Informatik

2

Skript

Prof. Dr.-Ing. Holger Vogelsang

Sommersemester 2014

Don’t panic

Inhaltsverzeichnis

1

Einleitung

Dieses Dokument besteht aus mehreren Kapiteln.

1. Einleitungskapitel: Es handelt sich um dieses Kapitel.
2. Kurzeinführung in Java: Hier werden die für die Übungen sowie die Vorlesung wichtigsten Eigenschaften von Java vorgestellt. Zu beachten ist, dass es sich hierbei nicht um ein komplettes Lehrbuch handelt. Die Idee besteht darin, die in der Vorlesung „Informatik 2“ vermittelten Java-Grundlagen kompakt darzustellen.
3. Fortgeschrittene Java-Themen: Dieses Kapitel stellt interessierten Studenten weitere Eigenschaften von Java vor, ohne dass diese jedoch in der Vorlesung behandelt und benötigt werden.
4. Ein- und Ausgabebehandlung: Hier wird beschrieben, wie Daten aus externen Quellen wie Dateien gelesen und in diese geschrieben werden. Dabei beschränkt sich das Kapitel auf die sehr einfachen Klassen.

Am Ende des Dokumentes befinden sich noch ein Abbildungs-, Literatur- sowie ein Stichwortverzeichnis.

2

Kurzeinführung in Java

2.1 Übersicht

Java ist eine objektorientierte Programmiersprache, die komplett auf prozedurale Elemente verzichtet. Java besitzt eine sehr umfangreiche Klassenbibliothek.

1990 hat James Gosling die Arbeiten an einem Vorgänger von Java (Oak) aufgenommen. Das Ziel war eine Sprache zur Steuerung von Geräten. Im Januar 1996 wurde die erste Java-Version 1.0 für die Entwickler freigegeben. Java hat folgende Eigenschaften:

- **Interpretiert und compiliert:** Der Quelltext muss mit Hilfe eines Compilers in einen plattformunabhängigen Zwischencode übersetzt werden. Dieser wird in einer virtuellen Maschine (JVM) interpretiert. Darin eingebaut ist ein sogenannter Hotspot-Compiler, der kritische Programmteile während der Programmausführung in plattformabhängigen Code übersetzt.
- **Objektorientierter Ansatz**: Alle Elemente außer primitiven Datentypen sind Objekte. Sogar Klassen sind wiederum Objekte.
- **Streng typisiert**: Java arbeitet mit Datentypen, die der Entwickler relativ eingeschränkt in andere Typen konvertieren kann. Die Umwandlung wird sowohl zur Übersetzungs- als auch Laufzeit überwacht.
- **Statisch und dynamisch gebunden**: Methodenaufrufe werden sowohl zum Zeitpunkt des Übersetzens als auch zur Laufzeit aufgelöst.
- **Sicher**: Java besitzt mehrere Sicherheitsmerkmale, die auch zur Laufzeit greifen:
  1. Java kennt keine direkten Speicher- und Hardwarezugriffe. Diese sind nur über den Umweg einer Anbindung an C-Code mittels des sogenannten JNI (Java Native Interface) erlaubt.
  2. Nach dem Laden einzelner Klassendateien werden diese zur Laufzeit überprüft, um sicherzustellen, dass die virtuelle Maschine nur gültigen Bytecode ausführen wird.

3. Security-Manager überprüfen den Programmzugriff auf die Umgebung. So kann beispielsweise verhindert werden, dass ein Applet auf die lokalen Ressourcen eines Anwenders zugreift. Die gezielte Vergabe von Rechten durch den Anwender erlaubt eine Aufweichung.

Die vom Compiler erzeugten binären Objektdateien lassen sich zur Weitergabe zu JAR-Archiven zusammenfassen. Wird ein Java-Programm compiliert oder ausgeführt, müssen die abhängigen Klassen im sogenannten Klassenpfad zu finden sein. Der Klassenpfad kann auf mehreren Wegen der virtuellen Maschine übergeben werden:

- durch einen Kommandozeilenparameter der Form: `-classpath cp1;cp2`. `cp1` und `cp2` sind hier Synonyme für die Verzeichnisnamen oder Pfade zu JAR-Archiven. Unter UNIX muss statt des unter Windows üblichen Pfadtrennzeichens `;` ein Doppelpunkt verwendet werden.
- durch Setzen der Umgebungsvariablen `CLASSPATH`;

Dieses Java-Kapitel stellt auch die Spracheigenschaften vor, die mit Java 6 eingeführt wurden. Unterschiede zu älteren Versionen sind auf Grund der schon relativ langen Verfügbarkeit der neuen Version nicht mehr explizit hervorgehoben.

2.2 Organisation der Sprache

Vergabe von Datei-Endungen:

- `.java`: Quelltext.
- `.class`: vom Compiler erzeugte Objektdateien mit Zwischencode.
- `.jar`: Archive mit mehreren Objektdateien und anderen Ressourcen.

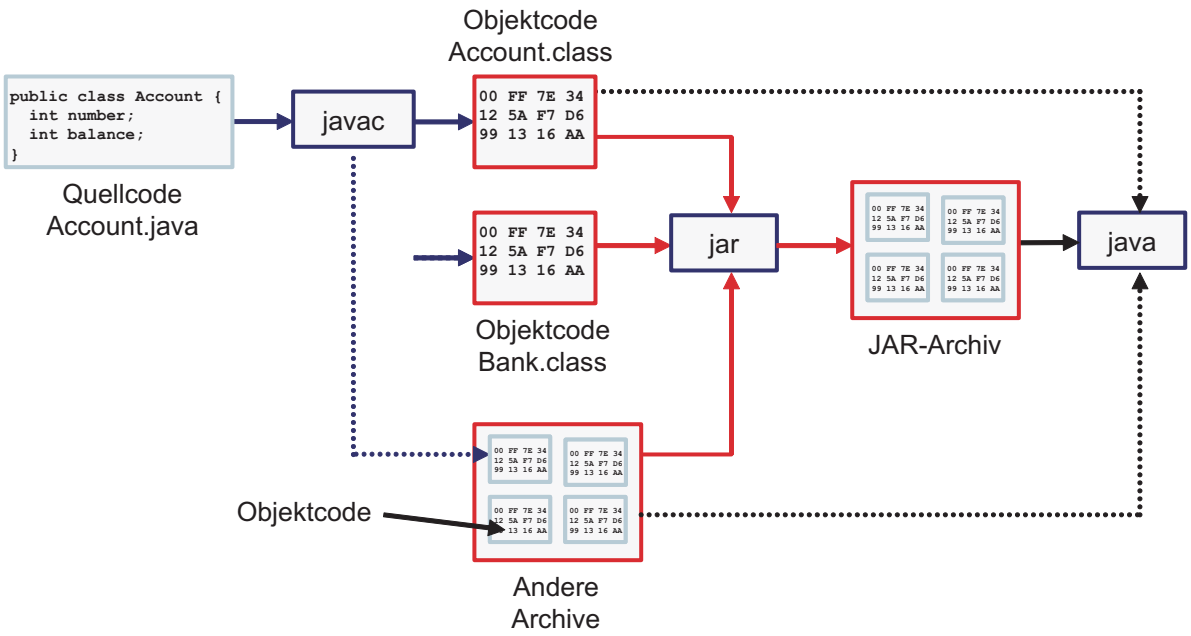

Abbildung 2.1: Ablauf der Programmerstellung in Java

2.2.1 Modularisierung des Quelltextes durch Pakete

Java bietet Klassen (Abschnitt 2.5) und Pakete zur Modularisierung des Quelltextes.

Mit **Paketen** lassen sich logisch zusammengehörige Klassen zu einer Einheit zusammenfassen. So können Klassen, die sonst auf Grund von Namenskollisionen nicht kombinierbar wären, zusammen eingesetzt werden.

Eigenschaften und Struktur:

- Pakete lassen sich hierarchisch zu einer Baumstruktur aufbauen. Die Pakethierarchie wird auf Dateiebene auf eine ebenso hierarchische Verzeichnisstruktur abgebildet.
- Klassen, die nicht in einem Paket liegen, werden automatisch einem Standardpaket zugeordnet. Auf diese Klassen kann aus anderen Paketen nicht zugegriffen werden. Standardpakete sollten daher vermieden werden.

Definition von Paketen

Alle Klassen einer Quelltextdatei werden durch die Anweisung `package` einem Paket zugeordnet. Diese Anweisung muss die erste in der Datei sein. Wenn Pakete hierarchisch geschachtelt werden, bleiben unter- und übergeordnete Pakete trotzdem logisch unabhängig voneinander. Damit ist gemeint, dass der Inhalt eines Unterpaketes nicht Inhalt des übergeordneten Paketes ist.

Beispiel:

```
package de.companyXY.math;
public class Fraction {
  /* ... */
}
```

Die Klasse `Fraction` wird auf das Verzeichnis `./de/companyXY/math` relativ zum Projektverzeichnis abgebildet.

Zugriff auf Paketinhalte

Auf Klassen im Paket wird aus anderen Paketen mit dem Operator `.` zugegriffen.

Beispiel:

```
de.companyXY.math.Fraction fr;
```

Import von Paketinhalten

Die Angabe eines kompletten Namens, bestehend aus Paket und Klasse, ist unhandlich. Daher kann mit `import` der Inhalt eines Paketes direkt angesprochen werden.

Beispiel zum Einbinden aller Klassen des Paketes:

```
import de.companyXY.math.*;
/* ... */
Fraction fr;
```

Diese Variante wird von Oracle nicht empfohlen. Besser ist es, nur die benötigten Klassen einzubinden.

Beispiel (Einbinden nur der Klasse `Fraction`*):*

```
import de.companyXY.math.Fraction;
/* ... */
Fraction fr;
```

Statisches Importieren aus einer Klasse

Diese Art des Imports hat aber nur wenig Bezug zu Paketen. Die Beschreibung ist daher erst in Abschnitt 2.5.6 über statische Methoden zu finden.

Aufbau von Paketnamen

Ein sogenannter „vollqualifizierten“ Methodenname hat den folgenden Aufbau, wobei durchaus Variationsmöglichkeiten bestehen:
Organisation.Paketname.Klassenname.Methodenname
Beispiele:

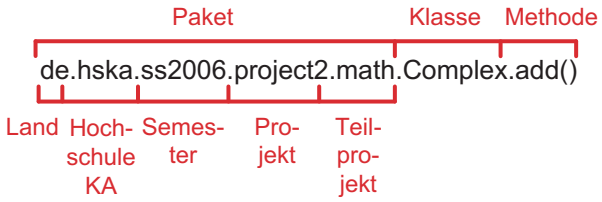

Abbildung 2.2: Vollqualifizierter Name für ein deutsches Projekt

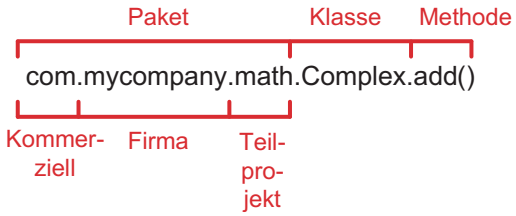

Abbildung 2.3: Vollqualifizierter Name für ein kommerzielles Projekt

Abbildung 2.4: Vollqualifizierter Name für eine JDK-Komponente

2.2.2 Namen und Literale

Namen

Die Bezeichner können beliebig lang sein, müssen aber mit den Buchstaben `A` ... `Z` oder `a` ... `z` bzw. _ oder `$` beginnen. Anschließend kann ein beliebiges Unicode-Zeichen folgen. Alle Zeichen sind signifikant, Groß- und Kleinschreibung wird unterschieden. Die genauen Regeln zur Namensvergabe sind im Abschnitt 2.13 zu finden.

Literale

Ein Literal ist eine Konstante eines bestimmten Datentyps. Es hat im Gegensatz zu einer Variablen keinen Namen. Es existieren Literale verschiedener Typen:

- **Numerische Literale** (ohne Dezimaltrennzeichen) können in verschiedenen Zahlensystemen angegeben werden:
  - Hexadezimale Konstante (Zahlenbasis 16), wenn sie mit `0x` oder `0X` beginnt. Beispiel: `0x42`.
  - Oktale Konstante (Zahlenbasis 8), wenn sie mit `0` beginnt. Beispiel: `042`.
  - Dezimale Ganzzahlkonstante (Zahlenbasis 10) sonst. Beispiel: `42`.

  Einer Konstanten darf ein `L` angehängt werden, um eine explizite Umwandlung von `int` in `long` zu erzwingen. Beispiel: `42L`. Bei sehr langen Zahlen besteht die Gefahr, die Übersicht über den Anzahl der Stellen zu verlieren. Daher dürfen ganze Zahlen auch den Unterstrich („_“) als optisches Trennzeichen beinhalten. Eine Semantik besitzt es nicht. Beispiel: `123_456_789`.
- **Gleitkommakonstanten** verwenden den Punkt als Trennzeichen, z.B. `3.14` zwischen Vor- und Nachkommastellen. Die Konstanten lassen sich auch in Exponentialschreibweise schreiben, beispielsweise `2.8191e10` ($2,8191^{10}$) oder `2e-12` (2^{-12}).
- **Boole'sche Literale** sind als Wahrheitswerte entweder `true` oder `false`.
- **Zeichen-Literale** können ein einzelnes Zeichen im Unicode-Format aufnehmen. Sie beginnen und enden mit einem einfachen Anführungszeichen `'`. Das Unicode-Format erlaubt die Verwendung aller weltweit vorhandenen Zeichen, wobei nicht jedes Zeichen auch unter jedem Betriebssystem darstellbar sein muss.
- **String-Literale** können Folgen von Zeichen aufnehmen. Sie beginnen und enden jeweils mit einem doppelten Anführungszeichen `"`.

Für Zeichen- und String-Literale gelten die folgenden Sonderzeichen (auch Escape-Sequenzen genannt):

Tabelle 2.1: Sonderzeichen in Zeichenketten

Zeichen	Bedeutung
`\b`	Backspace
`\n`	Newline
`\t`	Hor. Tabulator
`\"`	Anführungszeichen "

Zeichen	Bedeutung
`\f`	Formfeed
`\r`	Carriage Return
`\\`	Backslash selbst
`\'`	Anführungszeichen ‘

2.2.3 Genereller Quelltextaufbau

- Alle **Anweisungen** werden mit einem Semikolon abgeschlossen.
- Ein Block ist ein eigener Gültigkeitsbereich, in dem Anweisungen vorhanden sein und lokale Variablen deklariert werden dürfen. Ein Block ist durch geschweifte Klammern eingeschlossen. Da ein Block selbst wiederum eine Anweisung ist, darf er überall dort verwendet werden, wo Anweisungen erlaubt sind. Ein Block wird nicht mit einem Semikolon beendet.
- Ein **synchronisierter Block** kann nur von einem Thread zu einem Zeitpunkt betreten werden. Als Synchronisationspunkt wird ein Objekt einer beliebigen Klasse verwendet (siehe Abschnitt 3.3.3).

2.3 Imperative Aspekte

Dieser Abschnitt beschreibt die eingebauten Datentypen, ihre Wertebereiche und die Operatorreihenfolgen. Zusätzlich stellt er Anweisungen vor.

2.3.1 Primitive Datentypen

Die Größen der Datentypen sind plattformunabhängig festgelegt.

Ganzzahlige Datentypen

Ganzzahlige Datentypen können vorzeichenlos oder vorzeichenbehaftet sein.

Tabelle 2.2: Ganzzahlige Datentypen in Java

Datentyp	Bedeutung
boolean	Wahrheitstyp, `true` oder `false`
char	Vorzeichenloser 16-Bit-Datentyp für ein Unicode-Zeichen
byte	Vorzeichenbehafteter 8-Bit-Datentyp
short	Vorzeichenbehafteter 16-Bit-Datentyp
int	Vorzeichenbehafteter 32-Bit-Datentyp
long	Vorzeichenbehafteter 64-Bit-Datentyp

Fließkommadatentypen

Es existieren zwei verschiedene Fließkommadatentypen für unterschiedliche Rechengenauigkeiten und -geschwindigkeiten.

Tabelle 2.3: Fließkommadatentypen in Java

Datentyp	Bedeutung
float	Kleiner Zahlenbereich, zwischen `1.4E-45f` und `3.4E+38f`
double	Großer Zahlenbereich, zwischen `4.9E-324` und `1.8E+308`

Wrapper-Klassen

Für alle primitiven Datentypen existieren Wrapper-Klassen. Dabei handelt es sich um Klassen, die genau einen Wert des Datentyps aufnehmen können. Der Wert ist unver-

änderlich. Damit können Werte primitiver Datentypen Methoden übergeben werden, die eigentlich Objekte erwarten. Zusätzlich sind diese Werte auch in Datenstrukturen wie Listen usw. speicherbar. Die Klassen besitzen eine Anzahl von Methoden, um Konvertierungen des Wertes durchführen zu können.

Beispiel für `int`:

```
Integer iValue = new Integer(3);
```

Beispiel zur Konvertierung eines Strings in einen `int`-Wert ohne Fehlerbehandlung:

```
int value = Integer.parseInt("123");
```

Umwandlungen

Die Umwandlung von kleineren Datentypen (außer `boolean`) in größere geschieht automatisch. Um größere Datentypen in kleinere umzuwandeln, muss der cast-Operator (siehe Abschnitt 2.3.5) verwendet werden. Im Gegensatz zu C++ entsprechen `enum`-Werte nicht einem `int`-Wert. Daher ist eine Konvertierung von `int` in einen `enum`-Typ (siehe folgender Abschnitt) und umgekehrt nicht möglich.

2.3.2 Eigene und zusammengesetzte Datentypen

Dieser Abschnitt behandelt alle Typen, die nicht explizit als eigene Klasse implementiert werden. Der folgende Aufzähltyp gehört genau genommen nicht an diese Stelle, da dadurch auch eine Klasse beschrieben wird. Diese hat aber einen speziellen Aufbau mit einer besonderen Bedeutung, so dass die Einordnung an dieser Stelle gerechtfertigt ist.

