelle zu einem Ziel. Zur bidirektionalen Kommunikation sind also immer 2 Streams nötig, ein Eingabe- und ein Ausgabestream. Ein Datenstrom verarbeitet die Daten grundsätzlich sequentiell. Je nach verwendeter Transporteinheit unterscheidet man zwei verschiedene Typen von Datenströmen:

(1) **Bytestreams**
Die verwendete Transporteinheit ist das Byte (8 bit - Integerwert im Bereich 0-255). Er eignet sich also zur Übertragung beliebiger Daten, speziell auch für die Verarbeitung von Binärdaten.

(2) **Character-Streams**
Die verwendete Transporteinheit ist das Character (16 bit - Integerwert im Bereich 0-65535). Er transportiert also Unicodezeichen und eignet sich ausschließlich zur Übertragung von Texten.

Technisch wird jeder Stream zunächst als abstraktes Konstrukt eingeführt, dessen Fähigkeit darin besteht, Bytes auf ein imaginäres Ausgabegerät zu schreiben oder von diesem zu lesen. Erst konkrete Unterklassen binden die Zugriffsroutinen an echte Ein- oder Ausgabegeräte, wie beispielsweise an Dateien, Felder oder Kommunikationskanäle im Netzwerk.

Neben der Unterscheidung von Byte- und Characterstreams ist für das Verständnis auch die Einteilung in Lowlevelstreams und Filterstreams wesentlich:

(1) Lowlevelstreams lesen und schreiben von bzw. auf ein konkretes Eingabe- bzw. Ausgabegerät (Datei, Socket, etc.) und stellen nur die Grundfunktionalität sowie einen einfachen Satz an Schreib- und Lesefunktionen zur Verfügung.

(2) Filterstreams lesen und schreiben Daten von bzw. auf einen anderen Stream (es kann sich dabei um einen Lowlevelstream oder einen anderen Filterstream handeln). Mit Hilfe von Filterstreams kann man die Funktionalität eines Lowlevelstreams erweitern. Die dabei angewendete Technik heißt Streamverkettung. Im Zentrum liegt immer ein Lowlevelstream. Zur Erweiterung seiner Funktionität werden ein oder mehrere Filterstreams in einer Art Schichtenmodell aufgesetzt. Damit kann die Funktionalität des Datenstromes bestimmten Erfordernissen angepasst werden.

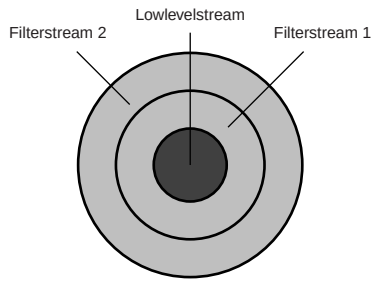


Abbildung 7.1: Streamverkettung

7.5 Bytestreams

7.5.1 Lowlevel - Bytestreams

Alle Lowlevel-Bytestreamklassen sind von `java.io.InputStream` (zum Lesen) und `java.io.OutputStream` (zum Schreiben) abgeleitet. Es handelt sich dabei um abstrakte Klassen, die mit Hilfe von Subklassen konkretisiert werden.

Die Klasse `java.io.OutputStream`

Die wichtigsten Methoden dieser Klasse sind:

`public OutputStream()`
Der parameterlose Konstruktor dient zur Erzeugung eines Subklassenobjektes.

`public void close() throws IOException`
Schließt den Stream.

`public void flush() throws IOException`
Schreibt allfällig gepufferte Daten physikalisch auf das Ausgabegerät.

`public abstract void write(int b) throws IOException`
Schreibt das niederwertigste Byte von `b` in den Stream.

`public void write(byte[] b) throws IOException`
Schreibt alle Bytes des Feldes `b` in den Stream.

```
public void write(byte[] b, int offset, int len)
throws IOException
```
Schreibt `len` Bytes des Feldes `b` ab der Feldposition `offset` in den Stream.

Die Klasse `java.io.InputStream`

Diese abstrakte Klasse ist Basis aller Lowlevel-Eingabestreams. Sie stellt im Wesentlichen die folgenden Methoden zur Verfügung:

`public InputStream()`
Parameterloser Konstruktor.

`public abstract int read() throws IOException`
Liefert das nächste Byte aus dem Stream bzw. -1 bei Streamende.

`public int read(byte[] b) throws IOException`
Versucht so viele Bytes aus dem Stream zu lesen um das Feld `b` zu füllen. Liefert die Anzahl der gelesenen Bytes bzw. -1 bei Streamende.

`public int read(byte[] b, int off, int len) throws IOException`
Versucht `len` viele Bytes aus dem Stream zu lesen und in das Feld `b` ab der Position `off` zu schreiben. Liefert die Anzahl der gelesenen Bytes bzw. -1 bei Streamende.

`public long skip(long n) throws IOException`
Versucht `n` Bytes des Eingabestreams zu überspringen. Liefert die Anzahl der tatsächlich übersprungenen Bytes.

`public int available() throws IOException`
Liefert die Anzahl jener Bytes, die ohne zu Blocken gelesen oder übersprungen werden können.

`public void close() throws IOException`
Schließt den Stream.

`public void mark(int readlimit)`
Setzt eine Marke, zu der mit `reset()` zurückgesprungen werden kann, solange nicht mehr als `readlimit` weitere Bytes aus dem Stream gelesen wurden.

`public void reset() throws IOException`
Springt zu einer allfällig gesetzten Marke.

`public boolean markSupported()`
Liefert wahr, wenn der Stream markieren unterstützt.

Beispiel

In folgenden Beispiel werden zwei Kopierfunktionen vorgestellt. Die erste kopiert Daten byteweise vom Inputstream `is` zum Outputstream `os`, während die zweite einen internen Puffer der Größe `size` verwendet.

```
1  import java.io.*;
2
3  public class CopyLowLevel {
4
5    public static void copySingleByte(InputStream is, OutputStream os) throws
         IOException {
6      int ch;                 // Zeichen zum lesen und schreiben
7      while((ch = is.read()) != -1)
8        os.write(ch);
9    }
10
11   public static void copyBuffer(InputStream is, OutputStream os, int size) throws
         IOException {
12     int n;     // Anzahl der tatsaechlich gelesenen Zeichen
13     byte []b = new byte[size];      // Buffer
14     while((n = is.read(b)) != -1)
15       os.write(b, 0, n);
16   }
17 }
```

Listing 7.15: Kopieren von Daten mit LowLevelStreams

Wichtig ist, in der Methode `copyBuffer()` in Zeile 15 nur so viele Bytes in den Ausgabestrom zu schreiben, wie tatsächlich gelesen wurden. Dies ist insbesondere im letzten Schleifendurchlauf wesentlich.
Den Methoden können beliebige konkrete Bytestreams übergeben werden, da alle Bytestreams (egal ob LowLevel- oder Filterstreams) von `java.io.InputStream` bzw. von `java.io.OutputStream` abgeleitet sind.

Konkrete LowLevelStreams

Im Folgenden werden wichtige konkrete LowLevelStreams im Überblick vorgestellt.

LowLevel-InputStreams

Klasse	Zweck
`java.io.FileInputStream`	liest aus einer Datei
`java.io.PipedInputStream`	liest aus einem `PipedOutputStream`
`java.io.ByteArrayInputStream`	liest aus einem Array von Typ `byte`

Ein wichtiger konkreter LowLevel-Inputstream ist `System.in`. Hier handelt es sich um ein statisches Datenelement der Klasse `java.lang.System`. Da es sich um einen LowLevel-Stream handelt, stehen auf `System.in` direkt nur jene Methoden zur Verfügung, die aus `java.io.InputStream` bekannt sind.

LowLevel-OutputStreams

Klasse	Zweck
`java.io.FileOutputStream`	schreibt in eine Datei
`java.io.PipedOutputStream`	schreibt in einen `PipedOutputStream`
`java.io.ByteArrayOutputStream`	Schreibt in ein dynamisch wachsendes Array von Typ `byte`

Die Klasse `FileOutputStream`

Zum Erzeugen eines Objektes vom Typ `FileOutputStream` stehen die folgenden Kostruktoren zur Verfügung. Bei einer erfolgreichen Objekterzeugung wird die Datei auch geöffnet:

```
public FileOutputStream(String name) throws FileNotFoundException
public FileOutputStream(File file) throws FileNotFoundException
```

Versuchen die über `name` bzw. `file` angegebene Datei zum Schreiben von Daten zu öffnen. Der Inhalt von bereits existierenden Dateien wird überschrieben. Wirft im Fehlerfall eine `FileNotFoundException`, das ist eine Subexception der `IOException`.

```
public FileOutputStream(String name, boolean append)
throws FileNotFoundException
public FileOutputStream(File file, boolean append)
throws FileNotFoundException
```

Wird dem zusätzlichen Parameter `append` der Wert `true` übergeben, so werden die in den Stream geschriebenen Daten an die Datei angehängt.

Das folgende Programm öffnet die Datei `"test.dat"` im aktuellen Arbeitsverzeichnis zum Anhängen von Daten und kopiert mit Hilfe der Methode `copyBuffer()` der oben vorgestellten Klasse `CopyLowLevel` ein Bytefeld in diese Datei.

```
1  import java.io.*;
2
3  public class FileOutput_1 {
4
5    public static void main(String[] args) {
6      byte []b = new byte[256];
7      for(int i = 0; i < b.length; i++)
8        b[i] = (byte) i;
9      FileOutputStream fos = null;
10     ByteArrayInputStream bis = null;
11     try {
12       bis = new ByteArrayInputStream(b);
13       fos = new FileOutputStream("test.dat", true);
14       CopyLowLevel.copyBuffer(bis, fos, 256);
15     }
16     catch(FileNotFoundException e) {
17       System.err.println("Fehler 1: " + e);
18     }
19     catch(IOException e) {
20       System.err.println("Fehler 2: " + e);
21     }
22     finally {
23       try { if(fos != null) fos.close(); }
24       catch(IOException e) {
25         System.err.println("Fehler 3: " + e);
26       }
27     }
28   }
```

Listing 7.16: Beispiel zu `FileOutputStream`

Der `ByteArrayInputStream` braucht nicht geschlossen zu werden.

Die Klasse `java.io.FileInputStream`

Wichtige Konstruktoren:

```
public FileInputStream(String name) throws FileNotFoundException
public FileInputStream(File file) throws FileNotFoundException
```

Öffnet die durch `name` bzw. `file` angegebene Datei zum Lesen. Wirft eine

`FileNotFoundException`, wenn die Datei nicht geöffnet werden kann.

Das folgende Programm liest aus der Datei `"test.dat"` im aktuellen Arbeitsverzeichnis zunächst ein Byte, dann ein Feld von 20 Bytes und schließlich 10 Bytes und schreibt sie ab der Position 5 in das Feld `morebytes`.

```
1  import java.io.*;
2
3  public class FileInput_1 {
4
5    public static void main(String[] args) {
6      FileInputStream fis = null;
7      byte b;
8      byte []bytes = new byte[20];
9      byte []morebytes = new byte[20];
10
11     try {
12       fis = new FileInputStream("test.dat");
13       b = (byte) fis.read();
14       fis.read(bytes);
15       fis.read(morebytes, 5, 10);
16       System.out.println(b);
17       for(byte x : bytes)
18         System.out.print(x + " ");
19       System.out.println();
20       for(byte x : morebytes)
21         System.out.print(x + " ");
22       System.out.println();
23     }
24     catch(FileNotFoundException e) {
25       System.err.println("Fehler 1: " + e);
26     }
27     catch(IOException e) {
28       System.err.println("Fehler 2: " + e);
29     }
30     finally {
31       try { if(fis != null) fis.close(); }
32       catch(IOException e) {
33         System.err.println("Fehler 3: " + e);
34       }
35     }
36   }
37 }
```

Listing 7.17: Beispiel zu `FileInputStream`

Mit der in Listing 2.5. erzeugten Datei erhält man folgende Ausgabe:

```
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
0 0 0 0 0 21 22 23 24 25 26 27 28 29 30 0 0 0 0 0
```

7.5.2 Filterstreams

Filterstreams[3] erweitern die Funktionalität der Lowlevelstreams. Sie sind von den Superklassen `FilterInputStrea` bzw. `FilterOutputStream` abgeleitet und kommunizieren nicht direkt mit physikalischen Datenquellen sondern gemäß dem Schichtenmodell (vgl. Abb. 2.1.) mit anderen Streams.

Die Klassen `FilterInputStream` und `FilterOutputStream`

`java.io.FilterInputStream` ist eine konkrete von `InputStream` abgeleitete Klasse und stellt keine weitere Funktionalität zu Verfügung. Sie bekommt aber beim Instanziieren ein `InputStream`-Objekt

[3] auch oft als Highlevelstreams bezeichnet

als Datenquelle übergeben. Analoges gilt für `FilterOutputStream`.

`protected FilterInputStream(InputStream in)`
Konstruktor der Klasse `FilterInputStream`.

`public FilterOutputStream(OutputStream out)`
Konstruktor der Klasse `FilterOutputStream`.

Wichtige von `FilterInputStream` abgeleitete Klassen sind:

Klasse	Beschreibung
`BufferedInputStream`	Erlaubt gepuffertes Lesen und unterstützt die Methoden `mark()` und `reset()`.
`DataInputStream`	Erlaubt das Lesen primitiver Datentypen und UTFStrings. Implementiert `DataInput`.
`PushbackInputStream`	Erlaubt das Zurückstellen einzelner oder mehrerer gelesener Bytes in den Stream.

Wichtige von `FilterOutputStream` abgeleitete Klassen sind:

Klasse	Beschreibung
`BufferedOutputStream`	Erlaubt gepuffertes Schreiben der Daten.
`DataOutputStream`	Erlaubt das Schreiben primitiver Datentypen und UTFStrings. Implementiert `DataOutput`.
`PrintStream`	Erlaubt das Schreiben der Stringdarstellung beliebiger Daten.

Die Klasse `java.io.PrintStream`

Diese Klasse stellt in mehrfach überladener Weise die von `System.out` bekannten Methoden `print()` und `println()` zur Verfügung. `System.out` und `System.err` sind Instanzen dieser Klasse. Die Methoden dieser Klasse werfen keine IOExceptions, sondern setzen bei Auftreten eines Fehlers ein internes Flag. Dieses kann mit der booleschen Methode `checkError()` abgefragt werden. Außerdem kann mit Hilfe eines zusätzlichen Parameters im Konstruktor die Eigenschaft Autoflush eingestellt werden.

Die Klassen `DataInputStream` und `DataOutputStream`

Die Klasse `DataInputStream` implementiert das Interface `DataInput` und die Klasse `DataOutputStream` das Interface `DataOutput`. Damit stehen neben den grundlegenden Lese- und Schreibemethoden auch Methoden zum binären Lesen und Schreiben primitiver Typen und zum Lesen und Schreiben von UTFStrings zur Verfügung (vgl. `RandomAccessFile`).
Im folgenden Beispiel werden zuerst 5 Shorts (also 10 Byte) in eine Datei `daten.dat` im aktuellen Arbeitsverzeichnis geschrieben. Danach werden solange Integerwerte aus der Datei gelesen, bis die Lesemethode `readInt()` eine `EOFException`[4] wirft. Die Methode `readInt()` kann bei Erreichen des Streamendes nicht den Wert -1 liefern, da -1 auch ein gültiger gelesener Wert sein kann. Statt dessen wird bei Erreichen des Dateiendes eine `EOFException` geworfen. Durch die `try-finally`-Strukturen in der Schreib- und in der Lesemethode wird sicher gestellt, dass die Datenströme (und damit die darunterliegende Datei) auch geschlossen wird, wenn vorher eine `IOException` auftritt.

```
1 import java.io.*;
2
3 public class DataInputOutput {
4
5   public static void main(String[] args) {
6     short []f = {0,10,-1,-1,3};
7     DataInputOutput o = new DataInputOutput();
8     try {
9       o.writeShorts("daten.dat", f);
```

[4] Eine Subklasse von `IOException`

```
      o.readInts("daten.dat");
    }
    catch(IOException e) {
      System.out.println("Fehler in main: " + e);
    }
  }

  public void writeShorts(String file, short []f) throws IOException {
    FileOutputStream fos = null;
    DataOutputStream dos = null;
    try {
      fos = new FileOutputStream(file);
      dos = new DataOutputStream(fos);
      for(short s : f)
        dos.writeShort(s);
    }
    finally {
      if(dos != null) dos.close();
    }
  }

  public void readInts(String file) throws IOException {
    FileInputStream fis = null;
    DataInputStream dis = null;
    try {
      fis = new FileInputStream(file);
      dis = new DataInputStream(fis);
      while(true) {
        try {
          int i = dis.readInt();
          System.out.print(i + " ");
        }
        catch(EOFException e) {
          System.out.println();
          throw new IOException("Dateiende erreicht");
        }
      }
    }
    finally {
      if(dis != null) dis.close();
    }
  }
}
```

Listing 7.18: Beispiel zu `DataInputStream` und `DataOutputStream`

Das obige Programm erzeugt folgende Ausgabe:

```
10 -1
Fehler in main: java.io.IOException: Dateiende erreicht
```

Beispiel

Im folgenden Beispiel wird demonstriert, wie man den Zusatnd eines Javaobjektes in ein ByteArray verwandeln kann und umgekehrt aus einem solchen Bytearray wieder eine Instanz erzeugen kann. Dabei werden primitive Variable binär in das bytearray geschrieben, während der String `str` immer mit 64 Byte Länge (gesteuert durch die Konstante `STR_LEN`) in das Bytearray geschrieben wird. Fehlende Zeichen werden mit Nullen aufgefüllt, überschüssige Zeichen werden abgeschnitten.

```
package inout;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
```

```
6  import java.io.DataOutputStream;
7  import java.io.IOException;
8  import java.util.Arrays;
9  import java.util.Objects;
10
11 public class Data {
12   // Laenge des Strings str im ByteArray
13   private final static int STR_LEN = 64;
14
15   private int i;
16   private String str;
17   private double d;
18
19   public Data(int i, String str, double d) {
20     this.i = i;
21     this.str = str;
22     this.d = d;
23   }
24
25   /**
26    * Konvertieren das this-Objekt in ein Bytearray
27    * i belegt immer 4 Byte (binaer)
28    * str belegt STR_LEN Byte, wird mit '\u0000' aufgefuellt bzw. abgeschnitten
29    * d belegt immer 8 Byte (binaer)
30    * @return das Bytearray mit den Daten, im Fehlerfall null
31    */
32   public byte[] getBytes() {
33     byte []result = null;
34     ByteArrayOutputStream bos = new ByteArrayOutputStream();
35     try (DataOutputStream dos = new DataOutputStream(bos)) {
36       dos.writeInt(i);
37       byte []str1 = Arrays.copyOf(str.getBytes(), STR_LEN);
38       dos.write(str1);
39       dos.writeDouble(d);
40       result = bos.toByteArray();
41     }
42     catch(IOException e) {
43       System.out.println("Fehler: " + e);
44     }
45     return result;
46   }
47
48   /**
49    * Erzeugt aus dem Bytearray b eine Data-Instanz
50    * @param b das Bytearray
51    * @return die erzeugte Data-Instanz, im Fehlerfall null
52    */
53   public static Data getDataInstance(byte []b) {
54     Data result = null;
55     try(DataInputStream dis = new DataInputStream(new ByteArrayInputStream(b))) {
56       int i = dis.readInt();
57       byte []str1 = new byte[STR_LEN];
58       dis.readFully(str1);
59       String str = new String(str1);
60       int idx = str.indexOf('\u0000');
61       if(idx != -1) {
62         str = str.substring(0, idx);
63       }
64       double d = dis.readDouble();
65       result = new Data(i, str, d);
66     }
67     catch(IOException e) {
68       System.out.println("Fehler: " + e);
```

```
69      }
70
71      return result;
72    }
73
74    @Override
75    public String toString() {
76      return "Data{" + "i=" + i + ", str=" + str + ", d=" + d + '}';
77    }
78
79    @Override
80    public int hashCode() {
81      int hash = 7;
82      hash = 67 * hash + this.i;
83      hash = 67 * hash + Objects.hashCode(this.str);
84      hash = 67 * hash + (int) (Double.doubleToLongBits(this.d) ^ (Double.
            doubleToLongBits(this.d) >>> 32));
85      return hash;
86    }
87
88    @Override
89    public boolean equals(Object obj) {
90      if (obj == null) {
91        return false;
92      }
93      if (getClass() != obj.getClass()) {
94        return false;
95      }
96      final Data other = (Data) obj;
97      if (this.i != other.i) {
98        return false;
99      }
100     if (!Objects.equals(this.str, other.str)) {
101       return false;
102     }
103     if (Double.doubleToLongBits(this.d) != Double.doubleToLongBits(other.d)) {
104       return false;
105     }
106     return true;
107   }
108
109   public static void main(String[] args) {
110     Data d = new Data(12, "Testprogramm      ", 23.678);
111     System.out.println(d);
112     byte []b = d.getBytes();
113     Data d1 = Data.getDataInstance(b);
114     System.out.println(d1);
115     System.out.println(d.equals(d1));
116   }
117 }
```

Listing 7.19: Konvertierung eines Objektes in ein ByteArray `Data.java`

Ausgabe:

```
Data{i=12, str=Testprogramm      , d=23.678}
Data{i=12, str=Testprogramm      , d=23.678}
true
```

7.6 Charakterstreams

Charakterstreams (Reader und Writer) dienen zum Schreiben und Lesen von Unicodezeichen, also Zeichen vom Typ `char`. Sie transportieren daher 16-bit-Größen (ganzzahlige Werte im Bereich 0-

65535) und sind bis auf diesen Unterschied vergleichbar mit Input- und Outputstreams. Die LowLevel-Variante kommuniziert mit konkreten I/O-Geräten, die Filtervariante mit LowLevel-Varianten oder anderen HighLevel-Varianten.

7.6.1 Lowlevel-Charakterstreams

Alle Lowlevel-Charakterstreamklassen sind von `java.io.Reader` (zum Lesen) und `java.io.Writer` (zum Schreiben) abgeleitet. Es handelt sich dabei um abstrakte Klassen, die mit Hilfe von Subklassen konkretisiert werden.

Die Klasse `java.io.Reader`

Die wichtigsten Methoden dieser Klasse sind:

`protected Reader()`
Parameterloser Konstruktor.

`public int read() throws IOException`
Liefert das nächste `char` bzw. -1 bei Streamende.

`public int read(char[] cbuf) throws IOException`
Versucht so viele Charakter aus dem Stream zu lesen um das Feld `cbuf` zu füllen. Liefert die Anzahl der gelesenen Charakter bzw. -1 bei Streamende.

`public abstract int read(char[] cbufb, int off, int len)`
`throws IOException`
Versucht `len` viele Charakter aus dem Stream zu lesen und in das Feld `cbuf` ab der Position `off` zu schreiben. Liefert die Anzahl der gelesenen Charakter bzw. -1 bei Streamende.

`public long skip(long n) throws IOException`
Versucht `n` Charakter des Eingabestreams zu überspringen. Liefert die Anzahl der tatsächlich übersprungenen Charakter.

`public boolean ready() throws IOException`
Liefert `true`, wenn der nächste Lesevorgang nicht blockt.

`public void close() throws IOException`
Schließt den Stream.

`public void mark(int readlimit) throws IOException`
Setzt eine Marke, zu der mit `reset()` zurückgesprungen werden kann, solange nicht mehr als `readlimit` weitere Charakter aus dem Stream gelesen wurden. Nur gepufferte Reader unterstützen diese Operation.

`public void reset() throws IOException`
Springt zu einer allfällig gesetzten Marke.

`public boolean markSupported()`
Liefert `true`, wenn der Stream markieren unterstützt.

Die Klasse `java.io.Writer`

Die wichtigsten Methoden dieser Klasse sind:

`protected Writer()`
Parameterloser Konstruktor.

`public void write(int c) throws IOException`
Schreibt die niederwertigsten 16 Bit von `c` in den Stream.

`public void write(char[] cbuf) throws IOException`
Schreibt alle Zeichen des Arrays `cbuf` in den Stream.

`public abstract void write(char[] cbuf, int off, int len) throws IOException`
Schreibt `len` viele Zeichen ab der Startposition `off` von `cbuf` in den Stream.

`public void write(String str) throws IOException`
Schreibt den String `str` in den Stream.

`public void write(String str, int off, int len) throws IOException`
Schreibt `len` viele Zeichen ab der Startposition `off` von `str` in den Stream.

`public abstract void flush() throws IOException`
Flusht den Stream und alle mit ihm verketteten Streams.

`public void close() throws IOException`
Schließt den Stream.

Konkrete LowLevel-Reader

Klasse	Zweck (liest aus)
`FileReader`	einer Datei
`PipedReader`	einem korrespondierenden `PipedWriter`.
`StringReader`	einem String.
`CharArrayReader`	einem Feld vom Typ `char`.