Aufzählungen

Aufzähltypen in Java sind wesentlich mächtiger als in C++. Damit ist eine typsichere Aufzählung zusammengehöriger Daten möglich. Der Aufzähltyp selbst ist die Klasse, ein einzelner Wert ein Objekt dieser Klasse. Im einfachsten Fall werden die Bezeichnungen nur aufgezählt.

Beispiel (`State` ist die Klasse , `RED` ein Objekt der Klasse):

```
public class TrafficLight {
  public enum State { YELLOW, RED, RED_YELLOW, GREEN }

  private State state = State.RED;  // Attribut vom enum-Typ

  public void nextState() {
    // Index innerhalb State
    int index = state.ordinal();
    // Verweist index auf GREEN?
    if (index == State.values().length - 1)
      index = 0;
    else
      index ++;
    // Neuer Zustand
    state = State.values()[ index ];
  }
}
```

Weiterhin können einem `enum`-Typ auch Attribute und Methoden zugeordnet werden. Dazu wird ein `enum` wie eine einfache Klasse deklariert.

Im folgenden Beispiel erhält jeder Wert eine Anzeigedauer des Signals. Werte füllen automatisch die Attribute des `enum`-Typs in der angegebenen Reihenfolge:

```
public enum State {
  // Belegungen des Attributs "duration" eines "State".
  YELLOW(4), RED(20), RED_YELLOW(4), GREEN(40);

  private int duration;

  // Wird von den vier enum-Werten aufgerufen
  private State(int nDuration) {
      duration = nDuration;
  }

  // Zum Auslesen der Dauer
  public int getDuration() {
      return duration;
  }
}
```

Beispiel zur Verwendung von `State`:

```
State st = State.RED;
int duration = st.getDuration();
```

Sollen die Methodenimplementierungen abhängig vom enum-Wert ausgelegt sein, so können einzelne enum-Werte auch die Standardimplementierung des Typs überschreiben. Näheres dazu ist in der Java-Dokumentation zum JDK zu finden [ORA01].

Eindimensionale Arrays

Arrays (Felder) sind Objekte, die eine feste Anzahl Elemente desselben Typs aufnehmen. Der Zugriff auf die Elemente erfolgt über ganzzahlige, nicht negative Indizes, deren Zählung bei `0` beginnt.

Ein Array kann Werte primitiver Datentypen direkt beinhalten. Im Falle von Objekten dagegen werden nur Referenzen auf diese im Array abgelegt. Alle Elemente im Array sind nach der Erzeugung automatisch initialisiert (`0`, `0.0`, `false` bzw. `null` bei Referenzen).

Beispiel für ein eindimensionales Array der Länge 10, in dem alle Elemente mit `0` initialisiert sind:

```
int[] values = new int[ 10 ];
```

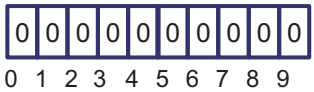

Abbildung 2.5: int-Array mit automatischer Initialisierung

Beispiel für ein Array der Länge 3 mit initialisierten Elementen:

```
int[] values = {42, 271, 314};
```

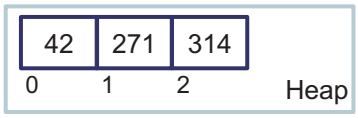

Abbildung 2.6: int-Array mit manueller Initialisierung

Wie bereits erwähnt, kann ein Array keine Objekte beinhalten. Stattdessen enthält es nur Referenzen auf Objekte.

Beispiel für ein eindimensionales Array mit Referenzen auf Objekte:

```
Fraction[] fr = new Fraction[ 10 ];
```

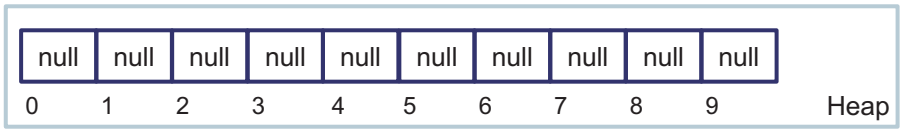

Abbildung 2.7: Objekt-Array mit automatischer Initialisierung

Alle Elemente sind mit `null` initialisiert. Um Objekte im Array abzulegen, müssen Referenzen auf die Objekte eingetragen werden.

Beispiel für ein Array der Länge 2 mit Referenzen auf Objekte:

```
Fraction[] fr = {new Fraction(),
                 new Fraction(1)};
```

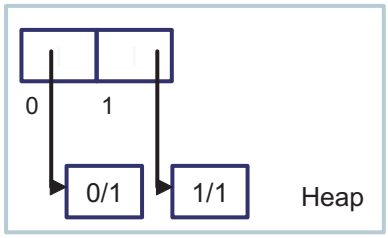

Abbildung 2.8: Objekt-Array mit manueller Initialisierung

Der Zugriff auf einzelne Arrayelemente erfolgt durch den Operator `[]`. Es findet beim Zugriff eine Indexprüfung statt, so dass Fehlzugriffe nicht möglich sind. Stattdessen würde eine Ausnahme (siehe Abschnitte 2.3.6 und 2.5.6) ausgelöst werden.

Beispiel für lesenden und schreibenden Zugriff:

```
int value = values[ 0 ];   // Lesen
values[ 0 ] = 42;          // Schreiben
```

Die Größe eines Arrays lässt sich durch das Auslesen des Attributs `length` ermitteln.

Beispiel:

```
int size = values.length;
```

Ein Arrayname in Java ist lediglich eine Referenz auf das eigentliche Array-Objekt, das immer auf dem Heap liegt. Deshalb wird mit der folgenden Anweisung zwar eine Referenz auf ein Array auf dem Stack abgelegt, das Array selbst existiert noch nicht. Es muss explizit erzeugt werden.

Beispiel:

```
// Noch kein Array erzeugt!
int[] values;
// Array auf dem Heap erzeugen.
values = new int[ 42 ];
```

Mehrdimensionale Arrays

Die in der Einführung zu eindimensionalen Arrays getroffenen Aussagen gelten auch hier. Hinzu kommt, dass mehrdimensionale Arrays asymmetrisch oder symmetrisch aufgebaut sein können. Ein mehrdimensionales Array ist ein eindimensionales Array, das seinerseits Referenzen auf Arrays enthält. Alle Elemente des Arrays werden wie im eindimensionalen Fall automatisch initialisiert.

Beispiel für ein mehrdimensionales Array, in dem alle Elemente mit `0` initialisiert sind:

```
int[][] values = new int[ 4 ][ 2 ];
```

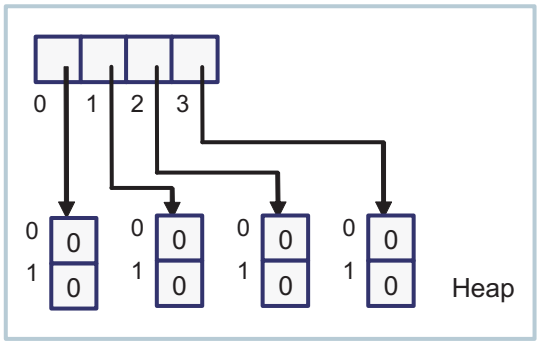

Abbildung 2.9: Mehrdimensionales int-Array mit automatischer Initialisierung

Beispiel für ein manuell initialisiertes, mehrdimensionales Array:

```
int[][] values = {{2,4},{8,16}};
```

Mehrdimensionale Arrays können asymmetrisch sein.

Beispiel für ein asymmetrisches, mehrdimensionales Array, dessen Elemente automatisch mit `0` initialisiert werden:

```
int[][] values = new int[ 3 ][];
for (int i = 0; i < values.length; i++)
  values[ i ] = new int[ i + 1 ];
```

Beispiel für ein manuell initialisiertes, asymmetrisches Array:

```
int[][] values = {{0},{1,2},{3}};
```

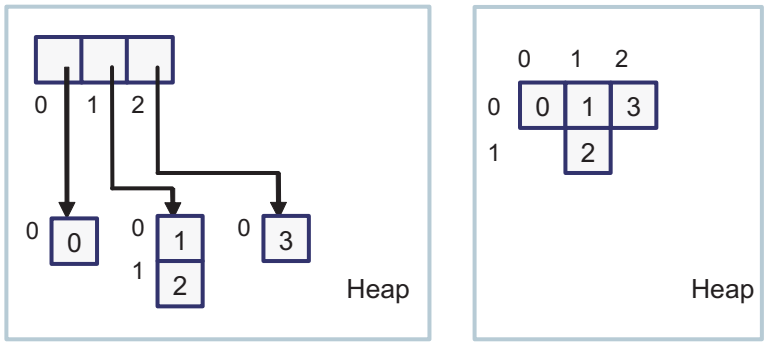

Abbildung 2.10: Mehrdimensionales, asymmetrisches int-Array

Die Abbildung zeigt links das Array mit den Referenzen, rechts fehlen sie in der Darstellung.
Der Zugriff auf einzelne Arrayelemente erfolgt durch mehrfache Angabe des Operators `[]`.

Beispiel für lesenden und schreibenden Zugriff:

```
int value = values[ 0 ][ 1 ];   // Lesen
values[ 0 ][ 1 ] = 42;          // Schreiben
```

Die einzelnen Größen der Zeilen und Spalten läßt sich durch Auslesen des Attributs `length` ermitteln.

Beispiel:

```
int columns = values.length;
int row0    = values[ 0 ].length;
```

2.3.3 Speicherbereiche für Daten

Java unterstützt zwei sogenannte Speicherklassen zur Datenablage:

- **Automatisch**: Die (lokalen) Daten werden beim Betreten eines Blocks automatisch erzeugt und beim Verlassen wieder entfernt. Lokale Variablen sind manuell zu initialisieren. Ihr Wert ist sonst undefiniert. Unveränderliche Daten werden als `final` deklariert.
  Beispiel:

  ```
  public void doWhatever() {
    int counter = 0;
    final int end = 10;
    // ...
  }
  ```

- **Statisch**: Nur Klassenattribute können statisch sein. Diese Daten werden einmalig beim Laden der Klasse angelegt und nicht wieder gelöscht (siehe Abschnitt 2.5.6).

Daneben existiert noch die programmgesteuerte Verwaltung von Daten auf dem **Heap** (siehe Abschnitt 2.4.2).

2.3.4 Ausdrücke

Ein Ausdruck wird in Java durch die Anwendung von Operatoren gebildet. Auch die Zuweisung ist ein Ausdruck und bekommt als Wert den zugewiesenen Ausdruck.

2.3.5 Operatoren und ihre Reihenfolgen

- Die Operatoren sind in der Tabelle ihrer Priorität nach sortiert. Alle Operatoren innerhalb eines Bereichs haben dieselbe Priorität.
- Der Zuweisungsoperator, die zusammengesetzten Operatoren, der Auswahloperator und alle einstelligen Operatoren werden von rechts nach links ausgewertet, alle anderen Operatoren von links nach rechts.

Eine andere Reihenfolge kann durch Klammern erzwungen werden.

Tabelle 2.4: Operatoren und ihre Priorität P

P	Operatoren	Bedeutung	Beispiel
13	`++ --`	Inkrement, Dekrement (Wert um `1` erhöhen oder verringern)	`int r = ++a;`
	`+ -`	Vorzeichen numerischer Daten	`int r = -a;`
	`~`	Einerkomplement ganzzahliger Daten	`int r = ~a;`
	`!`	Logisches „Nicht“ ganzzahliger Daten	`if (!(a == b))`
	`( )`	Cast (zur Laufzeit geprüfte Typkonvertierung)	`int r = (int)2.3;`
12	`* /`	Multiplikation bzw. Division von Zahlen	`int r = 12 * 23;`
	`%`	Modulo-Division ganzzahliger Werte	`int r = a % 42;`
11	`+ -`	Addition oder Subtraktion	`int r = a + 12;`
	`+`	Verkettung von Strings	`String n = "H"+"A";`
10	`<<`	Bitweises Linksschieben um eine angegebene Anzahl Stellen. Dieses entspricht einer Multiplikation mit 2.	`int r = a << 12;`
	`>>`	Bitweises Rechtsschieben um eine angegebene Anzahl Stellen. Das Vorzeichenbit bleibt erhalten. Das entspricht einer Ganzzahldivision durch 2.	`int r = a >> 12;`
	`>>>`	Bitweises Rechtsschieben um eine angegebene Anzahl Stellen, wobei das Vorzeichenbit `0` wird.	`int r = a >>> 12;`
9	`< <=` `> >=`	Numerischer Vergleich (kleiner, kleiner-gleich, größer, größer-gleich)	`if (r >= 0)`
	`instanceof`	Der Operator liefert als Resultat `true` zurück, wenn das Objekt der angegebenen Klasse oder einer davon abgeleiteten Klasse entstammt, siehe auch 2.12.	`No n = new Int();` `if (n instanceof Int)`
8	`== !=`	Vergleich primitiver Daten (gleich, newline ungleich)	`if (r == 0)`
	`== !=`	Vergleich von Referenzen auf Objekte (gleich, ungleich)	`Fraction fr;` `if (fr == null)`
7	`&`	Bitweises „Und“ zweier ganzzahliger Daten	`int r = b & 2;`

P	Operatoren	Bedeutung	Beispiel
	`&`	Logische „Und“-Verknüpfung zweier Boole’scher Ausdrücke mit vollständiger Auswertung: Das Ergebnis wird von links nach rechts ermittelt. Auch wenn das Ergebnis bereits feststeht, werden alle Bedingungen ausgewertet.	`if (r == 0 & s == 1)`
6	`^`	Bitweises „XOR“ zweier ganzzahliger Werte	`int r = b ^ 0x10;`
	`^`	Logisches „XOR“ mit vollständiger Auswertung des Ausdrucks: Die Ermittlung des Resultates erfolgt von links nach rechts. Auch wenn das Ergebnis bereits feststeht, werden alle Bedingungen ausgewertet.	`if (r == 0 ^ s == 1)`
5	`|`	Bitweises „Oder“ zweier ganzzahliger Argumente	`int r = b | 0x01;`
	`|`	Logisches „Oder“ zweier Boole’scher Ausdrücke mit vollständiger Auswertung: Das Ergebnis wird von links nach rechts ermittelt. Auch wenn das Ergebnis bereits feststeht, werden alle Bedingungen ausgewertet.	`if (r == 0 | s == 1)`
4	`&&`	Logisches „Und“ zweier Boole’scher Ausdrücke. Die Auswertung erfolgt mit Hilfe der sogenannten „Kurzschlusslogik“: Sobald das Ergebnis der Berechnung feststeht, werden die restlichen Bedingungen nicht mehr untersucht.	`if (r == 0 && s == 1)`
3	`||`	Logisches „Oder“ zweier Boole’scher Ausdrücke. Die Auswertung erfolgt auch mit Hilfe der „Kurzschlusslogik“: Sobald das Ergebnis der Berechnung feststeht, werden die restlichen Bedingungen nicht mehr untersucht.	`if (r == 0 || s == 1)`
2	`? :`	Bedingte Zuweisung. Wenn die Bedingung vor dem Fragezeichen wahr ist, ist das Ergebnis des Ausdrucks der Ausdruck `a`, sonst `b`.	`min = a < b ? a : b;`
1	`= >>= += >>>= -= <<= /= &= *= |= %= ~=`	Zuweisungsoperator und alle zusammengesetzten Operatoren (Operation mit anschließender Zuweisung)	`int re += 42;`

2.3.6 Anweisungen

Jeder Ausdruck wird dadurch zu einer Anweisung, dass er durch ein Semikolon abgeschlossen ist. Aber auch ein Block ist eine Anweisung.
In Abfragen und Schleifen ist ein anschließender Block nur dann erforderlich, wenn zur Abfrage oder Schleife mehr als eine Anweisung gehört.

if-Abfrage

Test eines Boole’schen Ausdrucks. Der anschließende Codeblock bzw. die folgende Anweisung wird ausgeführt, wenn die Bedingung `true` ist. Der optionale `else`-Zweig wird ausgeführt, wenn die Bedingung `false` ausgewertet wird.

Beispiel:

```
if (age == 65)
  pension = true;
else
  job = true;
```

switch-Anweisung

Hierbei handelt es sich um eine Mehrfachauswahl einer Anweisung in Abhängigkeit eines ganzzahligen numerischen Wertes oder eines Strings. Dazu wird der Wert mit mehreren möglichen Belegungen verglichen. Bei Übereinstimmung wird die dazugehörige Anweisung oder der Block mit Anweisungen ausgeführt. Ist keine Übereinstimmung mit einem der Werte vorhanden, kann eine optionale Standardanweisung (`default`) ausgeführt werden. Der `default`-Test darf nur nach den `case`-Anweisungen auftreten.

Beispiel:

```
switch (phase) {           // phase ist ein ganzzahliger Wert
  case 1:  statement1();
           break;
  case 2:  statement2();
           break;
  default: defaultStatement();
}
```

Die `break`-Anweisungen verhindern, dass nach Ausführung eines Befehls die darauf folgenden Anweisungen ebenso ausgeführt werden. Stattdessen wird der `switch`-Block verlassen. Sollen dagegen die folgenden Anweisungen auch bearbeitet werden, so kann der `break`-Befehl entfallen.

Beispiel:

```
switch (phase) {          // phase ist ein ganzzahliger Wert
  case 1:  statement1(); // bearbeitet auch Statement2
  case 2:  statement2();
           break;
}
```

Für Zeichenketten funktioniert die `switch`-Anwendung ähnlich. Allerdings werden hier die Strings durch Aufruf der `equals`-Methode verglichen (siehe auch Abschnitt 2.7.6).

Beispiel:

```
switch (name) {           // name ist ein String
  case "Vogelsang": statement1();
                    break;
  case "Gmeiner":   statement2();
                    break;
  default: defaultStatement();
}
```

while-Schleife

Die Schleife wird so lange ausgeführt, bis die angegebene Bedingung `false` ist. Die Schleifenbedingung wird vor dem ersten Durchlauf geprüft.

Beispiel:

```
while (xpos > 0)
  newPos(xpos--); // Kein Block (nur eine Anweisung)
```

do/while-Schleife

Die Schleife wird so lange ausgeführt, bis die angegebene Bedingung `false` ist. Die Schleifenbedingung wird nach dem ersten Durchlauf geprüft.

Beispiel:

```
do
  newPos(xpos--); // Kein Block (nur eine Anweisung)
while (xpos > 0);
```

for-Schleife

Es gibt zwei Varianten der `for`-Schleife. Die „klassische“ Form besteht aus drei Teilen:

1. Beim Start der Schleife kann einmalig eine Initialisierung (z.B. eines Schleifenzählers) erfolgen. Die Variable darf an dieser Stelle auch deklariert werden. Sie ist nach dem Verlassen der Schleife nicht mehr zugreifbar.
2. Vor einer (erneuten) Ausführung des Schleifenrumpfes wird geprüft, ob die Abbruchbedingung erreicht ist.
3. Nach jeder Ausführung des Schleifenrumpfes besteht die Möglichkeit, eine zusätzliche Aktion durchzuführen. Hierbeit handelt es sich häufig um das Erhöhen oder Verringern eines Schleifenzählers.

Beispiel:

```
int[] values = {1, 2, 4, 8, 16};
for (int i = 0;          // 1. Initialisierung
     i < values.length; // 2. Laufbedingung
     i++) {             // 3. Nach jedem Schleifendurchlauf
  count += values[ i ];
}
```

Das folgende Beispiel verwendet Iteratoren, um auf die Elemente einer Datenstruktur zuzugreifen. Die Containerklassen aus dem Paket `java.util` implementieren die Iteratoren, mit deren Hilfe alle Elemente des Containers unabhängig von dessen konkreter Realisierung ausgelesen werden. Somit werden Algorithmus und Datenstruktur entkoppelt.

Beispiel zum Zugriff auf alle Elemente eines Vektors zu finden:

```
Vector<Integer> values = new Vector<Integer>();
values.add(new Integer(1));
values.add(new Integer(2));

for (Iterator<Integer> iter = values.iterator();
     iter.hasNext(); ) {
  count += iter.next().intValue();
}
```

Eine vereinfachte Syntax der Schleife erlaubt das Vorwärts-Iterieren über Arrays und bestimmte Datenstrukturen.

Beispiel für ein Array:

```
int[] values = {1, 2, 4, 8, 16};
for (int val: values) {
  count += val;
}
```

In der Schleife wird mit `val` eine neue Variable deklariert. Diese erhält in jedem Schleifendurchlauf den Folgewert aus dem Array, bis das Ende des Arrays erreicht wurde.
Mit der zweiten Schleifenform kann über alle Klassen iteriert werden, die die generische Schnittstelle `java.lang.Iterable` implementieren. Dieses ist bei vielen Container-Klassen aus `java.util` der Fall.

Vereinfachtes Iterator-Beispiel:

```
Vector<Integer> values = new Vector<Integer>();
values.add(new Integer(1));
values.add(new Integer(2));

for (Integer val: values) {
  count += val.intValue();
}
```

In der Schleife wird mit `val` eine neue Variable deklariert. Diese erhält in jedem Schleifendurchlauf den Folgewert aus dem Vektor, bis das Ende des Vektors erreicht wurde.
Die neue Syntax lässt sich auch gut dazu verwenden, alle Werte einer Aufzählung zu durchlaufen.

break-Anweisung

Die `break`-Anweisung hat mehrere Aufgaben:

- Verlassen der aktuellen Schleife und Fortsetzen mit der der Schleife folgenden Anweisung.
  Beispiel:

  ```
  while (i < 10) {
    if (error()) {
      break;          // Geht zu NextStatement()
    }
  }
  nextStatement();
  ```

- Verlassen der Schleifen, die durch eine Sprungmarke gekennzeichnet sind.
  Beispiel:

  ```
  outer:
  while (j < 99) {
    while(i < 10) {
      if (error()) {
        break outer;     // Geht zu NextStatement()
      }
    }
  }
  nextStatement();
  ```

In dem Beispiel werden alle Schleifen verlassen, die sich direkt an das Label `outer` anschließen. So können mehrere geschachtelte Schleifen beendet werden, was mit einem einfach `break` nicht möglich ist.

- Innerhalb einer `switch`-Anweisung wird mit `break` verhindert, dass nachfolgende Anweisungen anderer Fälle auch abgearbeitet werden. Ein Beispiel ist im Abschnitt über die `switch`-Anweisung zu finden.

continue-Anweisung

`continue` wird innerhalb von Schleifen eingesetzt. Mit dieser Anweisung wird die Bearbeitung des Schleifenrumpfes abgebrochen und ein neuer Schleifendurchlauf gestartet. Analog zur `break`-Anweisung gibt es zwei Varianten:

- Beendigung des Schleifenrumpfes und Fortsetzen mit einem neuen Durchlauf der Schleife, in der die `continue`-Anweisung steht.

  Beispiel:

```
while(i < 10) {
  if (error()) {
    continue;       // Neuer Schleifendurchlauf
                    // (Sprung zu while)
  }
  nextStatement();
  i++;
}
```

- Beendigung des Schleifenrumpfes und Fortsetzen mit einem neuen Durchlauf einer Schleife durch Angabe einer Markierung.

  Beispiel:

```
outer:
while(j < 100) {
  while(i < 10) {
    if (error()) {
      continue outer;  // Komplett neuer Durchlauf
                       // der äusseren Schleife
    }
    nextStatement();
    i++;
  }
  j++;
}
```

  Wäre die äußere Schleife eine `for`-Schleife, dann würde diese neu mit ihrem Initialwert gestartet werden.

return-Anweisung

Mit `return` wird eine Methode beendet und zum Aufrufer zurückgekehrt. Ein Rückgabewert wird dann übergeben, wenn die Methode einen solchen erlaubt.

```
int min(int a, int b) {
  return a < b ? a : b;
}
```

goto-Anweisung

Das Schlüsselwort `goto` ist in Java reserviert, aber (glücklicherweise) nicht implementiert.

try/catch-Anweisungen

Die Fehlerbehandlung mit Exceptions ist in Abschnitt 2.8 beschrieben.

throw-Anweisung

Die Fehlerbehandlung mit Exceptions ist in Abschnitt 2.8 beschrieben.

2.4 Referenzen

Die folgenden Abschnitte beschreiben die „normalen“ Referenzen. Weiterführende Informationen zu den unterschiedlichen Arten von Referenzen sind in Abschnitt 3.1 zu finden.

2.4.1 Starke Referenzen

Referenzen sind Verweise auf Objekte auf dem Heap. Es existieren zwei vordefinierte Referenzwerte:

- `null`: Die Referenz verweist auf kein Objekt.
- `this`: Die Referenz verweist innerhalb einer nicht statischen Methode auf das Objekt, auf dem die Methode aufgerufen wurde.

Beispiele:

```
Fraction fr;            // Nicht initialisierte Referenz
fr = new Fraction(10); // Verweist auf ein neues Objekt.
```

Referenzen haben einen Einfluss auf die Speicherverwaltung. Nähere Informationen dazu sind in Abschnitt 2.4.2 zu finden.

2.4.2 Dynamische Speicherverwaltung

In Java wird der dynamisch vom Heap angeforderte Speicher mittels Operatoren verwaltet:

- `new`: Es wird ein Objekt auf dem Heap erzeugt. Der Konstruktor der Klasse wird aufgerufen, um das Objekt zu initialisieren. Die Rückgabe des Operators besteht aus einer Referenz auf das neu angelegte Objekt.

Beispiel für Speicheranforderung mit Konstruktoraufruf

```
Fraction fr1 = new Fraction();
Fraction fr2 = new Fraction(1);
```

- `new[]`*: Der Operator erzeugt Arrays von Objekten oder Variablen. Bei Objekten wird nur eine mit* `null` *initialisierte Referenz im Array abgelegt. Enthält das Array Daten primitiver Typen, so werden diese mit* `0` *bzw.* `false` *initialisiert (siehe Abschnitt 2.3.2).*

Objekte können nicht manuell vom Heap gelöscht werden. Stattdessen entfernt der sogenannte Garbage-Collector die Objekte, die nicht mehr über starke Referenzen erreichbar sind. Dieses geschieht immer dann, wenn auf dem Heap nicht mehr genügend Speicherplatz zur Verfügung steht. Der genaue Ablauf wird in Abschnitt 2.5.4 beschrieben.

2.5 Objekte und Klassen

In Java werden Objekte immer als Instanzen von Klassen auf dem Heap erzeugt. Das Unterscheidungsmerkmal eines Objektes (seine Identität) ist die Referenz auf das Objekt.

2.5.1 Deklaration und Aufbau

Eine Klasse wird durch das Schlüsselwort `class` eingeleitet.

Beispiel für einen Klassenrumpf ohne Methoden und Attribute:

```
public class Fraction {
}
```

2.5.2 Klassenrechte

Jede Klasse erhält ein **Zugriffsrecht**.

Java unterscheidet zwei verschiedene Zugriffsrechte:

- `public`: Die Klasse kann von beliebigen anderen Klassen verwendet werden. Je Quelltextdatei darf nur eine öffentliche Klasse vorhanden sein. Der Klassenname muss dem Namen der Quelltextdatei (ohne Endung `.java`) entsprechen.

  Beispiel für ein öffentliches Recht einer Klasse:

  ```
  public class Fraction {
  }
  ```

- *<keine Angabe>*: Die Klasse kann von anderen Klassen aus demselben Paket verwendet werden. Es dürfen sich mehrere Klassen mit diesem Paketrecht in einer Quelltextdatei befinden.

  Beispiel für das Paketrecht einer Klasse:

  ```
  class Fraction {
  }
  ```

2.5.3 Konstruktor und Initialisierungsreihenfolge

Der **Konstruktor** wird aufgerufen, nachdem der Speicher für das Objekt bereitgestellt wurde.

- Er ist für die Initialisierung des Objektes verantwortlich.
- Der Name entspricht dem der Klasse.
- Ein Konstruktor hat keinen Rückgabewert.
- Eine Klasse kann mehrere Konstruktoren besitzen, die sich durch ihre Signaturen unterscheiden.
- Ist kein Konstruktor vorhanden, so wird automatisch der Standardkonstruktor (Konstruktor ohne Parameter) erzeugt.
- Für die Übergabeparameter gelten dieselben Bedingungen wie für Methoden (siehe Abschnitt 2.5.6).

Beispiel für Konstruktoren:

```
public class Fraction {
  public Fraction(long cnt, long denom){
    /* ... */
  }

  public Fraction() {
    /* ... */
  }
  // ...
}
```

Beispiel für das Erzeugen eines Objektes:

```
Fraction fr1 = new Fraction();
Fraction fr2 = new Fraction(2, 3);
```

Die Zuweisung einer Referenz an eine andere erstellt keine Kopie des Objektes. Stattdessen verweisen beide Referenzen danach auf dasselbe Objekt.
Beim Erzeugen eines Objektes wird eine feste Initialisierungsreihenfolge eingehalten:

1. Die Attribute werden in der Reihenfolge ihrer Deklaration initialisiert.
2. Anschließend erfolgt der Konstruktoraufruf.

Beispiel (die Kommentare geben die Reihenfolge an):

```
public class Calculator {
  private Fraction fr1 = new Fraction();    // 1.
  private Fraction fr2 = new Fraction(2);   // 2.

  public Calculator() {                     // 3.
    /* ... */
  }
  // ...
}
```

2.5.4 Freigabereihenfolge

Bevor der Speicher eines Objektes wieder freigegeben wird, ruft die virtuelle Maschine eine spezielle Methode der Klasse auf. Diese kann ihrerseits noch Aufräumarbeiten durchführen.
Vor dem Löschen des Objektes wird dessen Methode `finalize` aufgerufen. Diese kann überschrieben werden, um eigene Aufräumarbeiten durchzuführen. Es ist aber nicht garantiert, dass ein Objekt überhaupt gelöscht und damit die Methode aufgerufen wird, wenn es keine Referenz auf das Objekt mehr gibt. Weiterhin sollte diese Methode auf keinen Fall zeitintensive Operationen durchführen, da die Methode während der Speicherbereinigung durch den Garbage-Collector aufgerufen wird und sich so die Zeit zur Speicherbereinigung für die virtuelle Maschine unvorhersehbar verlängern würde.

Beispiel für die `finalize`-Methode:

```
public class Fraction {
  // ...
  protected void finalize() {
    /* ... */
  }
}
```

Das Verhalten bei vererbten Klassen ist in Abschnitt 2.7.2 beschrieben.

2.5.5 Attribute

Attribute beinhalten die Daten und somit den Zustand eines Objektes.

Normale Attribute

Für jedes Objekt existiert eine eigene Kopie jedes Attributs.

Beispiel für Attribute:

```
public class Fraction {
  private long counter     = 0;
  private long denominator = 1;
  // ...
}
```

Attribute können direkt während ihrer Deklaration oder aber durch den Konstruktor initialisiert werden.

Statische Attribute

Sie beinhalten die gemeinsam genutzten Daten aller Objekte einer Klasse. Statische Attribute existieren auch, wenn es kein Objekt der Klasse gibt.

Beispiel für ein statisches Attribut:

```
public class Fraction {
  private static long referenceCount = 0;
  // ...
}
```

Statische Attribute sollten während ihrer Deklaration möglichst auch sofort initialisiert werden.

2.5.6 Methoden

Durch Methoden kann auf ein Objekt oder eine Klasse zugegriffen werden. Methoden werden anhand ihrer Signatur unterschieden.

Normale Methoden

Die Methode manipuliert ein Objekt.

Beispiel für eine Methode:

```
public class Fraction {
  private long counter     = 0;
  private long denominator = 1;

  public long getCounter() {
    return counter;
  }

  public void setCounter(long nCounter) {
    counter = nCounter;
  }
}
```

Beispiel für einen Methodenaufruf:

```
Fraction fr = new Fraction();
fr.setCounter(42);   // Aufruf von setCounter
```

Soll kein Ergebnis zurückgegeben werden, so muss der Rückgabetyp `void` lauten. Übergabeparameter werden immer als Paare, bestehend aus Typ und Namen des Parameters, angegeben.

Beispiel:

```
public void setCounter(long nCounter) {
  counter = nCounter;
}
```

Die Parameterübergabe und Ergebnisrückgabe primitiver Datentypen erfolgt als Kopie. Objekte werden nur als Referenzparameter übergeben. Änderungen an den Parametern wirken sich bei Objekten also immer auf das Original aus.
Methoden können im Fehlerfall Ausnahmen (Exceptions) auslösen. Die Details sind im Abschnitt 2.8 zur Fehlerbehandlung beschrieben.

Überladen von Methoden

Man spricht vom Überladen von Methoden, wenn eine Klasse mehrere Methoden mit identischem Namen und unterschiedlichen Übergabeparametern besitzt. Der Compiler wählt die Methode aus, für die am wenigsten Parameterkonvertierungen erforderlich sind. Durch die Konvertierung gehen niemals Informationen verloren.

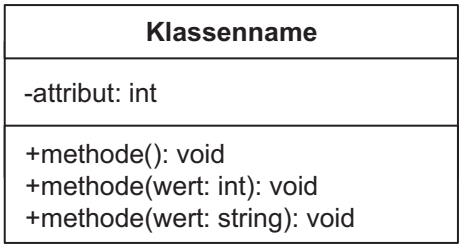

Abbildung 2.11: Überladen einer Methode als UML-Diagramm

Im Wesentlichen stellt das Überladen einen Komfortgewinn für den Entwickler dar: Er muss keine explizite (manuelle) Konvertierung der Argumente im Aufruf durchführen.

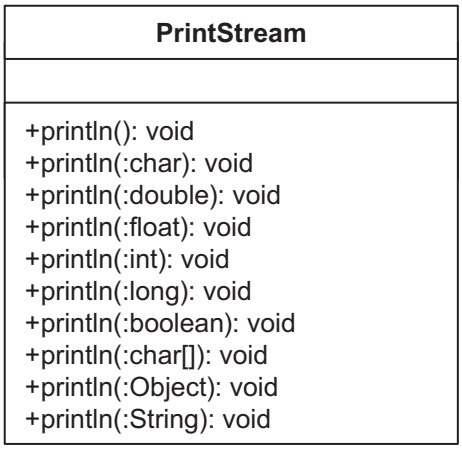

Abbildung 2.12: Überladen am Beispiel der PrintStream-Klasse

Das Überladen von Methoden kann auch als Ersatz für die Standard-Parameter, die es in C++ gibt, verwendet werden. C++ erlaubt die Vergabe von Standardwerten, die im Aufruf weggelassen werden dürfen.

Im folgenden Beispiel kann die Methode `setTaxClass` *mit oder ohne Parameter aufgerufen werden:*

```
public class Employee extends TaxPayer {
  // ...
  public void setTaxClass(){
    setTaxClass(4);
  }
  public void setTaxClass(int aTaxClass){
    this.taxClass = aTaxClass;
  }
}
```

Der Compiler ermittelt die am besten passende Methode anhand der folgenden Regeln:

- Die Anzahl der Parameter muss zum Aufruf passen.
- Alle Übergabeparameter müssen in die Parametertypen der Methode konvertierbar sein.

  Beispiel:

  ```
  public class Klasse {
    void methode(long x) { ... }
  }
  ```

  Aufruf:

  ```
  Klasse kl = new Klasse();
  kl.methode(123);    // int wird zu long
  ```

- Für alle Übergabeparameter wird der kleinste passende Typ gesucht, wobei niemals in einen kleineren Typ umgewandelt wird. Beispiel:

  `byte` → `int`, vor `byte` → `long`.
- Es wird die Methode mit der kleinsten Summe der „Konvertierungsabstände“ aller Parameter gewählt. Existieren mehrere Methoden mit gleichem, minimalem Abstand, so wird ein Übersetzungsfehler gemeldet.

  Beispiel:

  ```
  public class Klasse {
    void methode(long x, int  y) { } // 1. Methode
    void methode(int  x, long y) { } // 2. Methode
  }
  ```

  Aufruf:

  ```
  Klasse kl = new Klasse();
  kl.methode(123,      456      ); // Fehler!
  kl.methode((long)123,456      ); // Aufruf der 1. Methode
  kl.methode(123,      (long)456); // Aufruf der 2. Methode
  ```

Statische Methoden

Diese werden direkt über den Klassenamen aufgerufen und benötigen daher kein Objekt. Sie können nur auf statische Attribute einer Klasse zugreifen.