Konkrete LowLevel-Writer

Klasse	Zweck (schreibt in)
`FileWriter`	eine Datei
`PipedWriter`	einen korrespondierenden `PipedReader`.
`StringWriter`	einen (dynamisch wachsenden) String.
`CharArrayWriter`	ein (dynamisch wachsendes) Feld vom Typ `char`.

7.6.2 Filter - Charakterstreams

Die folgende Tabelle gibt eine Übersicht über die wichtigsten Filter-Charakterstreams.

Klasse	Zweck
`BufferedReader`	Dient zum gepufferten Lesen. Hat eine Methode `readLine()` zum zeilenorientierten Lesen von Strings.
`BufferedWriter`	Dient zum gepufferten Schreiben. Hat eine Methode `newLine()` zum Schreiben von systemunabhängigen Zeilentrennzeichen.
`PrintWriter`	Erlaubt mit Hilfe von `print()` und `println()` das Schreiben der Stringdarstellung beliebiger Ausdrücke.

Die Klasse `java.io.BufferedReader`

Konstruktor:

```
public BufferedReader(Reader in)
```

Hier handelt es sich um die einzige Streamklasse, die mit Hilfe der Methode `readLine()` das zeilenorientierte Lesen von Texten unterstützt:

```
public String readLine() throws IOException
```
Liest eine Textzeile bis zum nächsten `'\n'`, `'\r'` oder `"\r\n"`. Liefert die gelesene Zeile als String (ohne Zeilentrennzeichen) bzw. `null` bei Streamende.

Das folgende Programmsegment zeigt, wie man mit Hilfe eines BufferedReaders `br` Textzeilen bis zum Streamende liest und verarbeitet:

```
1 String s;
2 while((s = br.readLine()) != null) {
3   // s verarbeiten
4 }
```

Listing 7.20: Verwendung von `readLine()`

Seit Java 7 kann ein `BufferedReader` auch über die statische Methode `newBufferedReader()` der Klasse `java.nio.files.Files` erzeugt werden:

```
public static BufferedReader newBufferedReader(Path path, Charset cs)
                                                 throws IOException
```

Die Klasse Files stellt auch die folgende Methode bereit, mit deren Hilfe alle Zeilen aus einem Chracterstream gelesen werden können:

```
public static List<String> readAllLines(Path path, Charset cs)
                                         throws IOException
```

Die Klasse `java.io.PrintWriter`

Diese Klasse stellt in mehrfach überladener Weise die von `java.io.PrintStream` bekannten Methoden `print()` und `println()` zur Verfügung. Die Methoden dieser Klasse werfen keine IOExceptions, sondern setzen bei Auftreten eines Fehlers ein internes Flag. Dieses kann mit der boolschen Methode `checkError()` abgefragt werden. Außerdem kann mit Hilfe eines zusätzlichen Parameters im Konstruktor die Eigenschaft Autoflush eingestellt werden.
Es handelt sich hier um einen Writer, dem auch direkt ein Objekt vom Typ `java.io.OutputStream` übergeben werden darf:

```
public PrintWriter(OutputStream out)
public PrintWriter(OutputStream out, boolean autoFlush)
```
Konstruktoren der Klasse `PrintWriter`, denen direkt ein `OutputStream` übergeben werden darf. Zur Konvertierung der Bytes in Unicodes wird das Standardencoding des jeweiligen Betriebssystems verwendet.

7.6.3 Konvertierung von Bytestreams in Charakterstreams

Oft ist es notwendig, Bytestreams in Charakterstreams zu konvertieren. Dabei müssen die 256 verschiedenen Bytewerte auf 256 der insgesamt 65536 Charakterwerte abgebildet werden und umgekehrt. Dies geschieht mit Hilfe eines sogenannten Charsets. In der Regel wird das Standardcharset des jeweiligen Betriebssystems verwendet. Möchte man ein spezielles Charset verwenden, so bedient man sich einer der beiden Brückenklassen.

Die Klasse `java.io.InputStreamReader`

Diese Klasse dient zur Konvertierung eines InputStreams in einen Reader. Sie ist selbst vom Typ `java.io.Reader` und akzeptiert im Konstruktor einen InputStream:

`public InputStreamReader(InputStream in)`
Konvertiert `in` in einen Reader und verwendet dabei das Standardcharset.

`public InputStreamReader(InputStream in, String charsetName)`
`throws UnsupportedEncodingException`
Konvertiert `in` in einen Reader und verwendet dabei das durch `charsetName` spezifizierte Charset.

`public InputStreamReader(InputStream in, Charset cs)`
Konvertiert `in` in einen Reader und verwendet dabei das durch `cs` spezifizierte Charset.

Die Klasse `java.io.OutputStreamWriter`

Diese Klasse dient zur Konvertierung eines OutputStream in einen Writer. Sie ist selbst vom Typ `java.io.Writer` und schreibt in den im Konstruktor übergebenen OutputStream:

`public OutputStreamWriter(OutputStream out)`
Schreibt Charakter nach `out` und verwendet dabei das Standardcharset.

`public OutputStreamWriter(OutputStream out, String charsetName)`
`throws UnsupportedEncodingException`
Schreibt Charakter nach `out` und verwendet dabei das durch `charsetName` spezifizierte Charset.

`public OutputStreamWriter(OutputStream in, Charset cs)`
Schreibt Charakter nach `out` und verwendet dabei das durch `cs` spezifizierte Charset.

Beispiel

Das folgende Beispiel liest zeilenweise von `System.in` und schreibt nach `System.out`. Dabei können über die Kommandozeile für das Lesen und Schreiben verschiedene Charsets angegeben werden. Zum Vergleich wird auch der gelesene String direkt auf `System.out` geschrieben.

```
1  import java.io.*;
2  import java.util.*;
3
4  public class Konvertierung {
5
6    public static void main(String[] args) {
7      Map<String,String> hm = new HashMap<String,String>();
8      for(int i = 0; i < args.length; i++) {
9        if(args[i].startsWith("-") && args.length > i+1)
10         hm.put(args[i], args[++i]);
11     }
12     String charsetin = hm.get("-i");
13     String charsetout = hm.get("-o");
14     InputStreamReader r = null;
15     OutputStreamWriter w = null;
16     if(charsetin != null) {
17       try {
18         r = new InputStreamReader(System.in, charsetin);
19       }
20       catch(UnsupportedEncodingException e) {
21         System.err.println("Ungueltiges Charset(in): " + charsetin);
22         System.exit(1);
23       }
24     }
25     else
26       r = new InputStreamReader(System.in);
27     if(charsetout != null) {
28       try {
29         w = new OutputStreamWriter(System.out, charsetout);
30       }
```

```
31        catch(UnsupportedEncodingException e) {
32          System.err.println("Ungueltiges Charset(out): " + charsetout);
33          System.exit(2);
34        }
35      }
36      else
37        w = new OutputStreamWriter(System.out);
38
39      System.out.println("Charset(in) : " + r.getEncoding());
40      System.out.println("Charset(out): " + w.getEncoding());
41      BufferedReader br = new BufferedReader(r);
42      PrintWriter pw = new PrintWriter(w, true);
43      String line = null;
44      try {
45        while((line = br.readLine()) != null) {
46          System.out.println("1: " + line);
47          pw.println("2: " + line);
48        }
49      }
50      catch(IOException e) {
51        System.err.println("Fehler 1: " + e);
52      }
53    }
54  }
```

Listing 7.21: Konvertierung Bytestreams ⇔ Charakterstreams

Das obige Programm erzeugt in einer WindowsXP-Shell folgende Ausgabe:

```
java Konvertierung -o iso-8859-1 -i iso-8859-1
Charset(in) : ISO8859_1
Charset(out): ISO8859_1
abcd(){}
1: abcd(){}
2: abcd(){}
äöüÄÖÜß
1: ??????ß
2: äöüÄÖÜß
```

7.7 Serializierung von Objekten

In diesem Abschnitt wird das Schreiben und Lesen von Objekten behandelt. Zu diesem Zweck müssen die Objekte in eine Bytefolge zerlegt und aus einer solchen wieder rekonstruiert werden. Diese Vorgänge werden als Serializierung bzw. Deserializierung von Objekten bezeichnet. Java unterstützt diese Technik durch zwei Streamklassen, `ObjectOutputStream` und `ObjectInputStream`.

7.7.1 Die Klassen `ObjectOutputStream` und `ObjectInputStream`

Die Klasse `java.io.ObjectOutputStream`

Mit Hilfe dieser Klasse werden Objekte serializiert. Die Klasse implementiert das Interface `java.io.DataOutput`, d.h. es werden auch Methoden zum Schreiben primitiver Datentypen und UTFStrings unterstützt. Die wichtigsten Methoden sind:

`public ObjectOutputStream(OutputStream out) throws IOException`
Dem Konstruktor wird der darunterliegende OutputStream `out` übergeben.

`public final void writeObject(Object obj) throws IOException`
Schreibt das serializierbare Objekt `obj` in den Stream. Die Methode wirft die von `IOException` abgeleitete `NotSerializableException`, wenn versucht wird, ein nicht serializierbares Objekt zu schreiben.

`public void defaultWriteObject() throws IOException`
Schreibt alle nicht transienten und nicht statischen Datenfelder einer Klasse `A` in den Stream. Darf nur von der Methode `writeObject()` dieser Klasse `A` aufgerufen werden.

`public void reset()`
Setzt den Status bereits serializierter Objekte zurück. Bereits geschriebene Objekte werden nach dem Aufruf dieser Methode also neu serialisiert.

`public void flush() throws IOException`
Flusht den Stream.

`public void close() throws IOException`
Schließt den Stream.

Der folgende Codeabschnitt zeigt (ohne Exceptionhandling) die prinzipielle Arbeitsweise:

```
1 FileOutputStream out = new FileOutputStream("test");
2 ObjectOutputStream oos = new ObjectOutputStream(out);
3 oos.writeInt(1234);
4 oos.writeObject("Heute");
5 oos.writeObject(new Date());
6 oos.flush();
7 oos.close();
```

Listing 7.22: Serializieren von Objekten

Die Klasse `java.io.ObjectInputStream`

Mit Hilfe dieser Klasse werden Objekte deserializiert. Die Klasse implementiert das Interface `java.io.DataInput`, d.h. es werden auch Methoden zum Lesen primitiver Datentypen und UTFStrings unterstützt. Die wichtigsten Methoden sind:

`public ObjectInputStream(InputStream in) throws IOException`
Dem Konstruktor wird der darunterliegende InputStream `in` übergeben.

`public final Object readObject() throws IOException, ClassNotFoundException`
Liest das nächste Objekt aus dem Stream. Gelesen werden die Klasseninformation und alle nicht transienten und nicht statischen Datenfelder. Referenzierte Objekte werden rekursiv gelesen.

`public void defaultReadObject() throws IOException`
Deserializiert alle nicht transienten und nicht statischen Datenfelder eines Objektes einer Klasse `A`. Darf nur von der Methode `readObject()` dieser Klasse `A` aufgerufen werden.

`public void close() throws IOException`
Schließt den Stream.

Der folgende Codeabschnitte zeigt (ohne Exceptionhandling) die prinzipielle Arbeitsweise:

```
1 FileInputStream in = new FileInputStream("test");
2 ObjectInputStream ois = new ObjectInputStream(in);
3 int i = ois.readInt();
4 String today = (String) ois.readObject();
5 Date date = (Date) ois.readObject();
6 ois.close();
```

Listing 7.23: Deserializieren von Objekten

7.7.2 Serializierbare Objekte

Implementieren von `Serializable`

Nicht alle Objekte sind serializierbar. Damit die Objekte einer Klasse serializierbar sind, muss diese Klasse das Interface `java.io.Serializable` implementieren. Dabei handelt es sich um ein sog. Marker-Interface[5] ohne Methoden, dessen Zweck darin besteht, Klassen zu kennzeichnen, deren Objekte serialisierbar sind.

```
1 public interface Serializable {}
```

Listing 7.24: Das Interface `java.io.Serializable`

Eine einfache Klasse, deren Objekte serializierbar sind, hat also die folgende Gestalt:

```
1 public MyClass implements Serializable {
2   // keine Methoden notwendig
3 }
```

Listing 7.25: Serializierbare Klasse

Zur Serializierung eines Objektes dieser Klasse werden die Methoden `writeObject()` und `readObject()` der Klassen `ObjectOutputStream` und `ObjectInputStream` verwendet. Dabei werden serializiert:

- Klasseninformationen
- Rekursiv die Werte aller nicht statischen und nicht transienten Mitglieder, also die Werte jener Instanzvariablen, die nicht mit dem Modifier `transient` deklariert wurden. Beinhaltet ein Objekt Instanzvariablen, die andere Objekte referenzieren, so ist gewährleistet, dass rekursiv auch die referenzierten Objekte mitserializiert werden.

Es ist nicht notwendig, dass innerhalb einer Vererbunghierarchie alle Klassen serializierbar sind[6]. Ist die unmittelbare Superklasse `A` einer serializierbaren Klasse `B` nicht serializierbar, so ist sicherzustellen, dass `A` einen parameterlosen Konstruktor bereitstellt. Dieser wird bei der Deserializierung von `B` aufgerufen. Wenn auch `A` serializierbar ist, so wird bei der Deserializierung von `B` kein Konstruktor von `A` aufgerufen.

Referenziert eine Klasse `A` im Rahmen der hasA-Beziehung andere Objekte, so ist `A` nur dann serializierbar, wenn auch die Klassen aller Objekte, welche von `A` aus referenziert werden, serializierbar sind. Das folgende Beispiel demonstriert diesen Sachverhalt:

```
1  class Wheels {
2  }
3
4  class Car1 implements Serializable {
5     Wheels w = null;
6  }
7
8  class Car2 implements Serializable {
9     Wheels w = new Wheels();
10 }
```

Listing 7.26: Serializierbare Klasse

In obigem Beispiel ist zwar die Klasse `Car1` serializierbar (`w` verweist auf `null`, es muss beim Deserializieren von `Car1` also kein `Wheels`-Objekt erzeugt werden), nicht aber die Klasse `Car2` (hier muss beim Deserializieren ein `Wheels`-Objekt erzeugt werden).

[5] auch Tag-Interface
[6] `java.lang.Object` ist nicht serializierbar

Anpassen der Serializierung

Es ist auch möglich, die Serializierung eines bestimmten Typs den eigenen Bedürfnissen anzupassen. Zu diesem Zweck verwendet man in der Klasse die folgenden Methoden:

```
private void writeObject(ObjectOutputStream s) throws IOException

private void readObject(ObjectInputStream s)
             throws IOException, ClassNotFoundException
```

Hier können auch statische Variablen serializiert und andere Berechnungen durchgeführt werden. Diese Methoden müssen exakt die oben angegebene Schnittstelle besitzen (speziell also den Zugriff `private` besitzen) und können die Standardmethoden zum Lesen und Schreiben benutzen:

```
private void writeObject(ObjectOutputStream s) throws IOException {
    s.defaultWriteObject();
    // angepasster Serializierungscode
}

private void readObject(ObjectInputStream s)
            throws IOException, ClassNotFoundException {
    s.defaultReadObject();
    // angepasster Serializierungscode
}
```

Listing 7.27: angepasste Methoden `writeObject()` und `readObject()`

7.7.3 Beispiel

Im folgenden Beispiel werden ObjectStreams mit Hilfe von PipeStreams verkettet. Dabei wird zunächst ein Objekt der Klasse `Test` geschrieben, verändert und nocheinmal geschrieben.

```
import java.io.*;

public class ObjectStreams_1 {

  public static void main(String[] args) {
    try {
      PipedInputStream pis = new PipedInputStream();
      PipedOutputStream pos = new PipedOutputStream(pis);

      ObjectOutputStream oos = new ObjectOutputStream(pos);
      ObjectInputStream ois = new ObjectInputStream(pis);

      System.out.println("Streams erfolgreich erzeugt");
      Test x = new Test(10,20), y = null;
      oos.writeObject(x);
      System.out.println("Erfolgreich geschrieben: " + x);
      y = (Test) ois.readObject();
      System.out.println("Erfolgreich gelesen:     " + y);
      // Das Objekt x wird nun veraendert
      x.setA(30);
      oos.reset();
      oos.writeObject(x);
      System.out.println("Erfolgreich geschrieben: " + x);
      y = (Test) ois.readObject();
      System.out.println("Erfolgreich gelesen:     " + y);
      oos.close();
      ois.close();
    }
    catch (IOException ioe) {
      System.err.println("Fehler 1: " + ioe);
    }
    catch (ClassNotFoundException cnfe) {
      System.err.println("Fehler 2: " + cnfe);
    }
  }
}

class Test implements Serializable {

  private int a;
  private transient int b;

  public int getA() { return a; }
  public void setA(int a) { this.a = a; }
```

```
45   public int getB() { return b; }
46   public void setB(int b) { this.b = b; }
47
48   public Test(int a, int b) {
49     this.setA(a);
50     this.setB(b);
51   }
52
53   @Override
54   public String toString() {
55     return String.format("[a=%4d, b=%4d]", this.a, this.b);
56   }
57 }
```

Listing 7.28: Beispiel 1 - Serializieren

Das Programm erzeugt folgende Ausgabe:

```
Streams erfolgreich erzeugt
Erfolgreich geschrieben: [a=  10, b=  20]
Erfolgreich gelesen:     [a=  10, b=   0]
Erfolgreich geschrieben: [a=  30, b=  20]
Erfolgreich gelesen:     [a=  30, b=   0]
```

In obigem Beispiel sind die folgenden Punkte zu beachten:

(1) Wichtig ist, dass der ObjectOutputStream-Konstruktor vor dem ObjectInputStream-Konstruktor aufgerufen wird. Der ObjectOutputStream-Konstruktor schreibt Initialisierungsdaten, die von ObjectInputStream-Konstruktor gelesen werden müssen. Andernfalls blockt das Programm. Bei gepufferten Streams ist es notwendig, den OutputStream nach Erzeugung zu flushen.

(2) Das mit `x` referenzierte Objekt wird 2 mal geschrieben. Zuerst in Zeile 15, dann wird es verändert und in Zeile 22 erneut geschrieben. Würde der ObjectOutputstream dazwischen nicht in Zeile 21 zurückgesetzt, so würde das Objekt nicht neu serialisiert und die Änderung würde nicht geschrieben. Mit auskommentierter Zeile 21 hätte das Programm das folgende Laufzeitverhalten:

```
Streams erfolgreich erzeugt
Erfolgreich geschrieben: [a=  10, b=  20]
Erfolgreich gelesen:     [a=  10, b=   0]
Erfolgreich geschrieben: [a=  30, b=  20]
Erfolgreich gelesen:     [a=  10, b=   0]
```

(3) Die transiente Instanzvariable `b` wird nicht serialisiert und beim Deserializieren auf den Standwert 0 gesetzt. Dies kann verhindert werden, indem man in der Klasse `Test` mit Hilfe der Methoden `readObject()` und `writeObject()` das Serializierungsverhalten anpasst:

```
1  private void writeObject(ObjectOutputStream oos) throws IOException {
2    System.out.println("writeObject von Test   : " + this);
3    oos.defaultWriteObject();
4    oos.writeInt(this.getB());
5  }
6
7  private void readObject(ObjectInputStream ois) throws IOException,
       ClassNotFoundException {
8    System.out.println("readObject von Test    : " + this);
9    ois.defaultReadObject();
10   this.setB(ois.readInt());
11   System.out.println("readObject von Test    : " + this);
12 }
```

Listing 7.29: Beispiel 1 - Anpassen der Serializierung

Nach Aufnahme dieser beiden Instanzmethoden in die Klasse `Test` zeigt das Programm das folgende Laufzeitverhalten:

```
Streams erfolgreich erzeugt
writeObject von Test   : [a=  10, b=  20]
Erfolgreich geschrieben: [a=  10, b=  20]
readObject von Test    : [a=   0, b=   0]
readObject von Test    : [a=  10, b=  20]
Erfolgreich gelesen:     [a=  10, b=  20]
writeObject von Test   : [a=  30, b=  20]
Erfolgreich geschrieben: [a=  30, b=  20]
readObject von Test    : [a=   0, b=   0]
readObject von Test    : [a=  30, b=  20]
Erfolgreich gelesen:     [a=  30, b=  20]
```

Kapitel 8

Erstellung graphischer Benutzerschnittstellen mit JavaFX

Objektorientierte Sprachen wie JAVA stellen zur Erstellung von Programmen mit grafischem Userinterface (GUI) in der Regel umfangreiche Klassenbibliotheken zur Verfügung. Die Fähigkeiten einer solchen Bibliothek lassen sich grob in folgende 4 Gruppen unterteilen:

- Elemente der Benutzeroberfläche: Fenster, Menüs, Schaltflächen, Kontrollfelder, Textfelder, Bildlaufleisten und Listenfelder und ihre Anordnung mit Hilfe von Containern und Layout-Managern.
- Steuerung des Programmablaufs auf der Basis von Nachrichten für die Handhabung von System- und Benutzerereignissen.
- Primitivoperationen zum Zeichnen von Linien oder Füllen von Flächen und zur Grafikausgabe von Text.
- Fortgeschrittene Grafikfunktionen zur Darstellung und Manipulation von Bitmaps und zur Ausgabe von Sound.

8.1 AWT und Swing

Zur Erstellung von Programmen mit grafischer Benutzerschnittstellen gibt es in Jave mehrere Möglichkeiten, die sich historisch entwickelt haben. Die beiden wichtigsten sind:

1. **Abstract Windowing Toolkit - AWT**
   Seit dem JDK 1.x gibt es das AWT zur Erzeugung graphischer Benutzeroberflächen. Nachteile sind:
   - Alle Fenster und Dialogelemente werden von dem darunterliegenden Betriebssystem zur Verfügung gestellt (native oder schwergewichtige Grafikelemente). Daher ist es schwierig, plattformübergreifend ein einheitliches Look and Feel zu realisieren.
   - Die Eigenarten jedes einzelnen Fenstersystems waren für den Anwender unmittelbar zu spüren.
   - Im AWT gibt es nur eine Grundmenge an Dialogelementen (Controls, die auf allen unterstützen Betriebssystemen existieren), mit denen sich aufwändige grafische Benutzeroberflächen nicht oder nur mit sehr viel Zusatzaufwand realisieren ließen. Spezielle Controls wie Tabellen oder Trees sind unter AWT nicht vorhanden.
2. **Swing**
   Swing gibte es seit Java 2 und es ist Bestandteil der JFC (Java Foundation Classes). In Swing sind nur die Fenster selbst schwergewichtig (nativ) und werden vom verwendeten Fenstersystem zur Verfügung gestellt. Alle anderen GUI - Elemente werden von Swing selbst gezeichnet. Damit ergeben sich folgende Vorteile:
   - Plattformspezifische Besonderheiten fallen weg , daraus ergibt sich eine einfachere Implementierung der Oberflächen.

- Einheitliche Bedienung auf unterschiedlichen Betriebssystemen.
- Es ist nicht nur die Schnittmenge der Komponenten aller Plattformen verfügbar.
- Swing unterstützt Pluggable Look and Feel (Umschaltung des Aussehens einer Anwendung zur Laufzeit - Windows, Motif, Metal, ...)

Nachteil:

- Swinganwendungen sind ressourcenhungrig. Das Zeichnen der Komponenten erfordert viel CPU-Leistung und Hauptspeicher (Unterstützung durch DirectDraw, OpenGL).

8.2 Grundlagen von JavaFX

Die moderntes Art grafische Benutzerschnittstellen und Java zu entwickeln stellt JavaFX dar. Java FX ist seit der JDK-Version in die Java Standardedition (SE) integriert und wird mittelfristig Swing ersetzen. Key-Features von JavaFX sind:

- JavaFX verwendet ähnlich zu Swing nur native Fenster. Ein natives Fenster in JavaFX heißt **Stage**.
- Jede Stage beinhaltet eine sogannte **Scene**, in der die Komponeneten der grafischen Oberfläche gespeichert sind.
- Jede Scene wird durch einen sogenannten **Scenegraph** beschrieben. Dabei handelt es sich um eine Baumstruktur, die aus einzelnen Knoten und Blättern besteht.
- Scenengraphen lassen sich programmatisch oder mit Hilfe eines SceneBuilders (grafischer Designer) erzeugen. Scenengraphen lassen sich auch mit Hilfe von FXML, das ist eine XML-basierte deklarative Sprache beschreiben.
- JavaFX stellt eine Vielzahl von Containern und Controls zur Verfügung, deren Look and Fell mit Hilfe von CSS gestaltet werden kann.
- JavaFX stellt ein einheitliches Eventhandling zur Verfügung, das sein JavaFX 8 auch mit Hilfe von Lambdaexpressions implementiert werden kann.
- JavaFX stellt Properties und ein BindingAPI zur Verfügung, mit dessen Hilfe Eigenschaften von Userkomponeneten (z.B. der Text in einem Textfeld) an Attribute von geschäftsklassen gebunden werden können.