Beispiel für eine statische Methode:

```
public class Fraction {
  private static long referenceCount = 0;

  public static long getReferenceCount() {
    return referenceCount;
  }
}
```

Beispiel für einen statischen Methodenaufruf (beide Varianten sind zulässig):

```
Fraction fr = new Fraction();
int z = Fraction.getReferenceCount();
int z = fr.getReferenceCount();  // unüblich
```

Java erlaubt auch das statische Importieren statischer Methoden. So kann die Angabe des Klassennamens entfallen.

Beispiel:

```
import static java.lang.Math.sqrt;

public class Test {
    public static void main(String args[]) {
        System.out.println("Wurzel aus 10: " + sqrt(10.0));
    }
}
```

`sqrt` ist eine statische Methode der Klasse `Math`. Es können auch alle statischen Methoden einer Klasse direkt verfügbar gemacht werden. Dazu wird die Schreibweise `.` verwendet.

```
import static java.lang.Math.*;
```

Das folgende Beispiel verwendet eine statische Methode, um ein sogenanntes „Singleton“ zu definieren. Dabei handelt es sich um eine Klasse, von der nur maximal ein Objekt erzeugt werden kann. Der Trick besteht darin, den einzigen Konstruktor als privat zu deklarieren und die Erzeugung des Objektes in eine statische Methode zu verlagern. Da die Methode zur selben Klasse gehört, darf sie den privaten Konstruktor verwenden. Die Methode prüft vor dem Erzeugen der Instanz nach, ob bereits eine vorliegt. Ist das der Fall, so wird die Referenz darauf zurückgegeben. Andernfalls wird ein neues Objekt erzeugt. Expertenhinweis: Diese Implementierung sollte nicht in einer Multithreading-Umgebung eingesetzt werden, da die Synchronisation fehlt.

Beispiel für ein Singleton:

```
public class Something {
  private static Something myInstance = null;

  // Privater Konstruktor!!!!
  private Something(){}

  public static Something getInstance() {
    if (myInstance == null) {
      myInstance = new Something();
    }
    return myInstance;
  }
}
```

Die Instanzerzeugung erfolgt nun — wie oben beschrieben — durch Aufruf der statischen Methode:

```
public class TestSomething {
  public static void main(String args[]) {
    Something some1 = Something.getInstance();
    Something some2 = Something.getInstance();
```

```
    // some1 und some2 verweisen auf dasselbe Objekt.
  }
}
```

Programmstart

Zum **Programmstart** wird die statische Methode `main` einer Klasse aufgerufen.
Aufbau der Methode:

- `public static void main(String[] args)`: `args` enthält alle Kommandozeilenparameter, die beim Aufruf des Programms übergeben wurden.
- Beliebig viele Klassen dürfen diese statische Methode beinhalten. Beim Programmstart wird die Klasse angegeben, deren main-Methode aufgerufen werden soll.

Standardmethoden und Namensschemata

Da Attribute möglichst privat sein sollten, muss der Zugriff darauf über Methoden erfolgen. Es gibt ein Namensschema für solche Methoden.
Soll der Zugriff auf das private Attribut `int counter` gewährt werden, so gilt für die Methodennamen:

- `int getCounter()`: Die Getter-Methode liest den Wert des Zählers.
- `void setCounter(int nCounter)`: Die Setter-Methode schreibt den Wert des Zählers.

Allgemein gilt:

- Der Attributname fängt immer mit einem Kleinbuchstaben an.
- Der Name des Getters beginnt mit `get` und angefügtem Attributnamen, dessen Anfangsbuchstabe großgeschrieben wird. Ausnahme: Ist der Attributtyp `boolean`, so beginnt der Name des Getters mit `is`.
- Der Name des Setters beginnt mit `set` und angefügtem Attributnamen, dessen Anfangsbuchstabe großgeschrieben wird.

Variable Anzahl Methodenparameter

Es gibt immer wieder Fälle, in denen die Anzahl der Übergabeparameter einer Methode nicht zur Übersetzungszeit feststeht. Bis Java 5 mussten dazu die Parameter Objekte sein, deren Referenzen innerhalb eines Arrays abgelegt wurden. Das Array wird anschließend übergeben. In Java 5 wurde ein Mechanismus aus C++ eingeführt, der eine variable Anzahl Übergabeparameter erlaubt. Dazu werden dem Typ des Parameters drei Punkte `...` angehängt. Die variablen Parameter müssen die letzten der Methode sein, falls auch noch „normale“ Parameter übergeben werden.

Im Beispiel akzeptiert die Methode `dump` beliebig viele Objekte als Parameter:

```
public static void dump(Object... args) {
  // Zugriff auf alle Objekte
  for (Object arg: args) {
    System.out.println(arg);
  }
}
```

Beispiel mit einem möglichen Aufruf der Methode:

```
dump("1", "Hallo");
```

2.5.7 Attribut- und Methodenrechte

Die Vergabe von Zugriffsrechten erlaubt eine Festlegung, welche Klasse auf Attribute oder Methoden einer anderen Klasse zugreifen darf.
Jedes Attribut und jede Methode erhält separat eine Rechtezuweisung:

- `public`: Alle anderen Klassen dürfen auf diese Methoden oder das Attribut zugreifen.
- `private`: Nur die Klasse selbst darf darauf zugreifen.
- `protected`: Nur die eigene Klasse, davon abgeleitete Klassen (siehe Abschnitt 2.7 zur Vererbung) und Klassen aus demselben Paket dürfen darauf zugreifen.
- *<keine Angabe>*: Der Zugriff ist auf Klassen aus demselben Paket beschränkt.

Die folgende Tabelle fasst die Rechte noch einmal kurz zusammen:

Tabelle 2.5: Methodenrechte

auf / **von**	**public**	**protected**	*keine Angabe*	**private**
Selber Klasse	ja	ja	ja	ja
Klasse im selben Paket	ja	ja	ja	nein
Abgeleiteter Klasse in anderem Paket	ja	ja	nein	nein
Keiner abgeleiteten Klasse, anderem Paket	ja	nein	nein	nein

Beispiel für Rechte:

```
public class Fraction {
  private long counter;
  private long denominator;

  public long getCounter(){ /* ... */ }
  public void setCounter(long nCounter){
    /* ... */
  }
}
```

Hinweise zu Zugriffsrechten:

- Es existiert kein Schlüsselwort `package`. Wird kein Zugriffsrecht angegeben, so handelt es sich um das Zugriffsrecht *package*.
- Normale Attribute sollten immer `private` sein. Zugriffe darauf können über Setter- und Getter-Methoden implementiert werden. Von dieser Regel gibt es natürlich Ausnahmen. Dazu gehören beispielweise sehr einfache Klassen, die nur Datencontainer darstellen, aus einer abgeleiteten Klasse manipulierbare Attribute oder primitive Attribute, die aus anderen Gründen nicht geschützt werden sollen.
- Statische, finale Attribute können `public` sein, weil es sich um Konstanten handelt.

- Die Klasse, die in Form der `main`-Methode den Startpunkt der Anwendung enthält, muss `public` sein.

2.6 Generische Klassen und Methoden

Eine generische Klasse ist eine Klasse, bei der nicht alle Datentypen bereits während der Deklaration der Klasse angegeben werden. Erst durch die Instantiierung werden die Platzhalter konkrete Typen ersetzt. So können sehr einfach Klassen mit einer identischen Funktionalität aber unterschiedlichen Datentypen realisiert werden. Eine Mehrfachimplementierung ist nicht notwendig.

Ziel der generischen Klassen ist es, die Typunsicherheit, die in den alten Implementierungen vorhanden war, zu umgehen. Container-Klassen geben beim Lesen jetzt nicht mehr nur `Object` als Referenz zurück. Stattdessen besitzt die Deklaration des Rückgabewertes die Klasse, aus der der Wert stammt.

2.6.1 Deklaration generischer Klassen

Die Deklaration unterscheidet sich kaum von der einer „normalen“ Klasse. Zusätzlich werden Platzhalter für die später konkreten Datentypen angegeben.

Beispiel einer fiktiven Stackklasse mit `T` als Platzhalter. Der Platzhalter wird dann im Einsatz durch den Datentyp ersetzt, dessen Werte auf diesem Stack abgelegt werden sollen:

```
public class MyStack<T> {
  private T[] values;

  public T pop() {
    // ...
  }

  public void push(T element) {
    // ...
  }
  // ...
}
```

Es dürfen beliebig viele Platzhalter, durch Kommata getrennt, verwendet werden. Die wichtigste Einschränkung für generische Datentypen besteht darin, dass der Platzhalter immer durch eine Klasse ersetzt werden muss. Primitive Datentypen wie in C++ sind nicht erlaubt.
Ein in C++ vorhandenes Problem wurde in Java gelöst. Es ist möglich festzulegen, dass der Platzhalter nur durch Klassen ersetzt werden darf, die von bestimmten Basisklassen erben. Damit lassen sich Methoden der Basisklasse auf dem Platzhalter aufrufen.

Beispiel, in dem der konkrete Datentyp von der Basisklasse `Point2D` aus dem Paket `java.awt.geom` erbt:

```
public class MyStack<T extends Point2D> {
  private T[] values;
```

```
  public void dump() {
    T val = pop();
    // Aufruf von Methoden der Klasse Point2D
    System.out.println(val.getX() + ", " + val.getY());
  }
  // ...
}
```

2.6.2 Einsatz generischer Klassen

Generische Klassen lassen sich fast genauso wie „normale“ Klassen einsetzen. Zusätzlich ist nur noch der konkrete Typ für den Platzhalter anzugeben.

Beispiel, in dem die Klasse `Point` *(Paket* `java.awt`*) verwendet wird:*

```
MyStack<Point> ms = new MyStack<Point>();
Point p = ms.pop();
```

Das obige Beispiel ist so ausführlich gar nicht erforderlich, da der Compiler hier in der Lage wäre, den Typ auf der rechten Seite der Zuweisung aus dem Kontext (der linken Seite) abzuleiten. Der Diamond-Operator, der eigentlich gar kein echter Operator ist, erlaubt eine kompaktere Darstellung im Quelltext:

```
MyStack<Point> ms = new MyStack<>();
Point p = ms.pop();
```

Der Compiler kann in diesem Fall den Typ der Referenz verwenden. Das klapps natürlich nur, wenn die Referenz von exakt dem selben Typ wie das erzeugte Objekt ist. Sobald die Referenz einen Basisklassentyp beinhaltet, lässt sich der Diamon-Operator nicht mehr verwenden.

2.6.3 Wildcards

Im Zusammenhang mit den neuen generischen Implementierungen der Datenstrukturen (z.B. Collection-Klassen aus dem Paket `java.util`) ergibt sich ein Problem, das anhand des folgenden Codefragments beschrieben wird.
Von der Basisklasse `Vehicle` erben die Klassen `Car` und `Bicycle`.

```
public class Vehicle {
  public void drive() {
    // ..
  }
}

public class Car extends Vehicle {
  public void drive() {
    // ..
  }
}
```

Die generischen Collection-Klassen `LinkedList` und `ArrayList` usw. implementieren die Schnittstelle `Collection`.

Im folgenden Code-Beispiel wird eine `LinkedList` deklariert, die sowohl Objekte der Klasse `Vehicle` als auch Objekte von abgeleiteten Klassen aufnehmen kann.

```
LinkedList<Vehicle> list = new LinkedList<>();
// Objekte erzeugen und in die Liste eintragen.
```

Das Problem tritt auf, wenn versucht werden soll, eine Methode zu schreiben, die beliebige Collection-Objekte mit `Vehicle` als generischen Typ erwartet.

Beispiel:

```
public class VehicleFleet {
  public void drive(Collection<Vehicle> vehicles) {
    // ..
  }
}
```

Wird diese Methode mit der oben genannten Liste `list` aufgerufen, dann meldet der Compiler einen Fehler: Die generische Schnittstelle `Collection<Vehicle>` ist kein Basisdatentyp der Klasse `LinkedList<Vehicle>`. Die Lösung besteht aus der Verwendung eines Platzhalters für `Vehicle`.

```
public class VehicleFleet {
  public void drive(Collection<? extends Vehicle> vehicles) {
    // ..
  }
}
```

Mit diesem „Upper bounded Wildcard“ kann die Methode `drive` mit allen Collections aufgerufen werden, deren Elemente von der Fahrzeugklasse oder einer ihrer abgeleiteten Klasse stammen. In der Methode können alle Methoden aufgerufen werden, die in der Klasse `Vehicle` deklariert sind. Dieser Wildcardtyp gibt also eine Klasse als Obergrenze vor. Damit ist als Parameter die Klasse selbst erlaubt, sowie alle Klassen, die davon erben.
Sollen Collections mit beliebigen Typen zugelassen werden, so kann der Wildcard auch in der einfacheren Form `?` verwendet werden.
Es existiert eine weitere Form des Wildcards. Soll beispielsweise eine Methode geschrieben werden, in der sich zur Collection auch Autos oder Objekte der Basisklassen hinzufügen lassen, aber keine Objekte der abgeleiteten Klassen, so kann das mit dem „Lower Bound Wildcard“ erreicht werden. Dieser gibt eine Klasse als Untegrenze vor. Erlaubt ist die Klasse selbst sowie deren Basisklassen.

```
public class VehicleFleet {
  public void buy(Collection<? super Car> vehicles) {
    vehicles.add(new Car());
  }
}
```

Wichtig: Bei Angabe eines „Lower Bound Wildcards“ können auf der Collection-Klasse keine Methoden aufgerufen werden, die den generischen Typ als Ergebnis zurückgeben, da der Compiler nicht typsicher bestimmen kann, zu welcher Klasse das Ergebnis gehört. Als Konsequenz erzeugt der Compiler einen Fehler.

2.6.4 Generische Methoden

Neben Klassen können auch einzelne Methoden einer Klasse als „generisch“ gekennzeichnet werden. Dieses ist insbesondere bei statischen Hilfsmethoden interessant.

Beispiel:

```
public class StackUtilities {
  public static <T> int compare(Stack<T> s1, Stack<T> s2) {
    // ...
  }
}
```

2.7 Vererbung

Java unterstützt die Vererbung von genau einer direkten Basisklasse, um deren Funktionalität einzuschränken oder zu erweitern. Alle Klassen haben als eine gemeinsame Basisklasse `Object`. Dem Schlüsselwort `extends` folgt der Name der Basisklasse. Im Gegensatz zu C++ bietet Java keine Mehrfachvererbung.

Beispiel (`Account` ist die Basis-, `GiroAccount` die abgeleitete Klasse):

```
public class GiroAccount extends Account {
  // ...
}
```

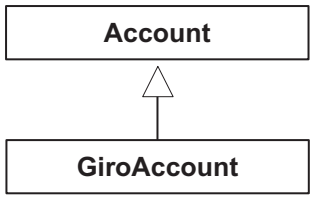

Abbildung 2.13: UML-Darstellung des Beispiels (ohne Attribute und Methoden)

2.7.1 Konstruktor und Initialisierungsreihenfolge

Beim Erzeugen eines Objektes wird erst die Basisklasse initialisiert, danach die abgeleitete Klasse. Die Initialisierungsreihenfolgen sind als Kommentar im folgenden Beispiel angegeben.

Beispiel, Basisklasse:

```
public class Account {
  private int accountNumber = 0;    // 1.

  public Account(int nNo) {         // 2.
    /* ... */
  }
  // ...
}
```

Beispiel, abgeleitete Klasse:

```
public class GiroAccount extends Account {
  private int debit = 0;           // 3.

  public GiroAccount(int nNo) {    // 4.
    /* ... */
  }
  // ...
}
```

Benötigt der Konstruktor der Basisklasse Parameter, so müssen diese ihm vom Konstruktor der abgeleiteten Klasse übergeben werden. Der Aufruf des Basisklassenkonstruktors muss die erste Anweisung im Konstruktor der abgeleiteten Klasse sein.

Beispiel (Konstruktor der abgeleiteten Klasse):

```
public GiroAccount(int nNo) {
  super(nNo);
}
```

Soll in der Basisklasse der Standardkonstruktor aufgerufen werden, so muss der Aufruf nicht explizit erfolgen.

2.7.2 Destruktor (finalize) und Freigabereihenfolge

Der Destruktor (genauer: die `finalize`-Methode) der abgeleiteten Klasse wird automatisch aufgerufen (siehe auch Abschnitt 2.5.4). Die abgeleitete Klasse muss die `finalize`-Methode der Basisklasse explizit aufrufen.

Beispiel für die `finalize`-Methode einer abgeleitete Klasse:

```
public class GiroAccount extends Account {
  // ...
  protected void finalize(){
    super.finalize();
    /* ... */
  }
}
```

2.7.3 Überschreiben von Methoden

Java unterstützt das Überschreiben von Methoden. Dazu müssen in der Basisklasse und der abgeleiteten Klasse zwei Methoden mit demselben Namen und identischen Übergabeparametern existieren.

Die Abbildung 2.14 zeigt die UML-Darstellung für Vererbung. Dort erbt die Klasse `GiroAccount` von der Klasse `Account`.

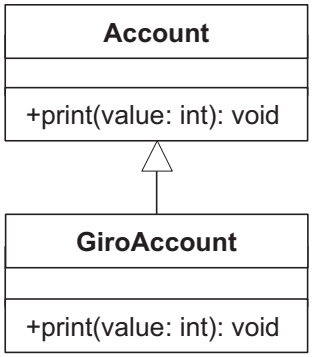

Abbildung 2.14: Überschreiben einer Methode als UML-Diagramm

Es gelten die folgenden Bedingungen:

- Die Methode der Basisklasse darf nicht `final`, `static` oder `private` sein. Dieses gilt auch für die Methode der abgeleiteten Klasse, wenn diese Klasse selbst wiederum als Basisklasse für andere Klassen dienen soll.
- Die Methode der abgeleiteten Klasse darf die Rechte der Basisklassenmethode nicht einschränken. Wenn z.B. die Basisklassenmethoden als `protected` deklariert wurde, dann muss die überschreibende Methode eine der beiden Rechteangaben `protected` oder `public` besitzen. Die Rechtevergabe als `private` ist nicht möglich, da diese strenger als `protected` ist.
- Die Methode der Basisklasse muss mindestens dieselben Ausnahmen auslösen können wie die abgeleitete Klasse. Das folgende Beispiel verdeutlicht den Zusammenhang.