8.2.1 Eine erste Beispielapplikation

Die folgende Abbildung zeigt eine erste Beispielapplikation. An Hand dieser Aoolikation werden die wichtigsten Bestandteile einer FX-Anwendung erklärt:

8.3 Properties und Binding

Properties und Binding sind zwei mächtige Sprach-Mechanismen in JavaFX. Die grundlegende Idee ist, dass Eigenschaften von Objekten nicht in einfachen Instanzvariablen gespeichert werden, sondern mit Hilfe sogenannter Properties. Solche Properties kapseln dabei die eigentlichen Daten und es wird auf diese Daten über die Property zugegriffen. Das hat den Vorteil, dass die Änderung eines Wertes überwacht werden kann und in weiterer Folge können auch zwei Properties aneinander gebunden werden (Binding). Dies bedeutet, dass die Änderung eines Wertes auch die automatische Änderung des anderenmit sich ziehen kann. Diese Aktualisierung erfolgt automatisch und man wird als Entwickler davon entlastet, sogenannten Schaukelcode zu schreiben, der den Zustand interner Geschäftsobjekte (z.B. den Namen einer Person) mit der Oberfläche synchronisiert und umgekehrt.

8.3.1 Properties

Grundlagen

Das Grundlegende Arbeiten mit einer Property wird an Hand eines einfachen Beispiels demonstriert. Die Klasse `javafx.scene.shape.Circle` verwendet speichert unter anderem den Radius (`double`) und wie die Kreisfläche zu füllen ist (`javafx.scene.paint.Paint`). Diese Informationen werden mit Hilfe von 2 Properties verwaltet:

- `DoubleProperty radius`
  Dabei handelt es sich um eine Property, die einen primitiven double-Wert kapselt
- `ObjectProperty<Paint> fill`
  Diese Property kapselt ein Objekt vom Typ `javafx.scene.paint.Paint`

Das folgende Listing zeigt den Zugriff auf diese Properties:

```
1  import javafx.beans.property.DoubleProperty;
2  import javafx.beans.property.ObjectProperty;
3  import javafx.scene.paint.Color;
4  import javafx.scene.paint.Paint;
5  import javafx.scene.shape.Circle;
6
7  public class PropDemo1 {
8
9      public static void main(String[] args) {
10         final Circle c = new Circle(10.0);
11         System.out.println(c);
12         c.setRadius(12.0);
13         System.out.println(c);
14         c.radiusProperty().set(13.0);
15         System.out.println(c);
16         System.out.println(c.radiusProperty());
17         System.out.println("----------------------");
18         DoubleProperty rp = c.radiusProperty();
19         System.out.println("Wert der Property: " + rp.getValue());
20         System.out.println("Wert der Property: " + rp.get());
21         System.out.println("Name der Property: " + rp.getName());
22         System.out.println("Bean der Property: " + rp.getBean());
23         c.setFill(Color.RED);
24         System.out.println("----------------------");
25         ObjectProperty<Paint> fp = c.fillProperty();
26         System.out.println("Wert der Property: " + fp.getValue());
27         System.out.println("Wert der Property: " + fp.get());
28         System.out.println("Name der Property: " + fp.getName());
29         System.out.println("Bean der Property: " + fp.getBean());
30     }
31 }
```

Listing 8.1: Zugriff auf Properties

Ausgabe:

```
Circle[centerX=0.0, centerY=0.0, radius=10.0, fill=0x000000ff]
Circle[centerX=0.0, centerY=0.0, radius=12.0, fill=0x000000ff]
Circle[centerX=0.0, centerY=0.0, radius=13.0, fill=0x000000ff]
DoubleProperty [bean: Circle[centerX=0.0, centerY=0.0, radius=13.0,
                fill=0x000000ff], name: radius, value: 13.0]
----------------------
Wert der Property: 13.0
Wert der Property: 13.0
Name der Property: radius
Bean der Property: Circle[centerX=0.0, centerY=0.0, radius=13.0,
                          fill=0x000000ff]
```

```
----------------------
Wert der Property: 0xff0000ff
Wert der Property: 0xff0000ff
Name der Property: fill
Bean der Property: Circle[centerX=0.0, centerY=0.0, radius=13.0, fill=0xff0000ff]
```

Das obige Beispiel demonstriert den Zugriff auf zwei Properties einer Circle-Instanz `c`. In Zeile 10 wird der Radius mit Hilfe der Methode `setRadius()` auf 12 gesetzt. In Wirklichkeit wird hier nicht eine Instanzvariable `radius` vom Typ `double` verändert, sondern der wert einer Property vom Typ `DoubleProperty`, die eine Doublevariable kapselt. Zeile 14 demonstriert, wie man den Radius direkt über die Property setzen könnte.
Properties finden sich im Paket `javafx.beans.property` Jede Property stellt 4 Methoden zum Zugriff auf ihre Daten bereit, die in obigem Beispiel verwendet wurden:
`get()`, `getValue()`, `getName()`, `getBean()`.
Der Unterschied zwischen `get()` und `getValue()` ist, dass bei einer Property, die einen primitiven Typ kapselt (wie z.B. bei einer `DoubleProperty`) die Methode `get()` den primitiven Typ (also `double`) und die Methode `getValue()` den entsprechenden Wrapper (also `Double`) retourniert. Bei einer Object-Property sind die Rückgabetypen der beiden Methoden gleich.

8.3.2 Verwendung von Listener - Properties beobachten

Alle Properties sind Observale (beobachtbar) gemäß dem Observer Pattern. Dies bedeutet, dass eine Property ihre Beobachter verständigt, wenn ihr Wert ungültig oder geändert wird. Dazu gibt es zwei Listener, die im Folgenden kurz vorgestellt werden:

- **InvalidationListener**

  ```
  @FunctionalInterface
  public interface InvalidationListener {
    void invalidated(Observable observable);
  }
  ```

  Dieses Interface liegt im Paket `javafx.beans` und seine abstrakte Methode `invalidated()` erhält das beobachtbare Objekt, also jene Property, deren Wert gerade ungültig geworden ist.

- **ChangeListener**

  ```
  @FunctionalInterface
  public interface ChangeListener<T> {
   void changed(ObservableValue<? extends T> observable,
                T oldValue, T newValue)
  }
  ```

  Dieses generische Interface liegt im Paket `javafx.beans.value` und seine abstrakte Methode `changed()` erhält das typbehaftete beobachtbare Objekt, also jene Property, deren Wert sich gerade geändert hat. Außerdem werden der alte und der neue Wert übergeben.

  Im Fall einer `IntegerProperty`, `LongProperty`, `FloatProperty` bzw. `DoubleProperty` ist wird der Typstellvertreter `T` mit dem Typ `java.lang.Number` versorgt.

Das folgende Beispiel demonstriert die Verwendung dieser ListenerInterfaces an Hand der DoubleProperty `radius` eines Circle:

```
1 import javafx.scene.shape.Circle;
2
3 /**
4  *
5  * @author reio
6  */
7 public class PropDemo2 {
```

```
8
9   public static void main(String[] args) {
10    final Circle c = new Circle(10.0);
11
12    // Listener registrieren
13    c.radiusProperty().addListener(new InvalidationListener() {
14      @Override
15      public void invalidated(Observable o) {
16        System.out.println("Invalidation detected for: " + o);
17      }
18    });
19
20    c.radiusProperty().addListener(new ChangeListener<Number>() {
21      @Override
22      public void changed(ObservableValue<? extends Number> o, Number oldValue,
            Number newValue) {
23        System.out.println("Change detected for: " + o);
24        System.out.println("Old Value: " + oldValue);
25        System.out.println("New Value: " + newValue);
26      }
27    });
28
29    System.out.println("Circle: " + c.getRadius());
30    c.setRadius(12.3);
31    System.out.println("Circle: " + c.getRadius());
32  }
33
34 }
```

Listing 8.2: Properties beobachten

Ausgabe:

```
Circle: 10.0
Invalidation detected for: DoubleProperty [bean: Circle[centerX=0.0,
   centerY=0.0, radius=12.3, fill=0x000000ff], name: radius, value: 12.3]
Change detected for: DoubleProperty [bean: Circle[centerX=0.0, centerY=0.0,
   radius=12.3, fill=0x000000ff], name: radius, value: 12.3]
Old Value: 10.0
New Value: 12.3
Circle: 12.3
```

Dabei hätten die FunctionalInterfaces, die im Listing mit Hilfe anonymer innerer Klassen konkretisiert wurden auch mit Hilfe von LambdaExpressions formuliert werden können:

```
c.radiusProperty().addListener(o ->
      System.out.println("Invalidation detected for: " + o) );
c.radiusProperty().addListener((o, oldValue, newValue) -> {
      System.out.println("Change detected for: " + o);
           System.out.println("Old Value: " + oldValue);
           System.out.println("New Value: " + newValue); });
```

Ausgabe:

```
Circle: 10.0
Invalidation detected for: DoubleProperty [bean: Circle[centerX=0.0,
  centerY=0.0, radius=12.3, fill=0x000000ff], name: radius, value: 12.3]
Change detected for: DoubleProperty [bean: Circle[centerX=0.0, centerY=0.0,
  radius=12.3, fill=0x000000ff], name: radius, value: 12.3]
Old Value: 10.0
New Value: 12.3
Circle: 12.3
```

Beispiel

Im folgenden Beispiel wird ein einfache Klasse `Counter` erstellt. Dabei wird der Zählerstand in einer IntegerProperty gespeichert. Diese Property wird beobachtet und immer wenn sich der Zählerstand ändert wird der neue Zählerstand auf einem Label aktualisiert.
Das UserInterface hat folgende Gestalt:

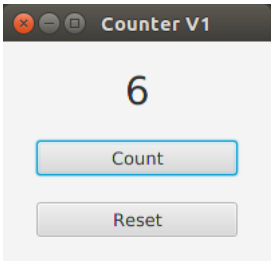


Abbildung 8.1: Properties: CounterApp

Die Geschäftsklasse model.Counter hat folgenden Aufbau:

```
1  package model;
2
3  import javafx.beans.property.IntegerProperty;
4  import javafx.beans.property.SimpleIntegerProperty;
5
6  public class Counter {
7      // Hier wird die IntegerProperty zur kapselung des Zaehlerstandes erszeugt
8      private final IntegerProperty value = new SimpleIntegerProperty(this, "value",
           0);
9
10     // liefert die Integerproperty
11     public final IntegerProperty valueProperty() {
12         return value;
13     }
14
15     // Getter fuer den in der Property gekapselten Zaehlerstand
16     public final int getValue() {
17         return value.get();
18     }
19
20     // Setter fuer den in der Property gekapselten Zaehlerstand
21     public void setValue(int n) {
22         this.value.set(n);
23     }
24
25     // Hochzaehlen des Zaehlers
26     public void count() {
27         this.value.set(this.getValue() + 1);
28     }
29
30     // Zuruecksetzen des Zaehlers
31     public void reset() {
32         this.value.set(0);
33     }
34 }
```

Listing 8.3: Properties: Counter.java

Wie in obigem Listing zu erkennen ist,werden zu einer Property in der Regel drei Zugriffsmethoden benötigt:

- Ein finaler Getter für die eigentliche finale Property
- Ein Getter für den in der Property gekapseten Wert

- Ein Setter für den in der Property gekapselten Wert

Die Zugehörige FXM-Datei sieht wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.text.*?>
<?import javafx.geometry.*?>
<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox alignment="CENTER" maxHeight="-Infinity" maxWidth="-Infinity"
  minHeight="-Infinity" minWidth="-Infinity" prefHeight="300.0" prefWidth="200.0"
  spacing="20.0" xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml
      /1" fx:controller="view.CounterGUIFXMLController">
   <children>
      <Label fx:id="lblValue" text="0">
         <font>
            <Font size="29.0" />
         </font>
      </Label>
      <Button fx:id="btnCount" mnemonicParsing="false" onAction="#onBtnCount"
          prefWidth="150.0" text="Count" />
      <Button fx:id="btnReset" mnemonicParsing="false" onAction="#onButtonReset"
          prefWidth="150.0" text="Reset" />
   </children>
   <padding>
      <Insets bottom="20.0" left="20.0" right="20.0" top="20.0" />
   </padding>
</VBox>
```

Listing 8.4: FXMLzu CounterApp

Der Controller:

```
package view;

import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;package view;

import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import model.Counter;

public class CounterGUIFXMLController implements Initializable {

    @FXML
    private Button btnCount;
    @FXML
    private Button btnReset;

    private Counter counter;
    @FXML
    private Label lblValue;
```

```
28
29      @Override
30      public void initialize(URL url, ResourceBundle rb) {
31          counter = new Counter();
32          counter.valueProperty().addListener(o -> lblValue.setText(Integer.toString(
                counter.getValue())));
33      }
34
35      @FXML
36      private void onBtnCount(ActionEvent event) {
37          counter.count();
38      }
39
40      @FXML
41      private void onButtonReset(ActionEvent event) {
42          counter.reset();
43      }
44  }
```

Listing 8.5: Controller zur CounterApp

Die Methode `invalidated()` im InvalidationListener der valueProperty des Counters wird immer dann aufgerufen,wenn sich der Wert der Property (also der Zählerstand) durch Aufruf der `set()` - Methode verändert hat. Hier wird der Text des Labels an den neuen Zählerstand angepasst.
Die eigentliche Applikation:

```
1   public class CounterApp extends Application {
2
3       @Override
4       public void start(Stage primaryStage) {
5         try {
6           VBox root = FXMLLoader.load(getClass().getResource("../view/CounterGUIFXML.
                fxml"));
7           Scene scene = new Scene(root);
8
9           primaryStage.setTitle("Counter V1");
10          primaryStage.setScene(scene);
11          primaryStage.show();
12        } catch (IOException ex) {
13            Logger.getLogger(CounterApp.class.getName()).log(Level.SEVERE, null, ex);
14        }
15      }
16
17      public static void main(String[] args) {
18        launch(args);
19      }
20  }
```

Listing 8.6: CounterApp.java

Arten von Properties

Das Property-API kennt zwei Arten unterschiedlicher Properties:

- Read/Writable Properties
- Read-Only Properties

Kurz gesagt sind Properties also Wrapper für aktuelle Werte, die nach dem Observer Pattern interessierte Beobachter von der Änderung des Wertes bzw.der Invalidierung benachrichtigen können. Außerdem können Properties über Binding aneinander gekoppelt werden (siehe später). Alle Property-Klassen findet man im Paket `javafx.beans.property`.
Es gibt Properties für die primitiven Typen `boolean`, `int`, `long`, `float` und `double` sowie für `String`. Außerdem gibt es eine generische `ObjectProperty<T>`, die beliebige Objekte kapseln kann. Der

Grundsätzliche Aufbau wird an Hand der `IntegerProperty` erklärt, gilt aber natürlich analog für alle anderen Properties auch.
Die Klasse IntegerProperty ist eine abstrakte Klasse, welche die Klasse ReadOnlyIntegerProperty erweitert und zwei Interfaces inplementiert:

```
public abstract class IntegerProperty
extends ReadOnlyIntegerProperty
implements Property<Number>, WritableIntegerValue
```

Eine konkrete Implementierung dieses abstrakten Typs ist z.B. die Klasse `SimpleIntegerProperty`. Diese stellt im Wesentlichen die folgenden Methoden zur Verfügung:

SimpleIntegerProperty

Konstruktoren:

```
SimpleIntegerProperty()
SimpleIntegerProperty(int initialValue)
SimpleIntegerProperty(Object bean, String name)
SimpleIntegerProperty(Object bean, String name, int initialValue)
```

Dabei ist `initialValue` ein Initalisierungswert, `name` der Name der Property und `bean` das zugehörige Objekt, zu dem die Property gehört.
Wetere Methoden:

`public Object getBean()`
Liefert das Objekt, die Property beinhaltet. Ist kein Objekt (Bean) zugewiesen, wird `null` geliefert.

`public String getName()`
Liefert den Namen der Property, ist kein Name zugewiesen so wird der Leerstring geliefert.

```
public int get()
public void set(int newValue)
```
Getter und Setter auf den gekapselten Wert.

`public Integer getValue()`
Liefert den gekapsten Interger als Wrappertyp.

```
public void addListener(ChangeListener<? super Number> listener)
public void removeListener(ChangeListener<? super Number> listener)
public void addListener(InvalidationListener listener)
public void removeListener(InvalidationListener listener)
```
Diese Methoden dienen zum Registrieren bzw. Abmelden von Invalidation- bzw.ValueChangeListener.

Darüber hinaus gibt es Methoden zum Binden an eine andere Property, diese werden im Abschnitt über Binding vorgestellt.

Read/Write Properties

Die wichtigsten Read/Write-Properties sind in folgendet Tabelle zusammengestellt:

Abstrakte Basisklasse	Konkrete Klasse	gewrappter Typ
`BooleanProperty`	`SimpleBooleanProperty`	`boolean`
`IntegerProperty`	`SimpleIntegerProperty`	`int`
`DoubleProperty`	`SimpleDoubleProperty`	`double`
`StringProperty`	`SimpleStringProperty`	`String`
`ObjectProperty<T>`	`SimpleObjectProperty<T>`	`T` (generischer Typ)

Read-Only Properties

JavaFX stellt auch ReadOnly-Properties zur Verfügung. Solche Properties kapseln zwei Werte, einen der intern verändert werden kann und einen dazu synchronisierten, der nach außen zur Verfügung gestellt wird und nur Getter (keine Setter) zur Verfügung stellt.

Da im obigen Beispiel der Wert des Zählers von außen nicht verändert werden soll, kann man die Property `value` als Read-Only Property definieren. Die Klasse `Counter` muss dann wie folgt abgeändert werden:

```
1  package model;
2
3  import javafx.beans.property.ReadOnlyIntegerProperty;
4  import javafx.beans.property.ReadOnlyIntegerWrapper;
5
6  public class CounterReadOnly {
7
8    private final ReadOnlyIntegerWrapper value = new ReadOnlyIntegerWrapper(this, "
         value", 0);
9
10   // auf den Zaehlerstand wird nur ein Getter (kein setter) sichtbar gemacht
11   public final int getValue() {
12     return value.get();
13   }
14
15   // intern kann der Wert der Property wie gewohnt veraendert werden
16   public void count() {
17     this.value.set(this.getValue() + 1);
18   }
19
20   public void reset() {
21     this.value.set(0);
22   }
23
24   // Nach aussen wird nur die ReadOnlyProperty zur Verfuegung gestellt
25   // Diese stellt nur Getter,keine Setter zur Verfügung
26   public final ReadOnlyIntegerProperty valueProperty() {
27     return value.getReadOnlyProperty();
28   }
29 }
```

Listing 8.7: Counter mit ReadOnly-Property

Die wichtigsten Read-Only Properties werden in obiger Tabelle zusammengefasst:

Konkrete Klasse	ReadOnlyProperty	gewrappter Typ
`ReadOnlyBooleanWrapper`	`ReadOnlyBooleanProperty`	`boolean`
`ReadOnlyIntegerWrapper`	`ReadOnlyIntegerProperty`	`int`
`ReadOnlyDoubleWrapper`	`ReadOnlyDoubleProperty`	`double`
`ReadOnlyStringWrapper`	`ReadOnlyStringProperty`	`String`
`ReadOnlyObjectWrapper<T>`	`ReadOnlyObjectProperty<T>`	`T` (generischer Typ)

8.4 Binding

Sehr oft werden Properties und ihre Listener dazu verwendet, den Wert der Property eines Objektes mit dem Werte einer Property eines anderen Objekts zu synchronisieren. Dies kann in der Regel einfacher mit Hilfe von Binding erfolgen. Unter Binding versteht man eine Technik, mit der man den Wert einer JavaFX-Property `p1` als abhängig vom Wert einer anderen Property `p2` definieren kann. Ändert nun die Property `p2` ihren Wert,so ändert sich automatisch auch der Wert von der abhängigen Property `p1`. Binding ist ein mächtiges Werkzeug, mit dem Werte von Properties ohne der Implementierung von Listener synchronisiert werden können. Das folgende Beispiel demonstriert diese Technik:

Beispiel zu unidirektionalem Binding

```java
import javafx.scene.shape.Circle;

public class BindingDemo_1 {

    private static final Circle c1 = new Circle(4.0);
    private static final Circle c2 = new Circle(6.0);

    public static void main(String[] args) {

        System.out.println(c1);
        System.out.println(c2);

        // Binde Radius von c1 an den Radius von c2
        // Der Radius von c1 wird damit abhaengig vomRadius von c2
        c1.radiusProperty().bind(c2.radiusProperty());
        System.out.println("------------------");
        System.out.println(c1);
        System.out.println(c2);

        checkBinding();

        c2.setRadius(15.0);
        // Es wird auch der radius von c1 veraendert
        System.out.println("------------------");
        System.out.println(c1);
        System.out.println(c2);

        try {
            c1.setRadius(17.0);
        } catch (RuntimeException ex) {
            System.out.println("Fehler: " + ex.getMessage());
        }

        // Bindung aufheben
        c1.radiusProperty().unbind();

        checkBinding();
    }

    private static void checkBinding() {
        if (c1.radiusProperty().isBound()) {
            System.out.println("Radius von c1 gebunden");
        } else {
            System.out.println("Radius von c1 nicht gebunden");
        }
    }
}
```

Listing 8.8: Unidirektionales Binding

Ausgabe:

```
Circle[centerX=0.0, centerY=0.0, radius=4.0, fill=0x000000ff]
Circle[centerX=0.0, centerY=0.0, radius=6.0, fill=0x000000ff]
------------------
Circle[centerX=0.0, centerY=0.0, radius=6.0, fill=0x000000ff]
Circle[centerX=0.0, centerY=0.0, radius=6.0, fill=0x000000ff]
Radius von c1 gebunden
------------------
Circle[centerX=0.0, centerY=0.0, radius=15.0, fill=0x000000ff]
Circle[centerX=0.0, centerY=0.0, radius=15.0, fill=0x000000ff]
```

```
Fehler: Circle.radius : A bound value cannot be set.
Radius von c1 nicht gebunden
```

Das obige Beispiel demonstriert einfaches unidirektionales Binding. Dabei wird die `radiusProperty` vom Circle `c1` an die `radiusProperty` von `c2` gebunden. Der Radius von `c2` ist damit die unabhängige Property und der Radius von `c1` die abhängige Property. Verändert sich der Wert des Radius von `c2` so wird automatisch der Radius von `c1` synchronisiert. Diese Synchronisation geschieht auch sofort bei der Bindung. Während der Bindung kann der Wert der abhängigen Property (also der Radius von `c1`) nicht direkt verändert werden. Der versuchte Aufruf von `c1.setRadius()` in Zeile 29 resultiert in einem Laufzeitfehler.

8.4.1 Unidirektionales Binding

Wie das obige Beispiel demonstriert, wird bei unidirektionalem Binding der Wert einer unabhängigen Property an den Wert einer abhängigen Property gebunden. Verändert sich der Wert der unabhängigen Property, so ändert sich automatisch auch der Wert der abhängigen Property. Beim unidirektionalen Bindung kommen die folgenden Methoden der Interfaces `javafx.beans.property.Property<T>` zum Einsatz:

`void bind(ObservableValue<? extends T> observable)`
Bindet diese Property an die unabhängige Property `observable`.

`boolean isBound()`
Liefert `true`, wenn diese Property an eine andere gebunden ist (also eine abhängige Property ist), sonst `false`.

`void unbind()`
Löst ein unidierktionales Binding dieser Property.

8.4.2 Bidirektionales Binding

Bidirektionales Binding erlaubt es, zwei JavaFX-Properties in beide Richtungen zu synchronisieren. Wann immer sich der Wert einer Property ändert, wird der Wert der anderen Property synchronisiert. Bei bidierktionalemBinding können die Werte beider beteiligten Properties mit Hilfe von Settern verändert werden.
Bidirektionales Binding wird in erster Linie dazu verwendet,GUI-Komponenten mit den Properties von Geschäftsklassen (Modelldaten) zu synchronisieren. Wenn sich Daten im Modell ändern, wird automatisch die Oberfläche aktualisiert und umgekehrt.

Beispiel zu bidirektionalem Binding

```
1  import javafx.scene.shape.Circle;
2
3  public class BindingDemo_2 {
4
5      private static final Circle c1 = new Circle(4.0);
6      private static final Circle c2 = new Circle(6.0);
7
8      public static void main(String[] args) {
9
10         System.out.println(c1);
11         System.out.println(c2);
12
13         // Binde Radius von c1 an den Radius von c2
14         // Der Radius von c2 wird damit abhaengig vomRadius von c1
15         c1.radiusProperty().bindBidirectional(c2.radiusProperty());
16         System.out.println("-------------------");
17         System.out.println(c1);
18         System.out.println(c2);
```