  Beispiel, Basisklasse

  ```
  public class Account {
    // ...
    public void calcInterest() throws ex1, ex2{
      // ...
    }
  }
  ```

  Beispiel, abgeleitete Klasse

  ```
  public class GiroAccount extends Account {
    // ...
    public void calcInterest() throws ex1 {
      // ...
    }
  }
  ```

- *Die überschreibende Methode der abgeleiteten Klasse kann die überschriebene Methode der Basisklasse aufrufen, indem sie `super` vor den Methodennamen setzt.*

  Beispiel, abgeleitete Klasse

  ```
  public class GiroAccount extends Account {
    // ...
    public void calcInterest() throws ex1 {
      super.calcInterest();
      // ...
    }
  }
  ```

Die folgende Tabelle fasst die Rechtevergabe noch einmal zusammen.

Tabelle 2.6: Methodenrechte beim Überschreiben

Methodenrecht Basisklasse	Erlaubtes Methodenrecht abgeleitete Klasse
`protected`	`protected`, `public`
keine Angabe	*keine Angabe*, `protected`, `public`
`public`	`public`

Im Gegensatz zu C++ ist es sehr gefährlich, eine überschriebene Methode aus einem Konstruktor heraus aufzurufen. Das Problem besteht darin, dass die überschreibende Methode in einer abgeleiteten Klasse verwendet wird, ohne dass die Attribute dieser Klasse initialisiert wurden. Die Initialisierung der Attribute der abgeleiteten Klasse erfolgt erst nach dem Konstruktoraufruf der Basisklasse. Deshalb sollte ein Konstruktor nur private oder finale Methoden aufrufen.
Um generell ein Überschreiben zu verhindern, kann eine Methode als `final` deklariert werden.

Beispiel für eine finale Methode:

```
public final long getCounter(){
  return counter;
}
```

2.7.4 Abstrakte Klassen

Ist eine Klasse abstrakt, so können von dieser Klasse keine Objekte erzeugt werden. Abstrakte Klassen dienen nur als Basisklassen. Eine Klasse ist abstrakt, wenn sie als `abstract` gekennzeichnet ist.
`Beispiel`

```
public abstract class Account {
  // ...
  public void calcInterest(){ /* ... */ }
}
```

- Die Klasse kann als normale Basisklasse verwendet werden.
- Eine abstrakte Klasse kann auch abstrakte Methoden (ohne Implementierung) besitzen.

  Beispiel (abstrakte Klasse mit abstrakter Methode):

  ```
  public abstract class Account {
    // ...
    public abstract void calcInterest();
  }
  ```

Die abstrakte Methode wird in einer abgeleiteten Klasse überschrieben und somit implementiert.

2.7.5 Schnittstellen

Deklaration und Eigenschaften

Eine besondere Form einer abstrakten Klasse ist eine Schnittstelle. Diese hatte bis einschließlich Java 7 ausschließlich abstrakte Methoden und statische Attribute. Erst mit Java 8 kamen sogenannte `default`-Methoden hinzu, die Implementierungen besitzen. Sie werden weiter unten beschrieben.
Das folgende Beispiel stammt aus dem Paket `java.lang`. Alle Klassen, deren Objekte mit einer Ordnung versehen werden, sollten die generische Schnittstelle `Comparable` implementieren. Damit können Objekte einer Klasse z.B. in aufsteigender Reihenfolge sortiert werden. Viele Klassen des JDK verwenden `Comparable`.

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

Alle Attribute von Schnittstellen sind `final` und `static`. Fehlen diese Schlüsselwörter, so werden sie implizit angenommen.

Beispiel (mit Angabe):

```
public interface Weigth {
  public static final int length = 23;
}
```

Beispiel (ohne Angabe):

```
public interface Weigth {
  int length = 23;  // genau wie oben
}
```

Beide Varianten sind zulässig und erzielen dasselbe Ergebnis.

Verwendung

Klassen können Schnittstellen verwenden und sie somit implementieren. Dabei gilt:

- Die Syntax ist der der Vererbung ähnlich. Es wird allerdings statt `extends` das Schlüsselwort `implements` verwendet.
- Eine Klasse darf beliebig viele Schnittstellen implementieren. Die Schnittstellen werden, mit Kommata getrennt, nacheinander angeführt.
- Erbt eine Klasse von einer Basisklasse, so darf sie trotzdem zusätzliche Schnittstellen implementieren.
- Werden nicht alle Methoden implementiert, so muss die Klasse als abstrakt gekennzeichnet werden.

  Beispiel (Klasse, die eine Schnittstelle implementiert):

  ```
  public class GiroAccount extends Account
                        implements Comparable<Account>{
    // ...
    public void calcInterest(){
      /* ... */
    }
  ```

```
  // Implementierte Methode der Schnittstelle:
  // Vergleich der Kontonummern
  public int compareTo(Account account) {
    return number - account.number;
  }
}
```

Die zu implementierende Methode `compareTo` liefert die folgenden Ergebnisse:

- < 0 : Das eigene Objekt ist kleiner als das Parameterobjekt `account`.
- $= 0$: Das eigene Objekt ist gleich dem Parameterobjekt `account`.
- > 0: Das eigene Objekt ist größer als das Parameterobjekt `account`.

Damit lassen sich jetzt beispielsweise Sortieralgorithmen schreiben, die als Argument ein Objekt einer Klasse erwarten, die `Comparable` implementiert. Der Algorithmus wird unabhängig von der Klasse der zu sortierenden Daten!

- Schnittstellen können voneinander erben:
  Beispiel:

```
public interface MyInterface {
  /* ... */
}
```

Vererbung:

```
public interface MyInterface2 extends MyInterface {
  /* ... */
}
```

default-Methoden

Ein Problem trat beim Einsatz von Schnittstellen immer wieder auf: Werden zu vorhandenen Schnittstellen neue abstrakte Methoden hinzugefügt, dann müssen alle Klassen, die diese Schnittstellen implementieren, angepasst werden, um die Methoden zu implementieren. Im Rahmen der starken Erweiterung der Datenstruktur-Klassen („Collections“-Klassen) wurden `default`-Methoden eingeführt. So dürfen Schnittstellen die Implementierungen von Methoden vorgeben. Beispiel (unvollständig):

```
public interface Interator<E> {
  boolean hasNext();
  E next();
  default void remove() {
    throw new UnsupportedOperationException();
  }
}
```

`default`-Methoden sollten gezielt dazu verwendet, existierende Schnittstellen zu erweitern.

2.7.6 Oberste Basisklasse Object

`Object` stellt, wie bereits erwähnt, die oberste Basisklasse aller Java-Klassen dar. Sie beinhaltet eine Anzahl Methoden mit Basisimplementierungen, die in abgeleiteten Klassen überschrieben werden können. Die folgende Aufstellung ist unvollständig:

- `protected Object clone()`: Diese Methode kann überschrieben werden, um eine (tiefe) Kopie eines Objektes zu erzeugen.
- `public boolean equals(Object obj)`: Diese Methode muss überschrieben werden, um einen inhaltlichen Vergleich des eigenen Objektes mit dem übergebenen `obj` zu ermöglichen. Diese Methode wird von vielen Container-Klassen des JDK verwendet.
- `protected void finalize()`: Der Garbage-Collector ruft die Methode auf, bevor er den Speicher des Objektes freigibt (siehe Abschnitte 2.5.4 und 2.7.2).
- `public int hashCode()`: Im JDK verwenden alle Hashtabellenklassen diese Methode, um den Hashwert eines Objektes zu ermitteln. Ein Objekt berechnet also seinen eigenen Hashwert.
- `public String toString()`: Die Methode liefert eine Zeichenkettendarstellung des Objektes.

Eine Kurzdarstellung der Methoden `notify`, `notifyAll` und `wait` ist im Abschnitt 3.3.3 über die Synchronisation von Threads zu finden.

2.7.6.1 Klonen von Objekten

Prinzipiell bieten sich mehrere Möglichkeiten, Kopien von Objekten zu erzeugen:

1. In C++ gibt es dazu den so genannten Kopierkonstruktor, der beim Erzeugen eines Objektes eine Referenz auf ein bereits existierendes Objekt übergeben bekommt. Aus diesem wird das neue Objekt als Kopie initialisiert. Dieser Mechanismus kann auch in Java nachgebaut werden (siehe z.B. Klasse `java.awt.Rectangle`), ist aber eher unüblich.
2. Der zweite Weg besteht darin, eine spezielle Methode eines Objektes aufzurufen. Diese Methode legt eine Kopie des eigenen Objektes an und gibt eine Referenz darauf zurück. In Java wird überwiegend dieser Ansatz verwendet. Die Methode heißt `clone` und ist in der Basisklasse `Object` bereits implementiert. Für eigene Zwecke kann die Methode überschrieben werden.

Beispiel zur Verwendung der `clone`-Methode:

```
Point p1 = new Point(42, 66);
Point p2 = (Point) p1.clone();
```

`p2` ist eine Kopie von `p1`. Der cast-Operator ist hier notwendig, weil die `clone`-Methode in der Regel eine Referenz auf `Object` zurück gibt.

Art der Kopie

Prinzipiell lassen sich zwei Arten von Kopien eines Objektes erzeugen:

- Flache Kopie: Enthält ein Objekt Attribute mit Referenzen auf andere Objekte, so werden im Falle einer flachen Kopie nicht die Ziele der Referenzen kopiert. Die Kopie sowie das Original besitzen demnach beide Referenzen auf dieselben Zielobjekte. Ist diese Art der Kopie gewünscht, so kann die automatische Erstellung durch die Basisklasse verwendet werden (siehe unten).

- Tiefe Kopie: Hier werden nicht einfach die Attribute mit Referenzen kopiert. Statt dessen werden auch Kopien der Zielobjekte angelegt. Auf diese Art und Weise teilen sich Original und Kopie nicht mehr andere Objekte, indem sie diese gemeinsam referenzieren. Soll eine tiefe Kopie erstellt werden, so muss manuell kopiert werden (siehe unten).

Natürlich sind auch Mischformen möglich, in der einige Zielobjekte gemeinsam referenziert werden.

Automatische Erstellung einer flachen Kopie

Die automatische Erstellung einer Kopie ist immer dann hilfreich, wenn lediglich eine flache Kopie erstellt werden soll. In diesem Fall übernimmt das Laufzeitsystem von Java fast die komplette Arbeit.

`clone` wird nicht automatisch unterstützt:

```
Object o = new Object();
o.clone();      // Fehler: clone ist protected in Object
```

In einer eigenen Klasse muss die `clone`-Methode der Basisklasse `Object` am besten öffentlich (`public`) überschrieben werden. Weiterhin muss die leere Schnittstelle `Cloneable` implementiert werden. Jetzt kann die eigene `clone`-Methode die Methode der Basisklasse `Object` aufrufen. Deren Implementierung erstellt eine direkte Kopie aller Attribute, sowohl der eigenen als auch der der Basisklasse. Wird die `clone`-Methode nicht überschrieben oder fehlt die Schnittstelle `Cloneable`, so wird eine Ausnahme vom Typ `CloneNotSupportedException` ausgelöst.

Beispiel einer Kreis-Klasse:

```
public class Circle implements Cloneable {
  private Point position;
  private int   radius;

  // Weitere Konstruktoren und Methoden

  // clone-Methode implementieren
  public Object clone() throws CloneNotSupportedException {
    return super.clone(); // ruft die Methode aus Object auf
  }
}
```

Die `clone`-Methode aus `Object` kopiert alle Attribute aus `Circle`. Demnach würden sich in diesem Fall der Originalkreis und die Kopie die Position teilen, da beide eine Referenz auf dasselbe `Point`-Objekt haben. Nur vom Radius als primitiven Datentyp hat jeder eine eigene Kopie. Wie das unerwünschte Verhalten bei der Position umgangen werden kann, folgt im kommenden Abschnitt.

Verwendung der `clone`-Methode:

```
Circle original = new Circle(12, 13, 55);
Circle copy     = (Circle) original.clone();
```

Der cast-Operator ist etwas störend. Seit Java 5 darf die `clone`-Methode auch eine Referenz auf den Typ der eigenen Klasse als Ergebnis haben (kovarianter Rückgabetyp).

Damit würde die `clone`-Methode der fiktiven Kreis-Klasse so aussehen:

```
public Circle clone() { // Circle als return-Typ
  return super.clone(); // ruft die Methode aus Object auf
}
```

Verwendung der modifizierten `clone`-Methode:

```
Circle original = new Circle(12, 13, 55);
Circle copy     = original.clone(); // kein cast erforderlich
```

Soll das Klonen verhindert werden, so reicht es aus, entweder die Methode der Basisklasse nicht zu überschreiben oder in der überschreibenden Methode die Ausnahme `CloneNotSupportedException` auszulösen.
Erbt eine eigene Klasse von einer anderen Klasse, die die `clone`-Methode schon implementiert hat, indem sie `super.clone()` aufruft, so werden automatisch auch alle Attribute der eigenen Klasse kopiert.

Manuelle Erstellung einer tiefen Kopie

Ist eine flache Kopie durch den Laufzeitmechanismus nicht ausreichend, so muss die `clone`-Methode manuell die Ziele aller Referenzattribute kopieren, um eine tiefe Kopie zu erzeugen. Vorgehensweise:

- Durch Aufruf der Basisklassenmethode `clone` wird ein neues Objekt der korrekten Klasse angelegt. Weiterhin werden alle Attribute als flache Kopie übertragen.
- Danach werden manuell die Zielobjekte der Referenzen geklont. Die Verwendung von Konstruktoren sollte in der `clone`-Methode unbedingt vermieden werden. Damit ist gemeint, dass im folgenden Beispiel weder ein neuer Kreis mit `new Circle()` noch eine Kopie der Position mit `new Position(position)` im Stile eines Kopierkonstruktors erstellt werden.

Die folgende Implementierung der `clone`-Methode der fiktiven Kreis-Klasse legt auch eine Kopie des Positions-Objektes an:

```
public class Circle implements Cloneable {
  private Point position;
  private int   radius;

  // Weitere Konstruktoren und Methoden

  // clone-Methode implementieren
  // Der Radius wird automatisch kopiert.
  // Die Referenz auf die Position wird überschrieben.
  public Circle clone() throws CloneNotSupportedException {
    Circle copy = (Circle) super.clone();
    copy.position = (Position) positon.clone();
    return copy;
  }
}
```

Der Vorteil der Implementierung einer tiefen Kopie liegt in diesem Beispiel darin, dass sich beide Kreise nicht mehr eine Position teilen (müssen). Wird jetzt bei einem Kreis das Positionsobjekt verändert, so beeinflusst das den anderen nicht.
http://www.javaworld.com/javaworld/jw-01-1999/jw-01-object.html
betrachtet das Klonen mit seinen Einschränkungen genauer. Hier wird auch begründet,

warum der Aufruf von Konstruktoren in der `clone`-Methode unbedingt vermieden werden soll, bzw. unter welchen Bedingungen er erlaubt ist.

Identität von Objekten

Zwei Objekte sind dann identisch, wenn die Referenzen auf sie identisch sind.

Vergleich auf Identität:

```
Circle c1 = new Circle(12, 13, 14);
Circle c2 = new Circle(12, 13, 14);
Circle c3 = c1;
if (c1 == c2) // false -> Referenzen auf
              // unterschiedliche Objekte
if (c1 == c3) // true -> Referenzen auf
              // dasselbe Objekt
```

Der Vergleich beinhaltet eine kleine Falle bei Zeichenketten: In C++ ist der inhaltliche Vergleich zweier Zeichenketten durch Überladen des ==-Operators möglich. In Java dagegen werden Strings, die ja Objekte sind, bei Verwendung des ==-Operators lediglich auf Identität verglichen.

Vergleich von Zeichenketten:

```
String s1 = "Hallo";
String s2 = "Hallo";
String s3 = "Ha";
s3 += "llo";
String s4 = s1;
if (s1 == s2) // true: Der Compiler legt identische
              // String-Objekte zu einem zusammen, da
              // Strings in Java unveränderlich sind
if (s1 == s3) // false: s3 wird zur laufzeit zusammen-
              // gebaut. Es ist nicht dasselbe Objekt
              // wie s1. Der Operator vergleicht nur
              // auf Identität, die inhaltliche Gleichheit.
if (s1 == s4) // true: Zwei Referenzen (s1 und s4)
              // verweisen auf dasselbe String-Objekt.
```

Fazit: Sollen die Inhalte zweier Zeichenketten verglichen werden, dann sollte der ==-Operator gemieden werden. In diesem Fall funktioniert er nur dann, wenn die Zeichenketten bereits zur Übersetzungszeit feststehen.

Vergleichen von Objekten

Im Gegensatz zum ==-Operator stellt die Methode `equals` fest, ob zwei Objekte inhaltsgleich sind. Die Methode ist in der Basisklasse `Object` bereits implementiert. Da diese Methode aber keine Aussagen über die Gleichheit von Objekten abgeleiteter Klassen treffen kann, vergleicht sie lediglich die Referenzen:

```
public boolean equals(Object other) {
  return this == other;
}
```

Dieses Verhalten ist aber in der Regel nicht hilfreich, weil der Inhaltsvergleich fehlt. So sollte also für jede eigene Klasse die Methode `equals` überschrieben werden. Am Beispiel der `String`-Klasse ist das Verhalten schön zu erkennen:

```
String s1 = "Hallo";
String s2 = "Ha";
s2 += "llo";
if (s1.equals(s2))
```

Die Klasse `String` hat die `equals`-Methode überschrieben und vergleicht jetzt die eigene Zeichenfolge mit der des übergebenen Objektes.
Die folgenden Implementierungshinweise gelten für eine eigene `equals`-Methode:

1. Die eigene Methode muss sich zwingend an die Signatur der Methode der Basisklasse `Object` halten, um diese Methode zu überschreiben:
   `public boolean equals(Object other).`
2. Wird als Parameter `null` übergeben, so ist das Ergebnis immer `false`.
3. Reflexivität: Für ein Objekt `o` gilt immer `o.equals(o)` ist `true`. Wird also als Parameter eine Referenz auf dasselbe Objekt übergeben (`this`), so kann vereinfacht immer `true` zurück gegeben werden, ohne dass noch inhaltlich verglichen wird.
4. Symmetrie: Das Argument der Methode ist formal zwar vom Typ `Object`, aber dennoch wird nur mit Objekten der eigenen Klasse verglichen. Wird ein Objekt einer abgeleiteten Klasse übergeben, so muss das Ergebnis `false` sein. Hintergrund: Beim Vergleich zweier Objekte `o1` und `o2` muss beim Aufruf der `equals`-Methode eine Symmetrie-Bedingung eingehalten werden: Die Vergleiche `s1.equals(s2)` und `s2.equals(s1)` müssen dasselbe Ergebnis liefern. Das ist aber nur schwer einzuhalten, wenn `s1` und `s2` nicht Objekte derselben Klasse sind.
5. Transitivität: Wenn `o1`, `o2` und `o3` Referenzen auf Objekte derselben Klasse sind, dann gilt immer:
   Wenn `o1.equals(o2)` das Ergebnis `true` hat und `s2.equals(s3)` den Wert `true` liefert, dann muss auch `s1.equals(s3)` den Wert `true` ergeben.
6. Konsistenz: Wenn `o1` und `o2` Referenzen auf zwei Objekte derselben Klasse sind, dass muss das Ergebnis des Vergleichs auf jeden Fall solange unverändert bleiben, solange sich nicht eines der Objekte verändert hat.

Beispielimplementierung anhand der Kreis-Klasse:

```
public class Circle {
  private Point position;
  private int   radius;

  // Weitere Konstruktoren und Methoden

  // equals-Methode implementieren (Bedingung 1)
  public boolean equals(Object other) {
    // null-Referenz (Bedingung 2)
    if (other == null)
      return false;

    // Reflexivität (Bedingung 3)
    if (other == this)
      return true;

    // Symmetrie (Bedingung 4)
    if (!other.getClass().equals(getClass()))
      return false;
```

```
    // Inhaltlicher Vergleich
    return radius == other.radius
           && position.equals(other.position);
  }
}
```

Die Einhaltung der Bedingungen 5 und 6 ergibt sich aus der Implementierung. Beispielsweise ändert sich das Ergebnis des Vergleichs nur dann, wenn eines der Objekte von außen durch Eintragen neuer Werte verändert wird.

Hashwert von Objekten ermitteln

Soll ein Objekt einen Schlüssel in einer Hashtabelle repräsentieren, dann muss die Tabelle in der Lage sein, aus dem Schlüssel den Hashwert zu bestimmen. Dazu wird der Schlüssel selbst aufgefordert, seinen Hashwert zu liefern. Das geschieht, indem die Methode `public int hashCode()` aufgerufen wird. Die Methode ermittelt aus den Attributen des Objektes den Hashwert in Form einer positiven oder negativen Integer-Zahl. `hashCode` ist in der Basisklasse `Object` bereits so implementiert, dass sie einen eindeutigen Wert für alle Objekte ermittelt. In der Regel wird die interne Speicheradresse verwendet. Dieses Kriterium ist häufig als Hashwert für reale Anwendungen ungeeignet. Daher sollte für eigene Klassen, deren Objekte als Schlüssel in Hashtabellen verwendet werden, die Methode überschrieben werden. Dabei ist folgendes zu beachten:

1. Wenn der Vergleich zweier Objekte mit der `equals`-Methode `true` ergibt, dann müssen sie auch denselben Hashwert besitzen.
2. Wenn der Vergleich zweier Objekte mit der `equals`-Methode `false` ergibt, dann müssen sich die Hashwerte nicht zwangsweise unterscheiden. In der Praxis wird die Geschwindigkeit einer Hashtabelle aber durchaus verbessert, wenn sich die Hashwerte in einem solchen Fall unterscheiden.
3. Wird die Methode `hashCode` mehrfach auf demselbe, unveränderten Objekt aufgerufen, dann darf sich der Hashwert nicht ändern. Ansonsten würde das Objekt in der Hashtabelle nicht wiedergefunden werden. Der Hashwert muss aber trotz eventuell identischer Attributwerte eines Objektes nicht bei allen Starts der virtuellen Maschine identisch sein.

Beispielimplementierung anhand der Kreis-Klasse:

```
public class Circle {
  private Point position;
  private int   radius;

  // Weitere Konstruktoren und Methoden

  // hashCode-Methode
  public int hashCode() {
    return radius + position.hashCode();
  }
}
```

Es ist sofort ersichtlich, dass die Bedingungen 1 bis 3 eingehalten werden.
Was passiert, wenn Attribute mit Fließkommazahlen in die Hashwert-Berechnung einbezogen werden sollen? Das ist relativ einfach, da die Wrapper-Klassen `Float` und

`Double` Methoden bereitstellen, die eine Wandlung der Fließkommazahl in eine Ganzzahl anbieten. Die Namen der Methoden lauten `Double.doubleToLongBits()` für `double`-Zahlen bzw. `Float.floatToLongBits()` für `float`-Zahlen. Zu beachten ist, dass diese Methoden für die Zahlen 0.0 und -0.0 unterschiedliche Ergebnisse produzieren.

2.8 Fehlerbehandlung

Java bietet mächtige Konzepte sowohl zum Auffinden von Fehlern während der Entwicklungszeit als auch zur Handhabung von Laufzeitfehlern.

2.8.1 Zusicherungen (Assertions)

Zusicherungen werden eingesetzt, um Methodenaufrufe mit falschen oder undefinierten Parametern festzustellen.

Beispiel:

```
public void print(String result) {
  assert result != null : "\"result\" darf nicht null sein";
  System.out.println(result);
}
```

Auf das Schlüsselwort `assert` folgt eine Bedingung. Diese wird zur Laufzeit ausgewertet. Ergibt der Ausdruck `false`, so löst `assert` eine Ausnahme vom Typ `Error` (siehe Abschnitt 2.8.2) aus. Diese sollte nicht abgefangen werden.
Die Meldung, die auf den Doppelpunkt folgt, ist optional. Sie wird im Falle einer Verletzung der Bedingung auf der Konsole ausgegeben.
Laufzeitprüfungen mit `assert` kosten Zeit. Daher müssen sie explizit beim Start des Programms eingeschaltet werden. Dieses geschieht durch Übergabe des Parameters `-ea` an die virtuelle Maschine.

2.8.2 Ausnahmen (Exceptions)

In Java werden Fehler durch sogenannte Ausnahmen (Exceptions) gemeldet. Es handelt sich immer um Objekte von Fehlerklassen.

2.8.2.1 Übersicht

Ablauf der Fehlerbehandlung:

- Eine Methode stellt einen Fehler fest und löst mit der Anweisung `throw` einen Fehler aus. Die Bearbeitung der Methode wird hiermit abgebrochen und zum nächsten passenden `catch`-Block gesprungen.
- Methoden werden innerhalb eines `try`-Blocks aufgerufen. Tritt bei der Bearbeitung kein Fehler auf, so erfolgt der Programmfluss unverändert. Löst die Methode dagegen eine Ausnahme aus, die sie nicht selbst behandelt, so wird der zum `try`-Block passende `catch`-Block aufgerufen um den Fehler zu behandeln. Existiert kein passender `catch`-Block, so wird die aufrufende Methode ebenfalls beendet. Der Vorgang wiederholt sich, bis im Extremfall die Anwendung beendet wird.

- Löst eine Methode einen Fehler aus, den sie nicht selbst behandelt, so muss die Art des Fehlers in der Methodensignatur mit `throws` deklariert werden (siehe Beispiel).
- Oft ist es erforderlich, gewisse Aktionen unabhängig von Erfolg oder Misserfolg eines Aufrufs durchzuführen. Dazu dient der `finally`-Block.

Beispiel:

```
public class Test {
  private void doSomething() throws MyException {
    doWhatever();

    throw new MyException();  // Ausnahme erzeugen


    // Wird nicht mehr erreicht
    doSomethingElse();
  }

  public void do() {
    try {
      doSomething();
    }
    catch (MyException ex) { // Fehler abfangen
      // Fehler behandeln
    }
    finally { // Wird mit und ohne Ausnahme aufgerufen
    }
  }
}
```

Weitere Eigenschaften von Ausnahmen in Java:

- Alle Ausnahmeklassen erben direkt oder indirekt von der Basisklasse `Throwable`. Somit kann im `catch`-Block auch `Throwable` abgefangen werden. Damit behandelt dieser Block alle Fehler.
- Exceptions werden anhand ihrer Klasse unterschieden.
- Es existieren bereits sehr viele vordefinierte Exception-Klassen, um bestimmte Fehler der API-Klassen wiederzugeben.
- Exception-Klassen haben in der Regel Methoden und Attribute, um Beschreibungen des Fehlers aufzunehmen.
- Es können eigene Exception-Klassen geschrieben werden.
- `catch`-Blöcke werden in der Reihenfolge ihres Auftretens im Quelltext untersucht. Somit ist es sehr wichtig, den in der Ableitungshierarchie spezialisiertesten `catch`-Block zuerst aufzuführen.

2.8.2.2 Fehlerkategorien

Es gibt drei Kategorien von Exception-Klassen:

1. Kritische Fehler: Sie erben direkt oder indirekt von der Basisklasse `Error` und müssen in der Methodensignatur nicht deklariert werden.
   Beispiel: `OutOfMemoryError`.

2. Laufzeit-Fehler: Sie treten nur durch fehlerhafte Programme auf und müssen in der Methodensignatur ebenfalls nicht deklariert werden. Sie erben direkt oder indirekt von der Basisklasse `RuntimeException`.
   Beispiel: `NullPointerException`.
3. Normale Ausnahmen: Es handelt sich um Fehler bei Methodenaufrufen, die nicht zur Übersetzungszeit erfasst werden können. Fehlt beispielsweise eine zu öffnende Datei, so wird eine `FileNotFoundException` ausgelöst. Diese Ausnahmen müssen in der Methodensignatur deklariert werden.

   - `void method()`: Die Methode löst keine normale Ausnahme aus.
   - `void method() throws Ex1, Ex2`: Diese Methode kann nur die beiden Ausnahmen `Ex1` und `Ex2` auslösen.

Wenn eine Methode eine Ausnahme auslöst und selbst wieder abfängt, so wird diese Ausnahme nicht im Methodenkopf deklariert.

2.8.2.3 Behandlung mehrerer Fehler

In der Praxis kann es vorkommen, dass in einem Codestück mehrere unterschiedliche Fehler ausgelöst werden, die aber identisch gehandhabt werden sollen. Anstatt jetzt im `catch`-Block die gemeinsame Basisklasse der Fehler anzugeben, lassen sich durch das sogenannte „Multicatch“ mehrere Fehler in einem Block abfangen.

Beispiel für die Multicatch-Anweisung:

```
public class Test {
  private void doSomething() throws MyException1,
                                    MyException2 {
    doWhatever();

    throw new MyException1();  // Ausnahme erzeugen

    // Wird nicht mehr erreicht
    doSomethingElse();
  }

  public void do() {
    try {
      doSomething();
    }
    // Multicatch: beide Fehler abfangen
    catch (MyException1 | MyException2 ex) {
      // Fehler behandeln
    }
    finally { // Wird mit und ohne Ausnahme aufgerufen
    }
  }
}
```

Die Variable `ex` ist in dem Beispiel implizit `final`.

2.8.2.4 Automatische Ressourcen-Verwaltung

Das Kapitel 4 führt in die Ein- und Ausgabebehandlung ein. Auch dort ist eine Fehlerbehandlung erforderlich, weil beispielsweise beim Öffnen einer Datei nicht garantiert ist,

dass diese überhaupt existiert. Das Fehlen der Datei wird als Ausnahme gemeldet. Nach der Verwendung der Datei muss diese wieder geschlossen werden. Auch hierbei können Fehler auftreten. Die automatische Ressourcen-Verwaltung vereinfacht den Umgang mit den Klassen, die die Schnittstelle `AutoCloseable` implementieren. Ein Beispiel soll das verdeutlichen.

Beispiel für die automatische Ressourcen-Verwaltung:

```
public class Test {
  private void doSomething() {
    // Datei zum Lesen öffnen, Ressource in
    // wird beim Verlassen des try-Blockes
    // wieder automatisch geschlossen.
    try {FileReader in  = new FileReader("Eingabe.txt")} {
      while ((c = in.read()) != -1) {
        System.out.println(c);
      }
    }

    catch (IOException ex) {
      System.out.println("Fehler beim Dateizugriff");
    }
  }
}
```

Hier ist sichergestellt, dass die Datei immer geschlossen wird. Außerdem muss der Entwickler den eventuell beim Schließen auftretenden Fehler nicht noch einmal mit einem try-catch-Block behandeln.
`try` unterstützt beliebig viele Ressourcen, die verwaltet werden sollen. Diese werden als normale Anweisungen nacheinander und mit jeweils einem Semikolon voneinander getrennt aufgeführt. Beispiele dazu sind in den Abschnitten 4.4.5.1 und 4.4.5.3 zu finden.

2.8.3 Fehlervermeidung

Dieser Abschnitt enthält eine lose Sammlung von Hinweisen auf Fehler, die leicht vermieden werden können:

- Bei Fließkommazahlen sollte wegen möglicher Darstellungs- und Rechenungenauigkeiten nicht mit dem Gleichheitsoperator `==` gearbeitet werden.
- Zum zeichenweisen Vergleich bei Strings kann nur die Methode `equals()` verwendet werden. Der gerne verwendete Gleichheitsoperator `==` dagegen vergleicht nur die Referenzen im Speicher.
- Ein direkter Zugriff auf Klassenattribute sollte in der Regel nicht zugelassen werden, um die Kapselung der Daten nicht zu zerstören. Hier bietet sich der Einsatz der Getter- und Setter-Methoden an. Generell gilt: Es sollten möglichst wenige Methoden als `public` deklariert werden.
- `int` ist schneller als alle anderen Grunddatentypen.
- Bei Schleifen, die Arrays durchlaufen, sollte immer mit `array.length` oder der neuen Schleifensyntax gearbeitet werden.
- Die kurzen Schleifenvariablennamen `i`, `j`, `k` sind für Integer-Indizes erlaubt. Ansonsten gilt: Aussagekräftige Variablennamen sind zu bevorzugen.

- Die Anweisungen `break` und `continue` sollten möglichst vermieden werden, da sie eine saubere Schleifenstruktur zerstören.
- Defensive Programmierung erlaubt eine frühzeitige Fehlererkennung. Damit ist gemeint, dass ein Entwickler seine Methoden mittels eines Prüfcodes gegen eine falsche Verwendung (z.B. falsche Übergabeparameter) schützt.

2.9 Funktionale Programmierung mit Lambdas

Mit Version 8 zog die funktionale Programmierung in Java ein. Dabei lassen sich funktionale Ausdrücke an Methoden als Parameter übergeben. Diese Ausdrücke werden in Objekte anonymer innerer Klassen, die sogenannte funktionale Schnittstellen implementieren, übersetzt. Das klingt kompliziert, ist es aber eigentlich nicht, was das folgende Beispiel anhand der Schnittstelle `Comparator` zeigt. Diese Schnittstelle dient dazu, zwei Objekte zu vergleichen. Sie wird z.B. dem Sortieralgorithmus in den Datenstrukturklassen übergeben.

```
sort(List<T> arg0,
    Comparator<? Super T> arg1);
}
```

Der `Comparator` besitzt eine an dieser Stelle wichtige Methode:

```
@FunctionalInterface
public interface Comparator<T> {
  int compare(T arg0, T arg1);
  // ...
}
```

Schnittstellen, die genau eine abstrakte Methode besitzen, werden auch als funktionale Schnittstellen bezeichnet. Sie können mit `@FunctionalInterface` annotiert werden, um dem Anwender und Compiler explizit mitzuteilen, dass es sich im eine solche Schnittstelle handelt. Überall, wo funktionale Schnittstellen als Parameter erwartet werden, können die funktionalen Ausdrücke (die „Lambdas“) übergeben werden. Aufruf der Sortiermethode, um eine `ArrayList` mit Kundenobjekten zu sortieren:

```
ArrayList<Customer> cust = ...
// cust befüllen
Collections.sort(cust, (c1, c1) -> c1.getNum() - c2.getNum());
```

Das entspricht dem bis zu Java 7 erforderlichen folgenden Code-Fragment:

```
sort(cust,
     new Comparator<Customer>() {
       @Override
       public int compare(Customer c1,
                          Customer c1) {
         return c1.getNum() -
                c2.getNum();
       }
     });
```

Ein Lambda-Ausdruck hat den folgenden Aufbau:

```
(Parameterliste) -> Ergebnis
```

Die Parameter haben im Beispiel keine Typen, weil der Compiler die Typen aus dem Kontext ermitteln kann. Die `return`-Anweisung kann auf der rechten Seite entfallen,

weil der Wert des Ausdrucks als Ergebnis zurückgegeben wird. Anhand der Signatur der `sort`-Methode erkennt der Compiler, dass es sich um die funktionale Schnittstelle `Comparator` handelt. Diese besitzt, da es sich um eine funktionale Schnittstelle handelt, nur eine abstrakte Methode. Der Lambda-Ausdruck muss syntaktisch auf diese Methoden abgebildet werden können. Er muss also zwei Objekte derselben Klasse wie die Datenstruktur entgegennehmen und einen `int`-Wert zurückgeben.
Ein weiteres Beispiel soll den Einsatz von Lambda-Ausdrücken verdeutlichen. Alle Datenstrukturen, die die Schnittstelle `Collection` implementieren, besitzen die Methode `forEach`. Diese erwartet als Parameter ein Objekt, dessen Klasse die funktionale Schnittstelle `Consumer` implementiert:

```
@FunctionalInterface
public interface Consumer<T> {
  void accept(T);
}
```

Die `forEach`-Methode durchläuft die Datenstruktur und ruft die `accept` auf jedem enthaltenen Objekt auf:

```
cost.forEach(n -> System.out.println(n));
```

Dieser Ausdruck lässt sich noch vereinfachen. Hier wird der Methoden `println` jedes Objekt einmal übergeben. Der Compiler muss eigentlich nur die Methode kennen, die er aufrufen soll. Dazu wurden Referenzen auf Methoden eingeführt:

```
cost.forEach(System.out::println);
```

Auf dem `out`-Objekt wird die `println`-Methode aufgerufen. Der Compiler erkennt, dass die `accept`-Methode ein Objekt übergeben bekommt, das `println` als Parameter erwartet.
Die folgenden Syntaxvarianten sind bei Lambda-Ausdrücken zulässig:

- `(int x, int y) -> x + y`: Typangaben sind nur erforderlich, wenn der Compiler sie nicht selbst ermitteln kann. Bei mehr als einem Parameter sind Klammern links erforderlich.
- `(x, y) -> x + y`: Der Compiler kann die Typen von `x` und `y` selbst bestimmen.
- `x -> 2 * x`: Bei einem Parameter sind die Klammern nicht erforderlich.
- `() -> 42`: Ohne Parameter müssen leere Klammern („burger“) gesetzt werden.
- `System.out::println`: Die Methodenreferenz ist eine Kurzform für den Lambda-Ausdruck `x -> System.out.println(x)`
- `Customer::new`: Referenz auf einen Konstruktur

Lambda-Ausdrücke können recht elegant zur Ereignisbehandlung in grafischen Oberflächen oder in der Stream-API, die mit Java 8 eingeführt wurde, verwendet werden. Der Vorteil gerade in der Stream-API besteht in einer besseren Parallisierbarkeit. Diese API wird hier aber nicht vorgestellt.