```
19
20          c2.setRadius(15.0);
21          // Es wird auch der radius von c1 veraendert
22          System.out.println("------------------");
23          System.out.println(c1);
24          System.out.println(c2);
25
26          c1.setRadius(17.0);
27          // Es wird auch der radius von c2 veraendert
28          System.out.println("------------------");
29          System.out.println(c1);
30          System.out.println(c2);
31      }
32  }
```

Listing 8.9: Bidirektionales Binding

Ausgabe:

```
Circle[centerX=0.0, centerY=0.0, radius=4.0, fill=0x000000ff]
Circle[centerX=0.0, centerY=0.0, radius=6.0, fill=0x000000ff]
------------------
Circle[centerX=0.0, centerY=0.0, radius=6.0, fill=0x000000ff]
Circle[centerX=0.0, centerY=0.0, radius=6.0, fill=0x000000ff]
------------------
Circle[centerX=0.0, centerY=0.0, radius=15.0, fill=0x000000ff]
Circle[centerX=0.0, centerY=0.0, radius=15.0, fill=0x000000ff]
------------------
Circle[centerX=0.0, centerY=0.0, radius=17.0, fill=0x000000ff]
Circle[centerX=0.0, centerY=0.0, radius=17.0, fill=0x000000ff]
```

Beim bidirektionalen Bindung kommen die folgenden Methoden der Interfaces `javafx.beans.property.Property<T>` zum Einsatz:

`void bindBidirectional(Property<T> other)`
Erzeugt ein bidirektionales Binding zwischen dieser Property und der Property `other`. Beim Binding wird der Wert dieser Property mit dem Wert von `other` aktualisiert. Bidirektionales Binding ist unabhängig von unidirektionalem Binding. So kann eine bidirektional gebundene Property gleichzeitig unidirektional an eine andere Property gebunden sein. Außerdem ist es möglich, dass eine Property gleichzeitig mehrere bidirektionale Bindings besitzt.

`void unbindBidirectional(Property<T> other)`
Löst die bidirektionaleBindung dieser Property von Bindung mit `other`. Ist diese Property nicht mit `other` bidirektional gebunden, so hat der Aufruf der Methode keine Wirkung.

8.4.3 Fluent Binding

Unidirektionales und bidirektional Binding sind hervorragend geeignet, um Properties mit gleichem Datentyp und gleichem Wert aneinander zu binden. Oft ist es aber notwendig, dass für die abhängige Property der Wert der unabhängigen Property in bestimmter Weise manipuliert wird (Umrechnung des Werts, anderer Datentyp). JavaFX stellt das Fluent-API für solche Situationen zur Verfügung. Das folgende Beispiel demonstriert die prinzipelle Arbeitsweise.

Beispiel zumFluent-API

Zunächst wird eine Klasse **Temperature** definiert, die mit Hilfe einer DoubleProperty einen Celsiuswert kapselt.

```
1  import javafx.beans.property.DoubleProperty;
2  import javafx.beans.property.SimpleDoubleProperty;
3
```

```
4  public class Temperature {
5
6      private final DoubleProperty celsius = new SimpleDoubleProperty(this, "celsius"
           ,0.0);
7
8      // Konstruktoren
9
10     public Temperature() {
11         this(0.0);
12     }
13
14     public Temperature(double celsius) {
15         this.celsius.set(celsius);
16     }
17
18     public void setCelsius(double celsius) {
19         this.celsius.set(celsius);
20     }
21
22     public double getCelsius() {
23         return celsius.get();
24     }
25
26
27     public final DoubleProperty celsiusProperty() {
28         return celsius;
29     }
30 }
```

Listing 8.10: Klasse Temperature.java

In der folgenden Applikation wird der Celsiuswert einer Temperaturinstanz an die Textproperty eines Labels gebunden, wobei mit Hilfe des Fluent-API der Celsiuswert in Fahrenheit umgerechnet und dann in einen formatierten String verwandelt wird:

```
1  import javafx.scene.control.Label;
2  import javafx.application.Application;
3  import javafx.stage.Stage;
4
5  public class FluentBindingDemo extends Application {
6
7      public static void main(String[] args) {
8          launch(args);
9      }
10
11     @Override
12     public void start(Stage primaryStage) throws Exception {
13         Temperature t =new Temperature(20.0);
14
15         Label lbl = new Label();
16
17         // Fahrenheit 9 / 5 * C + 32
18         lbl.textProperty().bind(t.celsiusProperty()
19                         .multiply(9.0)
20                         .divide(5.0)
21                         .add(32.0)
22                         .asString("%.2f F"));
23
24         System.out.println(lbl.getText());
25         t.setCelsius(40);
26         System.out.println(lbl.getText());
27         System.exit(0);
28     }
29 }
```

Listing 8.11: FluentAPIDemo.java

Ausgabe

```
68.00 F
104.00 F
```

In obigem Beispiel wurden folgende Methoden der Klasse `DoubleProperty` verwendet:

```
public DoubleBinding add(double other)
public DoubleBinding multiply(double other)
public DoubleBinding divide(double other)
public StringBinding asString(String format)
```

Alle numerischen Properties stellen einen Satz solcher (mehrfach überladenen) Methoden zur Verfügung:

- `add()`, `subtract()`, `multiply()`, `divide()`, `negate()` für numerische Berechnungen.
- `asString()` zur Verwandlung in eine StringProperty
- `greaterThan()`, `lessThan()` und viele andere Vergleichsfunktion zur Verwandlung in eine BooleanProperty

Die oben angegebenen Methoden sind mehrfach überladen, so gibt es z.B. die multiply()-Methode in folgenden Versionen:

```
public DoubleBinding multiply(double other)
public DoubleBinding multiply(float other)
public DoubleBinding multiply(long other)
public DoubleBinding multiply(ObservableNumberValue other)
```

Mit der letzten methoden kann nicht nur mit Konstanten gerechnet werden, sondern es können auch die Werte mehrere Properties miteinander kombiniert werden:

```
1 Rectangle r = new Rectangle(30.0, 20.0);
2 Label lbl = new Label();
3 lbl.textProperty().bind(r.widthProperty().multiply(r.heightProperty()).asString());
```

In obigem Codeabschnitt wird die Fläche des Rechtecks `r` in die TextProperty des Labels `lbl` gebunden. Nähere Auskunft gibt die API: z.B. DoubleProperty

8.4.4 Custom Binding

Wenn man mit den Möglichkeiten des Binding-API bzw. des Fluent-API nicht das Auslangen findet, kann man auch ein Custom-Binding-Objekt erzeugen. Mit Custom-Binding legt man zwei Eigenschaften fest:

- Die Abhängigkeiten der JavaFX-Properties.
- Die Berechnung des neuen Wertes.

Die Vorteile des CustomBindinge sind, dass man beliebig viele unabhängige Properties verwenden kann und bei der Berechnung des abhängigen Wertes nicht nur auf die Berechnungsmethoden des Fluent-API beschränkt ist.
Das folgende Beispiel demonstriert die prinzipielle Arbeitsweise. Mit Hilfe eines Custom-Bindings werden die Fläche und der Umfang eines Rechtecks ermittelt und als String retourniert.

```
1  import javafx.application.Application;
2  import javafx.beans.binding.StringBinding;
3  import javafx.scene.control.Label;
4  import javafx.scene.shape.Rectangle;
5  import javafx.stage.Stage;
6
7  public class CustomBindingDemo_1 extends Application {
8
9    public static void main(String[] args) {
10     launch(args);
11   }
12
13   @Override
14   public void start(Stage primaryStage) throws Exception {
15
16     final Rectangle r = new Rectangle(200, 100);
17     final Label lbl = new Label();
18
19     StringBinding rectInfoBinding = new StringBinding() {
20
21       // Instanzinitialisierer
22       // Legt die unabhaengigen Properties fest
23       {
24         super.bind(r.widthProperty(), r.heightProperty());
25       }
26
27       // Berechnet den erwuenschten abhaengigen Wert, hier einen String
28       @Override
29       protected String computeValue() {
30         double a = r.getWidth() * r.getHeight();
31         double u = 2.0 * (r.getWidth() + r.getHeight());
32
33         return String.format("Flaeche: %.2f E^2, Umfang: %.2f E", a, u);
34       }
35     };
36
37     lbl.textProperty().bind(rectInfoBinding);
38
39     System.out.println(lbl.getText());
40     r.setWidth(300);
41     System.out.println(lbl.getText());
42     System.exit(0);
43   }
44 }
```

Listing 8.12: CustomBindingDemo1.java

Ausgabe:

```
Fläche: 20000.00 E^2, Umfang: 600.00 E
Fläche: 30000.00 E^2, Umfang: 800.00 E
```

In obigem Beispiel wird eine anonyme innere Klasse erzeugt, die von der abstrakten Klasse `javafx.beans.binding.StringBinding` erbt. Dabei muss die abstrakte Methode

```
protected abstract String computeValue()
```

überschrieben werden, die den abhängigen Wert (hier einen String liefert). Von welchen unabhängigen Properties dieser berechnete Wert anhängig ist, wird durch Aufruf der Methode

```
protected final void bind(Observable... dependencies)
```

festgelegt. Dies geschieht in der Regel beim Erzeugen des Objekts, also in einem Instanzinitialisierer(Zeilen 23-25). Hier ist der errechnete String von den beiden Properties `widthProperty` und `heightProperty` der Rectangles `r` abhängig.

Zeile 37 zeigt, wie der Wert dieses StringBindings an die Textproperty des Labels `l` gebunden wird. Das Paket `javafx.beans.binding` stellt für CustomBinding die folgenden abstrakten Klassen zur Verfügung, die alle nach dem oben beschriebenem Prinzip arbeiten:

`BooleanBinding`, `DoubleBinding`, `FloatBinding`, `IntegerBinding`, `StringBinding`, `LongBinding`, `ObjectBinding<T>`

Im Wesentlichen unterscheiden sich diese abstrakten Klassen durch den Rückgabetyp der abstrakten Methode `computeValue()`:

```
BooleanBinding:
protected abstract boolean computeValue()

DoubleBinding:
protected abstract double computeValue()

FloatBinding:
protected abstract float computeValue()

IntegerBinding:
protected abstract int computeValue()

LongBinding:
protected abstract double computeValue()

StringBinding:
protected abstract String computeValue()

ObjectBinding<T>:
protected abstract T computeValue()
```

Beispiel

Das folgende Beispiel rendert eine Scene, in der eine Rectangle-Instanz in einer StackPane gerendert wird. Dabei werden die Breite und die Höhe des Rechtecks mit Hilfe von Fluent-Binding an die halbe Breite bzw. Höhe der Scene gebunden und die Füllfarbe des Rechtecks wird mit Hilfe von CustomBinding über den Umfang des Rechtecks berechnet.

```
1  import javafx.application.Application;
2  import javafx.beans.binding.ObjectBinding;
3  import javafx.scene.Scene;
4  import javafx.scene.layout.StackPane;
5  import javafx.scene.paint.Color;
6  import javafx.scene.paint.Paint;
7  import javafx.scene.shape.Rectangle;
8  import javafx.stage.Stage;
9
10 public class BindingDemo_3 extends Application {
11   public static void main(String[] args) {
12     launch(args);
13   }
14
15   @Override
16   public void start(Stage primaryStage) throws Exception {
17
18     Rectangle r = new Rectangle(100, 50);
19     r.setArcHeight(30.0);
20     r.setArcWidth(30.0);
21
22     StackPane root = new StackPane(r);
```

```
23
24      Scene scene = new Scene(root, 400, 200);
25      primaryStage.setTitle("Binding Demo");
26
27      r.widthProperty().bind(scene.widthProperty().divide(2.0));
28      r.heightProperty().bind(scene.heightProperty().divide(2.0));
29
30      ObjectBinding<Paint> fillBinding = new ObjectBinding<Paint>() {
31
32        {
33          super.bind(scene.widthProperty(), scene.heightProperty());
34        }
35
36        @Override
37        protected Paint computeValue() {
38          double d = scene.getWidth() + scene.getHeight();
39          if(d >= 0.0 && d < 500.0) {
40            return Color.RED;
41          } else if(d < 1000.0) {
42            return Color.BLUE;
43          } else if(d < 1500.0) {
44            return Color.GREEN;
45          } else {
46            return Color.ORANGE;
47          }
48        }
49      };
50
51      r.fillProperty().bind(fillBinding);
52
53      primaryStage.setScene(scene);
54      primaryStage.show();
55
56    }
57  }
```

Listing 8.13: BindingDemo.java

Ausgabe:

Abbildung 8.2: Binding_Demo_3

Eine weitere Möglichkeit, CustomBinding zu realisieren steht mit Hilfe der statischen Methoden `createXXXBinding()` der Klasse `javafx.beans.binding.Bindings` zur Verfügung. Exemplarisch wird eine dieser Methoden im Folgenden vorgestellt:

```
public static BooleanBinding createBooleanBinding(Callable<Boolean> func,
                                                   Observable... dependencies)
```

Diese Methode erlaubt es, ein `BooleanBinding` zu realisieren. Dabei stellt die Methode `Boolean call()` des Interfaces `Callable` die Berechnungsmethode dar (diese kann als Lambdaexpression realisiert werden) und im Varargsparameter `dependencies` werden alle zur Berechnung des Ergebniswertes notwendigen abhängigen Properties übergeben.

Beispiel

Der folgende Codeabschnitt setzt die Selected-Property auf einer Checkbox `cb` auf `true`, wenn der Wert der IntegerProperty `i1` kleiner als der Wert der Integerproperty `i2` des des Objektes `o` ist:

```
1 cb.selectedProperty().bind(Bindings.createBooleanBinding(
2   () -> o.getI1() < o.getI2(),
3   o.i1Property(), o.i2Property()));
```

Listing 8.14: CustomBinding2.java

Nach diesem Muster stellt die Klasse `Bindings` die folgenden Methoden zur bequemen Realisition eines CustomBindings bereit:

```
public static BooleanBinding createBooleanBinding(Callable<Boolean> func,
                                                   Observable... dependencies)
public static DoubleBinding createDoubleBinding(Callable<Double> func,
                                                 Observable... dependencies)
public static FloatBinding createFloatBinding(Callable<Float> func,
                                               Observable... dependencies)
public static IntegerBinding createIntegerBinding(Callable<Integer> func,
                                                   Observable... dependencies)
public static LongBinding createLongBinding(Callable<Long> func,
                                             Observable... dependencies)
public static StringBinding createStringBinding(Callable<String> func,
                                                 Observable... dependencies)
public static <T> ObjectBinding<T> createObjectBinding(Callable<T> func,
                                                       Observable... dependencies)
```

8.4.5 Bidirektionales Binding unterschiedlicher Datentypen

Möchte man eine Property vom Datentyp `T` bidirektional an eine Stringproperty binden, so kann dies z.B. mit Hilfe eines Konverters geschehen. Die Klasse `StringProperty` stellt dazu eine Methode

```
public <T> void bindBidirectional(Property<T> other, StringConverter<T> converter)
```

zur Verfügung, mit deren Hilfe der in der Property `other` gekapselte Wert vom Typ `T` in einen String und umgekehrt verwandelt werden kann. Alternativ kann zu diesem Zweck auch folgende statische Methode der Klasse `Bindings` herangezogen werden:

```
public static <T> void bindBidirectional(Property<String> stringProperty,
                                         Property<T> otherProperty,
                                         StringConverter<T> converter)
```

Zu beachten ist dabei, dass in diesen beiden Methoden wenigstens einer der beiden übergebenen Parameter den Typstellvertreter `T` mit einem konkreten Typ versorgen muss.

`StringConverter<T>` ist dabei eine abstrakte Klasse folgender Bauart:

```
public abstract class StringConverter<T> extends Object {
  public abstract String toString(T object);
  public abstract T fromString(String string);
}
```

Konkerte Implementierungen dieser abstrakten Klasse müssen also in der Lage sein Objekte vom Typ `T` in Strings (Methode `toString()`) und umgekehrt (Methode `fromString()`) zu verwandeln.

Beispiel

Das folgende Beispiel verwendet wieder die in einem obigen Beispiel vorgestellte Klasse `Temperature`. Diese Klasse kapselt mit Hilfe einer `DoubleProperty` einen Temperatur in Grad Celius. In der folgenden Anwendung wird mit Hilfe dieser Klasse ein Temepraturkonverter von Celsius in Fahrenheit und umgekehrt realisiert.

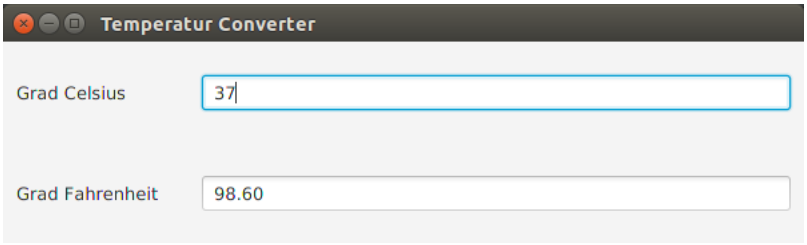


Abbildung 8.3: TemperaturConverter 01

Im Folgenden wird der Controller dieser Anwendung vorgestellt:

```
1  import java.net.URL;
2  import java.util.Locale;
3  import java.util.ResourceBundle;
4  import javafx.fxml.FXML;
5  import javafx.fxml.Initializable;
6  import javafx.scene.control.TextField;
7  import javafx.util.StringConverter;
8
9  public class TempConverterFXMLController implements Initializable {
10
11   @FXML
12   private TextField txtFahrenheit;
13
14   @FXML
15   private TextField txtCelsius;
16
17   private Temperature temp;
18
19   @Override
20   public void initialize(URL url, ResourceBundle rb) {
21     temp = new Temperature(20.0);
22
23     txtCelsius.textProperty().bindBidirectional(temp.celsiusProperty(), new
         StringConverter<Number> () {
24       @Override
25       public String toString(Number object) {
26         return String.format(Locale.US, "%.2f", object.doubleValue());
27       }
28
29       @Override
30       public Double fromString(String string) {
31         txtCelsius.setStyle(null);
32         try {
33           return Double.parseDouble(string);
34         } catch(NumberFormatException ex) {
35           txtCelsius.setStyle("-fx-border-color: red");
36           return null;
37         }
38       }
39     });
40
41     txtFahrenheit.textProperty().bindBidirectional(temp.celsiusProperty(), new
         StringConverter<Number> () {
42       @Override
```

```
43        public String toString(Number object) {
44          return String.format(Locale.US, "%.2f", object.doubleValue() * 9.0 / 5.0 +
                32.0);
45        }
46
47        @Override
48        public Double fromString(String string) {
49          txtFahrenheit.setStyle(null);
50          try {
51            double f = Double.parseDouble(string);
52            return (f - 32.0) * 5.0 / 9.0;
53          } catch(NumberFormatException ex) {
54            txtFahrenheit.setStyle("-fx-border-color: red");
55            return null;
56          }
57        }
58      });
59    }
60  }
```

Listing 8.15: Verwendung von Konvertern

Wie obiges Beispiel zeigt, kann mit Hilfe der Konverter auch eine Fehlerbehandlung implementiert werden. Die Methode `toString()` ist in der Regel unproblematisch und hier wird nur der übergebene Doublewert bei Bedarf umgerechnet und in ein entsprechendes Ausgabeformat konvertiert. In der umgekehrten Richtung, also in der Methode `fromString()`, wird versucht, den (vom Benutzer eingegebenen) String in einen Doublewert zu parsen und diesen Wert wenn nötig umzurechnen. Sollte die Umwandlung fehlschlagen, so wird der Rahem des entsprechenden Textfeldes auf rot gesetzt und `null` retourniert. Zu Beginn dieser Methode wird der Stil des Textfeldes mit `setStyle(null)` auf den in der FXML-Datei angegebenen Stil zurückgesetzt.

Die folgende Abbildung zeigt den Zustand der Oberfläche nach einer falschen Eingabe:

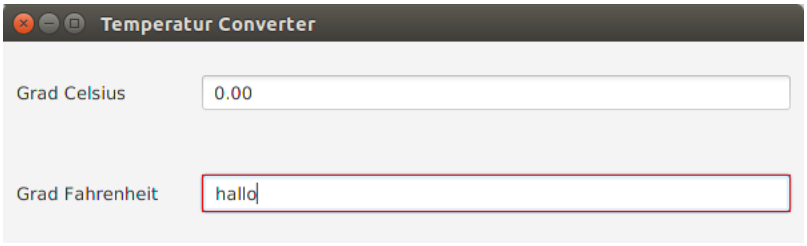


Abbildung 8.4: TemperaturConverter 02

Zu beachten ist auch, dass bei Verwendung der generischen Konverters für einen Wrappertyp (`Integer`, `Double`, `Long`, ...) der generische Typ `Number` sein muss.
Im Paket `javafx.util.converter` gibt es eine Reihe nicht abstrakter Konverter für verschiedene Datentypen, die statt dem generischen Konverter verwendet werden können. Einge dieser Konverter sind:

```
IntegerStringConverter
DoubleStringConverter
LocalTimeStringConverter
...
```

Mit Hilfe eines `DoubleStringConverters` hätte z.B. die Konvertierung für den Celsiuswert wie folgt abgeändert werden können (zu diesem Zweck ist aber gleichzeitig die `DoubleProperty` in der Klasse `Temperature` auf eine `ObjectProperty<Double>` abzuändern, da in den `bind()`-Methoden zumindest einer der beiden Parameter den generischen Typ `T` versorgen muss):

```
// Die celsiusProperty in Temperature hat nun den Typ ObjectProperty<Double>
txtCelsius.textProperty().bindBidirectional(temp.celsiusProperty(), new
    DoubleStringConverter () {

  // es müssen nicht mehr beide Methoden überschrieben werden

  @Override
  public Double fromString(String string) {
  txtCelsius.setStyle(null);
    try {
      return Double.parseDouble(string);
    } catch(NumberFormatException ex) {
      txtCelsius.setStyle("-fx-border-color: red");
      return 0.0;     // return null erzeugt nun eine NullpointerException
    }
  }
});
```

Listing 8.16: Verwendung eines DoubleStringConverters

8.5 Observable Collections

In den obigen Abschnitten wurde erklärt, wie an FX-Properties beobachten kann und auf deren Veränderung mit Hilfe von Listenern oder Binding reagieren kann. FX stellt aber auch Observable Collections zur Verfügung, die sich nach dem ObserverPattern beobachten lassen, ob Elemente hinzugefügt, entfernt oder aktualisiert wurden. Man reagiert auf solche Veränderungen mit Hilfe von InvalidationListener oder ChangeListener.
Dabei verwenden die Observable Collections die `equals()` - Methode der gespeicherten Elemente. Wenn man also ein Element `o1` durch ein anderes Element `o2` ersetzt und die Methode `o1.equals(o2)` liefert `true` werden keine Listenermethoden aufgerufen.
Zum Erzeugen einer Observable Collection verwendet man statische Methoden der Klasse `javafx.collections.`
Wird eine Observable Collection einer ListView oder TableView als Datenquelle zugeordnet, so verständigt die Observable Collection bei Änderungen die View, welche sich bei Bedarf neu darstellen kann.

Beispiel

Das folgende Beispiel demonstriert, wie man mit Hilfe eines `ListChangeListener` auf Änderungen in einer ObservableList reagieren kann:

```
import javafx.collections.FXCollections;
import javafx.collections.ListChangeListener;
import javafx.collections.ObservableList;

public class ObservableListDemo {

  public static void main(String[] args) {
    final ObservableList<Integer> list = FXCollections.observableArrayList();

    list.addListener(new ListChangeListener<Integer>() {
      @Override
      public void onChanged(ListChangeListener.Change<? extends Integer> c) {
        c.next();
        if(c.wasAdded()) {
          System.out.println("Elemente hinzugefügt: " + c.getAddedSubList());
          list.sort(null);
          System.out.println(list);
        } else if(c.wasPermutated()) {
          System.out.println("Elemente umgeordnet");
        } else if(c.wasRemoved()) {
          System.out.println("Elemente entfernt: " + c.getRemoved());
```

```
22            System.out.println(list);
23          }
24        }
25      });
26
27      list.addAll(3,6,1,2);
28      list.add(3);
29      list.removeAll(3);
30    }
31  }
```

Listing 8.17: ObservableListDemo.java

Das obige Programm erzeugt die folgende Ausgabe:

```
Elemente hinzugefügt: [3, 6, 1, 2]
Elemente umgeordnet
[1, 2, 3, 6]
Elemente hinzugefügt: [3]
Elemente umgeordnet
[1, 2, 3, 3, 6]
Elemente entfernt: [3, 3]
[1, 2, 6]
```

Abbildungsverzeichnis

Listings