2.10 Autoboxing

Java besitzt diverse Containerklassen zur Aufnahme von Objekten beliebiger Klassen (siehe Paket `java.util`). Sollen darin Werte primitiver Datentypen wie `int` gespeichert werden, so entsteht das Problem, dass diese Werte erst in ihren Wrapper-Klassen wie `Integer` gekapselt werden müssen. Ab Version 5 unterstützt Java das sogenannte Autoboxing. Damit werden Werte primitiver Datentypen automatisch in ihre Wrapper-Objekte verpackt und bei Bedarf wieder ausgepackt. Das geschieht komplett transparent ohne Eingriff des Entwicklers.

Beispiel:

```
// Verkette Liste mit Integer-Objekten
LinkedList<Integer> list = new LinkedList<Integer>();
list.add(1);                // 1 wird zu new Integer(1)

int var = list.getFirst(); // Der int-Wert wird mit
                           // Integer.intValue ausgepackt.
```

Der Mechanismus führt aber auch einige Nebeneffekte ein, die teilweise schwer zu erkennen sind:

```
Integer i1 = 127;
Integer i2 = 127;
System.out.println(i1 == 127);  // true
System.out.println(i1 == i2);   // true
Integer i3 = 128;
Integer i4 = 128;
System.out.println(i3 == 128);  // true
System.out.println(i3 == i4);   // false
```

Der letzte Vergleich ergibt `false`, da nur Integer-Objekte im Bereich von -128 bis $+127$ über ihren Inhalt verglichen werden. Beim Autoboxing von Zahlen aus diesem Bereich werden diese aus einem Pool entnommen. Für größere Zahlen dagegen werden neue Objekte erzeugt, deren Referenzen dann auf Gleichheit geprüft werden!

2.11 Metadaten (Annotationen)

Annotationen sind deklarative Beschreibungen, mit denen sich z.B. Klassen und Methoden markieren lassen. Sie haben keinen direkten Einfluss auf die Ausführung des Programms. Stattdessen können Annotationen beispielsweise von externen Programmen ausgewertet werden. Die Deklaration einer Annotation ähnelt der einer Schnittstelle. Das Schlüsselwort ist `@interface`.

Beispiel:

```
public @interface ExtendedInfo {
  String author();
  int    type() default 1;
}
```

Wird die Annotation eingesetzt, so werden die angegebenen Typen `author` und `type` mit konkreten Werten belegt. `type` besitzt einen Standardwert, der nicht angegeben werden muss.

Die Annotation wird nach der Rechteangabe und vor allen anderen Modifikatoren aufgeführt. Alle hier verwendeten Ausdrücke müssen zur Übersetzungszeit bekannt sein.

Beispiel, in dem die Annotation `ExtendedInfo` in der Klasse `Executor` verwendet wird:

```
@ExtendedInfo (
  author = "H. Vogelsang",
  type   = 42
)
public class Executor {
    // ...
}
```

Beispiel einer Annotation ohne Typen (sogenannte Marker-Annotation), die Methoden und Klassen als Testwerkzeuge auszeichnet soll:

```
public @interface Test {
}
```

Beispiel zur Verwendung der Annotation:

```
public class Executor {
  public @Test void showContents() {
    // ...
  }
}
```

Tabelle 2.7: Einige vordefinierte Annotationen

Name	Einsatz	Bedeutung
`@Deprecated`	Methode, Klasse, Interface	Das Element ist veraltet und sollte nicht mehr verwendet werden. Bei Verwendung des markierten Sprachelements erzeugt der Compiler eine Warnung.
`@Override`	Methode	Die Methode überschreibt eine andere Methode. Ist das nicht der Fall, erzeugt der Compiler einen Fehler.
`@SuppressWarnings`	Alle	Der Compiler ignoriert die angegebenen Warnungen in dem Element selbst oder in dessen Unterelementen.
`@Documented`	Annotation	Dieser Metadatentyp soll in die erzeugte JavaDoc-Dokumentation aufgenommen werden.
`@Inherited`	Annotation	Wird diese Annotation in einer Klasse verwendet, so gilt sie auch für abgeleitete Klassen der Klasse.
`@Retention`	Annotation	Speicherungsdauer der Annotation: Laufzeitverfügbarkeit, verfügbar in Klassendateien, verfügbar nur im Quelltext (siehe Klasse `RetentionPolicy`).
`@Target`	Annotation	Gibt an, für welches Ziel (Methode, Klasse, ...) dieser Metatyp verwendet werden darf (siehe auch Klasse `ElementTyp`).

Es sind bereits einige Annotationen vordefiniert. In Java 6 wurden sehr viele weitere Annotationen eingeführt, die hier nicht näher betrachtet werden sollen. Zur programmgesteuerten Auswertung von Annotationen wurden neue Methoden zu den Klassen `Class`, `Method`, `Field` und `Constructor` hinzugefügt.

2.12 Laufzeittypinformationen

Java bietet einen sehr mächtigen Mechanismus, um zur Laufzeit alle Informationen zu einer Klasse eines Objektes herauszufinden. Im einfachsten Fall kann geprüft werden, ob ein Objekt einer bestimmten Klasse entstammt:

Beispiel:

```
Figure fig = new Circle(20, 20);
if (fig instanceof Circle) {
  // Referenz fig verweist auf ein Objekt
  // der Klasse Circle
}
```

Somit kann bei Referenzen sehr leicht das Ziel der Referenz festgestellt werden. Der Operator `instanceof` sollte aber sparsam eingesetzt werden, da er die Prinzipien der sauberen objekt-orientierten Programmierung wieder zerstört. Häufig lässt sich diese Aufgabe durch das Überschreiben von Methoden eleganter lösen.

2.12.1 Weitergehende Informationen

Dazu werden die Klassen aus dem Paket `java.lang.reflect` benötigt. Mit ihrer Hilfe kann beispielsweise zur Laufzeit ermittelt werden, welche Methoden und Attribute eine Klasse besitzt, welche Rechte sie haben und welches die Basisklassen sind. Dazu gibt es zu jeder Klasse ein Objekt der Klasse `Class`, das diese Informationen beinhaltet. Im Softwarelabor wird dieses Wissen nicht benötigt, daher erfolgt an dieser Stelle keine weitere Behandlung.

2.13 Programmierrichtlinien

2.13.1 Namensschemata

Oracle hat für Java sehr enge Richtlinien zum Namensaufbau von Bezeichnern vorgegeben (siehe [ORA03]). Die folgende Tabelle ist eine Zusammenfassung der Web-Seite.

Tabelle 2.8: Regeln zur Namensvergabe

Typ	Regel	Beispiel
Paket	Ein Paketname fängt mit einem kleingeschriebenen Top-Level-Domainnamen wie `com` oder einem Länderkürzel wie `de` an. Darunter angeordnete Pakete erhalten firmen- bzw. organisationsspezifische Bezeichner.	`de.hska.sl`
Klasse, Schnittstelle	Der Bezeichner ist ein Hauptwort aus Groß- und Kleinbuchstaben. Jedes Teilwort des Bezeichners sowie der Bezeichner selbst beginnen mit einem Großbuchstaben.	`MyProxyServer`
Methode	Der Bezeichner ist ein Verb, bestehend aus Groß- und Kleinbuchstaben. Jedes Teilwort des Bezeichners beginnt mit einem Großbuchstaben, der Bezeichner selbst mit einem Kleinbuchstaben.	`getFontName()`
Variable	Der Bezeichner besteht aus Groß- und Kleinbuchstaben. Jedes Teilwort des Bezeichners beginnt mit einem Großbuchstaben, der Bezeichner selbst mit einem Kleinbuchstaben. Die Zeichen _ und `$` sollten vermieden werden.	`int colCounter`

Typ	Regel	Beispiel
Konstante	Eine Mischung aus Großbuchstaben und _ als Trenner einzelner Teilwörter.	`MIN_VALUE`

2.13.2 Aufbau einer Quelltextdatei

Die folgende Reihenfolge sollte immer eingehalten werden:

1. Kommentar (Klassenname, Version, Copyright)
2. Package-Name
3. Import-Anweisungen
4. Deklaration der Klassen (und Schnittstellen) in dieser Reihenfolge:
   (a) Dokumentationskommentar (siehe Abschnitt 2.13.3)
   (b) `class`-Anweisung
   (c) statische Attribute der Klasse in der Reihenfolge `public`, `protected`, `private`
   (d) Attribute in der Reihenfolge `public`, `protected`, `private`
   (e) Konstruktoren
   (f) Methoden, gruppiert nach Funktionalität und nicht nach Zugriffsrechten

Jeder öffentlichen Klasse oder Methode wird ein Dokumentationskommentar vorangestellt.

2.13.3 Kommentierung

Kommentarblöcke beginnen mit `/*` und enden mit `*/`. Es existieren auch Kurzkommentare, die von der Stelle ihrer Verwendung bis zum Zeilenende gelten: `//`. Darüber hinaus besitzt Java sogenannte Dokumentationskommentare, die eine besondere Semantik aufweisen. Sie werden vom Programm `javadoc` ausgewertet, um daraus eine Programmdokumentation (HTML, . . .) zu erzeugen. Aufbau:

- `/**`: Start des Kommentars
- `*/`: Ende des Kommentars

Tabelle 2.9: Einige Schlüsselwörter (Tags) im Kommentar

Klassen, Schnittstellen und Dateien	
`@author`	Name des Autors
`@version`	Version
`@since`	Seit wann vorhanden
`@see`	Verweis auf eine andere Datei oder Klasse, die einen Bezug zu dieser haben (z.B. Basisklasse)

Methoden und Konstruktoren	
`@param`	Name und Funktion eines Parameters
`@return`	Beschreibung des Rückgabewertes
`@exception` `@throws`	Durch die Methode ausgelöste Ausnahmen
`@see`	Verweis auf eine andere Methode (z.B. auf die in der Basisklasse überschriebene Methode)

Beispiel für einen Methodenkommentar:

```
/**
  * Sorts an Array of int-Values.
  * @param  values Unsorted Array.
  * @return Sorted Array.
  * @throws MyException if whatever happens
  * @see    java.util.Arrays.sort(int[] a)
  */
int[] sort (int[] values) throws MyException { /* ... */ }
```

Die Kommentare dürfen HTML-Tags enthalten. Diese werden direkt in den erzeugten HTML-Code eingefügt.

Kommentierung von Paketen

Die Datei `package.html` wird im Quelltext-Verzeichnis des Paketes abgelegt:

- Ihr Inhalt wird mit HTML-Code beschrieben.
- Alles zwischen `<body>` und `</body>` wird als Inhalt interpretiert.
- Der erste Satz im Body wird als Zusammenfassung verwendet.
- Es sollten keine Überschriften verwendet werden.
- Der Inhalt wird von dem Programm `javadoc` verwendet, um eine Paketübersicht zu erstellen.
- Einige der in Paketkommentaren erlaubten Tags sind in Tabelle 2.9 aufgeführt.

2.13.4 Längenbeschränkungen

Der Compiler akzeptiert sehr große Quelltextdateien mit beliebig langen Zeilen. Für den Entwickler werden solche Dateien aber schnell unhandlich. Deshalb empfiehlt Oracle einige Einschränkungen sowohl bezüglich der Quelltextnamen also auch einiger Längen:

- Der Dateiname für Quelltextdateien entspricht dem Namen der öffentlichen Klasse, welche in der Datei enthalten ist.
- Pro Datei ist nur eine öffentliche Klasse erlaubt.
- Quelltextdateien sollten nicht länger als 2000 Zeilen und nicht breiter als 80 Spalten sein.
- Methoden sollten nicht länger als ca. 80 Zeilen sein.
- Es sind nicht mehr als acht Übergabeparameter für Methoden oder Konstruktoren empfohlen.

2.13.5 Einige Formatierungsregeln

Im Folgenden sind einige wichtige Regeln zur Formatierung wichtiger Sprachelemente anhand von Beispielen aufgeführt.

Einfache Anweisungen

Nur eine Anweisung pro Zeile:

```
a++;
b--;
```

Bedingungen

Selbst wenn im Rumpf der Bedingungen nur eine Anweisung enthalten ist, sollte er in einem Block geklammert sein. Die Platzierung der öffnenden und schließenden Klammern ist nicht geregelt, sollte aber einheitlich gehandhabt werden:

```
if (condition) {
  statements;
}

if (condition) {
  statements;
} else {
  statements;
}

if (condition) {
  statements;
} else if (condition) {
  statements;
} else if (condition) {
  statements;
}
```

Kontrollstrukturen

```
for (init; condition; updt) {
  statements;
}

do {
  statements;
} while (condition);

while (condition) {
  statements;
}

switch (condition) {
  case ABC:
       statements;
       /* falls through */
  case DEF:
       statements;
       break;

  case XYZ:
       statements;
       break;

  default:
       statements;
       break;
}
```

2.14 Einbettung der Sprache

Java verfügt über eine ausgezeichnete API-Dokumentation [ORA01]. Daher erfolgt an dieser Stelle nur eine kurze Einführung.

2.14.1 Strings

Java besitzt mehrere sehr leistungsfähige Klassen zur Bearbeitung von Zeichenketten. Diese sind im Gegensatz zu C oder C++ nicht 0-terminitert.

Klasse String

Die Klasse `String` stammt aus dem Paket `java.lang`. Sie verwaltet unveränderliche Zeichenketten. Das heißt, dass durch das „Anhängen“ eines Zeichens an eine Zeichenkette komplett neuer Speicher reserviert wird, in den die Zeichenkette zusammen mit dem anzuhängenden Zeichen kopiert werden. Somit sind Strings nur dann effizient, wenn sie unveränderlich sind. Identische Zeichenketten im Quelltext werden vom Compiler nicht mehrfach abgelegt. Statt dessen verweisen alle Referenzen dann auf dasselbe Objekt. Somit können nur Zeichenketten, die zur Übersetzungszeit existieren, mit Hilfe des Operators == auf Gleichheit überprüft werden.

Beispiel:

```
public class TestString {
  public static void main(String[] args) {
    String test = "Hallo";
    String test2;
    test2 = test;          // test2 verweist auch auf "Hallo"
    test2 += ", Tester";  // Hängt ", Tester" an test2 an
    System.out.println(test2);
  }
}
```

Die Klasse `String` besitzt eine Anzahl Methoden zur Manipulation der Zeichenkette. Einige Methodenbeispiele:

- `int indexOf(int ch)`: Liefert den Index innerhalb der Zeichenkette, an der sich das Zeichen `ch` das erste Mal befindet.
- `int indexOf(int ch, int fromIndex)`: Ermittelt den Index in der Zeichenkette, an der sich das Zeichen `ch` das erste Mal hinter der Position `fromIndex` befindet.
- `boolean equals(Object anObject)`: Test, ob die Zeichenkettenrepräsentation des übergebenen Objekts dem eigenen entspricht. In der Klasse Object existiert eine überschreibbare Methode `String toString()`, die eine Zeichenkettendarstellung eines Objektes zurückgeben kann. Diese Form des Vergleichs von Zeichenketten muss gewählt werden, wenn die Zeichenketten zur Laufzeit verändert oder erzeugt wurden und gleichzeitig zeichenweise verglichen werden sollen.
- `int length()`: Gibt die Länge der Zeichenkette zurück.
- `String substring(int beginIndex, int endIndex)`: Ermittelt die Teilzeichenfolge zwischen `beginIndex` (inklusive) und `endIndex` (exklusive).
- `String toLowerCase()`, `String toUpperCase()`: Konvertiert die Zeichenfolge in Klein- bzw. Großbuchstaben.
- `String trim()`: Entfernt alle Leerzeichen am Ende der Zeichenfolge.
- `static String valueOf(int i)`: Erzeugt ein String-Objekt mit einer textuellen Darstellung des Integer-Wertes `i`. Die Methode existiert auch für alle anderen primitiven Datentypen.

Die genaue Dokumentation der Klasse ist in der API-Dokumentation zu finden. String unterstützt die Operatoren `+` und `+=` zum Verbinden zweier String-Objekte.

Klasse StringBuffer

Die Verwendung von Objekte der Klasse `String` kann sehr ineffizient sein, da ihr Inhalt nicht mehr veränderbar ist. Die Klasse `StringBuffer` dagegen eignet sich hervorragend für sich ändernde Zeichenketten. Die Speicherverwaltung ist in diesem Fall wesentlich effizienter, da Objekte dieser Klasse nicht konstant sind.
`StringBuffer` bietet teilweise nicht so mächtige bzw. komfortable Methoden wie String an. Beispiel:

```
public class TestStringBuffer {
  public static void main(String[] args) {
    StringBuffer test  = new StringBuffer("Hallo");
    StringBuffer test2 = new StringBuffer();
```

```
    test2.append(test.toString()); // Kopiert alle Zeichen
    test2.append(", Tester");      // Hängt ", Tester" an
    System.out.println(test2.toString());
  }
}
```

Die Klasse `StringBuffer` besitzt eine Anzahl Methoden zur Manipulation der Zeichenkette. Einige Methodenbeispiele:

- `StringBuffer append(String str)`: Hängt den Inhalt von `str` an den eigenen Inhalt an.
- `StringBuffer append(int i)`: Hängt die textuelle Darstellung von `i` an den eigenen Inhalt an. Die Methode existiert auch für alle anderen primitiven Datentypen.
- `int length()`: Gibt die Länge der Zeichenkette zurück.
- `StringBuffer insert(int off, String str)`: Fügt den Inhalt des String-Objektes `str` an der Position `off` in die eigene Zeichenkette ein.

Klasse StringBuilder

Die Klasse `StringBuilder` bietet einen ähnlichen Funktionsumfang wie die oben vorgestellte Klasse `StringBuffer`. Allerdings sind alle Methodenzugriffe nicht synchronisiert, was den Zugriff aus nur einem Thread extrem beschleunigt. Im Softwarelabor ist dieses die am besten geeignete Klasse für sich ändernde Zeichenketten.

2.14.2 Arrays

Die Klasse `java.util.Arrays` enthält eine Reihe statischer Methoden zur Manipulation von Arrays. Beispiele exemplarisch für den Datentyp `byte`:

- `int binarySearch(byte[] a, byte key)`: Sucht einen Wert im Array mit Hilfe des Halbierungsverfahrens.
- `boolean equals(byte[] a, byte[] a2)`: Vergleicht beide Arrays anhand ihres Inhaltes.
- `void fill(byte[] a, byte v)`: Füllt das Array mit dem Wert `v`.
- `void fill(byte[] a, int sInd, int eInd, byte v)`: Füllt ein Teil-Array zwischen den Indizes `sInd` (inklusive) bis `eInd` (exklusive) mit dem Wert `v`.
- `void sort(byte[] a)`: Sortiert das Array.
- `void sort(byte[] a, int sInd, int eInd)`: Sortiert den Teil des Arrays zwischen den Indizes `sInd` (inklusive) bis `eInd` (exklusive).

Diese Methoden sind für alle primitiven Datentypen sowie für den Typ `Object` vorhanden. Die API-Dokumentation der Klasse `java.util.Arrays` beinhaltet nährere Informationen dazu. Diese Methoden unterstützen allerdings nur eindimensionale Arrays.

2.14.3 Mathematik

Siehe API-Dokumentation zur Klasse `java.lang.Math`.

3

Fortgeschrittene Java-Themen

Dieses Kapitel beschreibt fortgeschrittene Java-Eigenschaften, die im Rahmen des Softwarelabors und der Klausur „Informatik 2“ **nicht** benötigt werden.

3.1 Schwache Referenzen

Neben den in Abschnitt 2.4 beschriebenen „normalen“ oder auch starken Referenzen existieren noch sogenannte schwache Referenzen.

3.1.1 Schwache Referenzen

Schwache Referenzen verweisen auf Objekte, ohne dass der Garbage-Collector daran gehindert wird, die Objekte bei Bedarf aus dem Speicher zu entfernen. Schwache Referenzen sind Objekte der Klasse `Reference`, die als Attribut eine Referenz auf das eigentliche Objekt besitzt. Die generelle Arbeit mit einer schwachen Referenz sieht so aus:

- Referenz mit dem Zielobjekt erzeugen.
- Beim Zugriff wird das Zielobjekt mit Hilfe der get-Methode ausgelesen. Wenn die Methode `null` liefert, dann wurde das Objekt aus dem Speicher entfernt. Es muss manuell neu erzeugt und in die Referenz eingetragen werden.

Java bietet drei unterschiedliche Typen schwacher Referenzen.

SoftReference

Eine `SoftReference` wird häufig dazu verwendet, um Daten in einem Cache zu halten. Dieses wird dadurch ermöglicht, dass der Garbage-Collector ein Objekt, das über eine solche Referenz angebunden ist, erst dann aus dem Speicher entfernt, wenn ansonsten eine Speicheranforderung nicht mehr befriedigt werden kann.

WeakReference

Eine `WeakReference` wird eingesetzt, um Zusatzinformationen an einem Objekt abzulegen. Ein über eine `WeakReference` erreichbares Objekt wird genau dann vom Garbage-Collector entfernt, wenn es ausschließlich über solche Referenzen erreichbar ist.

PhantomReference

Phantom-Referenzen dienen dazu, festzustellen, ob ein Objekt bereits vom Garbage-Collector entfernt wurde. Die `get`-Methode der Referenz liefert immer `null`. Sie eignet sich also nicht für den normalen Zugriff. Das Verhalten widerspricht somit der eigentlichen Spezifikation schwacher Referenzen. Ablauf:

- Sobald ein Objekt vom Garbage-Collector gelöscht werden soll, wird es für alle Referenzen unerreichbar.
- Wenn noch Phantom-Referenzen auf das Objekt existieren, so wird das Objekt in einer sogenannten `ReferenceQueue` gespeichert.
- Das Programm kann die `ReferenceQueue` durchlaufen und Aufräumarbeiten ausführen.

3.2 Geschachtelte Klassen

Klassen sind ineinander verschachtelbar. Damit kann eine Klasse, die logisch zu einer anderen gehört, innerhalb dieser deklariert werden, um die Zugehörigkeit auszudrücken. Es existieren drei Arten von inneren Klassen.

3.2.1 Nicht statische lokale Klassen

Eine nicht statische lokale Klasse wird innerhalb des Definitionsteils einer anderen Klasse definiert.

Beispiel:

```
public class Outer {
  class Inner {
    private int attr1;
  }
  private int attr2;
}
```

Anmerkungen:

- Das Erzeugen eines Objektes der Klasse `Inner` muss innerhalb der Grenzen der äußeren Klasse `Outer` erfolgen (z.B. in einer der Methoden oder während der Initialisierung).
- Innere Klassen dürfen weder statische Methoden noch statische Attribute besitzen.
- Verdeckt ein Element der inneren Klasse ein Element der äußeren, so kann die innere Klasse z.B. auf eine verdeckte Methode mit `Outer.this.method()` zugreifen.
- Im oben gezeigten Beispiel erzeugt der Compiler aus dem Quelltext die beiden Objektdateien `Outer.class` und `Outer$Inner.class`.
- Zugriffsrechte:
  - Die innere Klasse kann auf Attribute und Methoden der äußeren Klasse zugreifen.
  - Die äußere Klasse kann auf Attribute und Methoden der inneren Klasse zugreifen.
  - Beide Aussagen gelten auch für private Attribute und Methoden.

3.2.2 Anonyme Klassen

Eine anonyme Klasse wird innerhalb einer Methode einer anderen Klasse definiert, wobei gleichzeitig ein Objekt der neuen, anonymen Klasse erzeugt wird.

Beispiel: Die folgende Schnittstelle soll von der anonymen Klasse implementiert werden:

```
public interface Random {
  public int random();
}
```

Beispiel zur Verwendung der Schnittstelle in der öffentlichen Klasse `Test`*:*

```
dump(new Random() {
    public int random() {
           return (int)Math.random() * 100; }
    } );
```

Anmerkungen:

- Es können keine weiteren Objekte der Klasse erzeugt werden.
- Aus dem Beispiel-Quelltext erzeugt der Compiler die drei Dateien
  - `Test.class`,
  - `Random.class` sowie
  - `Test$1.class`

  und vergibt die Nummern automatisch.

Ein häufiges Einsatzgebiet anonymer Klassen ist die Ereignisbehandlung in grafischen Benutzungsoberflächen, wenn der Ereigniscode nicht sehr komplex ist.

3.2.3 Statische lokale Klassen

Eine statische, lokale Klasse wird innerhalb des Definitionsteils einer anderen Klasse definiert.

Beispiel:

```
public class Outer {
  static class Inner {
    private int attr1;
  }
  private int attr2;
}
```

Anmerkungen:

- Die Instanziierung kann aus beliebigen anderen Klassen erfolgen.
- Die innere Klasse ist nicht an die äußere gekoppelt und kann daher nicht auf Attribute und Methoden der äußeren Klasse zugreifen.
- Der Name der inneren Klasse im Quelltext lautet `Outer.Inner`, der Name der erzeugten Datei `Outer$Inner.class`.

3.3 Multithreading

Java besitzt eine sehr mächtige und umfangreiche API zur Erstellung multithreadingfähiger Programme. In Java 5 wurde diese API nochmals erweitert. In den folgenden Abschnitten werden nur die wichtigsten Grundfunktionalitäten vorgestellt. Die Pakete `java.util.concurrent` und deren Unterpakete werden hier aus Platzgründen ebenso wenig betrachtet wie viele andere Eigenschaften von Threads.

3.3.1 Threads erzeugen und starten

Ein Thread ist ein Objekt der Klasse `Thread`. Im Wesentlichen wird ein Thread auf zwei verschiedene Arten erzeugt:

1. Eine Klasse erbt von `Thread` und überschreibt die Methode `run`. Diese implementiert die (Endlos-)Schleife des Threads und wird beim Start aufgerufen.
2. Eine Klasse implementiert die Schnittstelle `Runnable` und überschreibt ebenso die Methode `run`. Ein Objekt der Klasse wird dem Konstruktor von `Thread` übergeben.

Beispiel:

```
public class MyThread extends Thread {
  public void run() {
    while (true) {
      // Implementierung
    }
  }
}
```

Der Thread wird schließlich mit `start` gestartet.

Beispiel:

```
MyThread mThr = new MyThread();
mThr.start();
```

Die Thread-Priorität liegt zwischen `MIN_PRIORITY` (0) und `MAX_PRIORITY` (10). Im Standardfall beträgt sie `NORM_PRIORITY` (5).

3.3.2 Threads beenden und löschen

Die einzige in aktuellen Java-Versionen unterstützte Möglichkeit, einen Thread zu beenden, besteht darin, dass der Thread seine run-Methode selbst verlässt. Ein Problem tritt auf, wenn der Thread durch einen anderen terminiert werden soll. Als Ausweg kann einem Thread ein Unterbrechungssignal gesendet werden. Der ausführende Thread testet zyklisch auf das Vorhandensein des Signals und beendet sich selbst.

Beispiel:

```
public class MyThread extends Thread {
  public void run() {
    while (true) {
      if (interrrupted()) {
        return;   // beenden
      }
      // ...
    }
```

```
  }
}
```

Dem Thread wird mit `interrupt` das Signal zur Unterbrechung geschickt.

Beispiel:

```
MyThread mThr = new MyThread();
mThr.start();
// ...
mThr.interrupt();
```

Ein Thread läuft häufig nicht permanent sondern legt sich für eine gewisse Zeit „schlafen“. Aus diesem Schlafzustand wird er ebenso mit `interrupt` geweckt. Um zwischen einem „normalen“ Aufwecken aus dem Schlafzustand und einem Signal zur Beendigung der Arbeit unterscheiden zu können, lässt sich der Thread sehr einfach durch Hinzufügen einer Boole'schen Variablen erweitern.

Beispiel (Thread):

```
public class MyThread extends Thread {
  private boolean stopped = false;

  public void run() {
    while (!stopped) {
      // Aufwecken aus dem Schlafzustand
      try {
        sleep(2000);
      }
      catch (InterruptedException ex) {
        if (!stopped) {
          // Arbeit nach dem Wecken
        }
      }
      // Arbeiten
      // Test auf Aufwecken während der Arbeit
      if (interrupted() && stopped) {
        return;
      }
      // Arbeiten
    }
  }

  public void stop() {
    stopped = true;
    interrupt();
  }
}
```

Der Aufruf der `stop`-Methode teilt dem Thread mit, dass er sich beenden soll. Dabei ist zu beachten, dass ein aktiver Thread unter Umständen erst nach einer gewissen Zeit das Stopsignal sieht. Es ist daher sinnvoll, die Variable `stopped` in regelmäßigen Abständen zu überprüfen.

3.3.3 Synchronisation von Threads

Zur einfachen Synchronisation von Threads existieren in der Basisklasse aller Klassen `Object` mehrere Methoden. Das Objekt, auf dem die Methoden aufgerufen werden,

dient dabei als Monitor. Alle folgenden Methoden sollten nur innerhalb synchronisierter Blöcke oder Methoden, die den Monitor als Synchronisationsobjekt verwenden, aufgerufen werden. Synchronisierte Methoden und Blöcke werden in den Folgeabschnitten beschrieben.

- `wait`: Beim Aufruf der Methode wird der gerade aktive Thread an diesem Synchronisationsobjekt blockiert. Optional ist die Angabe einer maximalen Wartezeit. Da diese Methode nur innerhalb eines synchronisierten Abschnitts (Block oder Methode) aufgerufen werden soll, wird beim Blockieren der Abschnitt wieder freigegeben. Sobald der Thread deblockiert wird, wird er wieder Eigentümer des Abschnitts und blockiert ihn somit. `wait` löst eine `InterruptedException` aus, wenn der wartende Thread ein Unterbrechungssignal während seines Wartens empfangen hat.
- `notify`: Ein am Monitor wartender Thread wird deblockiert. Im Falle mehrerer an diesem Monitor wartender Threads hängt es von der Implementierung der virtuellen Maschine ab, welcher Thread deblockiert wird.
- `notifyAll`: Alle am Monitor wartenden Threads werden wieder freigegeben. Die Threads bekommen der Reihe nach den synchronisierten Abschnitt zugewiesen, so dass sichergestellt ist, dass sich immer nur ein Thread in dem kritischen Abschnitt befindet.

In Java 5 wurden weitere Klassen zur Synchronisation eingeführt. Diese befinden sich im Paket `java.util.concurrent` und dessen Unterpaketen.

Synchronisierte Methoden

Sie stellen sicher, dass nur ein Thread zu einem Zeitpunkt die Methode betreten kann.

Beispiel für synchronisierte Methoden:

```
public class Timer {
  // ...
  public synchronized void set(long time) {
    /* ... */
  }
  public synchronized void set(Date date) {
    /* ... */
  }
}
```

Befinden sich mehrere Methoden mit dem Merkmal `synchronized` in einer Klasse, so ist sichergestellt, dass immer nur eine dieser Methoden zu einem Zeitpunkt aufgerufen werden kann. Die Methode verwendet das Objekt, auf dem sie aufgerufen werden, als Monitor.

Die `set`-Methode aus dem vorherigen Beispiel entspricht somit:

```
  public void set(long time){
    synchronized (this) {
      /* ... */
  }
```

Solche synchronisierten Blöcke sind im Folgeabschnitt beschrieben.
Wird eine statische Methode als `synchronized` markiert, so wird als Monitor das Klassenobjekt der Klasse verwendet (Klasse `java.lang.Class`). So ist sichergestellt, dass

nur eine synchronisierte, statische Methode der Klasse zu einem Zeitpunkt ausgeführt werden kann,

Synchronisierte Blöcke

Es handelt sich um Monitore, die einen gegenseitigen Ausschluss mehrerer Threads in diesem Block bewirken. Die Synchronisation findet immer an einem Objekt statt, wobei das Objekt als der Monitor agiert.

Beispiel, in dem `object` *eine Referenz auf das Monitorobjekt ist:*

```
synchronized (object) {
  // Kritischer Abschnitt
}
```

4

Ein- und Ausgabebehandlung

Dieses Kapitel beschreibt den Umgang mit externen Datenquellen und -zielen. Dazu gehören auch Dateien. Das Verständnis dieses Kapitels ist teilweise für die Lösung der Aufgaben der Übungen zu „Informatik 2“ erforderlich. Das Kapitel beschränkt sich auf die Grundlagen, soweit sie für die Vorlesung und Übung erforderlich sind. Es existieren im Paket `java.nio` und dessen Unterpaketen teilweise wesentlich mächtigere Klassen.

4.1 Einleitung

Programme müssen häufig Daten von einer externen Quelle lesen oder auf ein externes Ziel schreiben. Probleme:

- Die zu lesende Information kann dabei überall vorkommen: In einer Datei, auf Festplatte, irgendwo im Internet, im Speicher oder in einem anderen Programm. Das gilt auch für das Ziel der Daten.
- Die Information kann von einem beliebigen Typ sein: Objekte, Zeichen, Bilder, Geräusche . . .

Java besitzt für einen einheitlichen Zugriff auf solche Informationen die Stream-Klassen des Pakets `java.io`. Diese erlauben einen einheitlichen, lesenden und schreibenden Zugriff auf Daten mit unterschiedlichen Quellen und Zielen.

- Eingabe: Um Informationen zu lesen, öffnet ein Programm einen Stream zu einer Informationsquelle und liest die Daten sequentiell.

Abbildung 4.1: Daten aus einer Quelle lesen

- Ausgabe: Um Informationen zu schreiben, öffnet ein Programm einen Stream zu einem Informationsziel und schreibt die Daten sequentiell.

Abbildung 4.2: Daten in ein Ziel schreiben

Unabhängig davon, ob gelesen oder geschrieben wird, sind die Algorithmen fast identisch:

Lesen:

```
open a stream
while stream has more data {
  read information
}
close the stream
```

Schreiben:

```
open a stream
while program has more data {
  write information
}
close the stream
```

Die Stream-Klassen bestehen aus zwei Klassen-Hierarchien: Zeichenströme werden für reine textuelle Daten eingesetzt, Byte-Ströme für Binärdaten.

4.2 Zeichen-Ströme

Zeichenströme („Character-Streams“) für rein textuelle Daten verwenden die Klassen, die von `Reader` bzw. `Writer` erben. Diese Klassen sind in der Lage, 16-Bit Unicode-Zeichen zu verwalten. Im Gegensatz dazu können Byte-Ströme nur 8-Bit Elemente des Zeichensatzes ISO-Latin-1 erkennen. Daher verwendet man für textuelle Daten in der Regel die Zeichen-Ströme.
Von `Reader` und `Writer` abgeleitete Klassen implementieren spezialisierte Ströme, die sich in zwei Klassen einteilen lassen:

- Lesen von Datenquellen und Schreiben auf Datenzielen (in den folgenden Diagrammen blau beschriftet)
- Bearbeitung von Daten (in den folgenden Diagrammen grau hinterlegt)

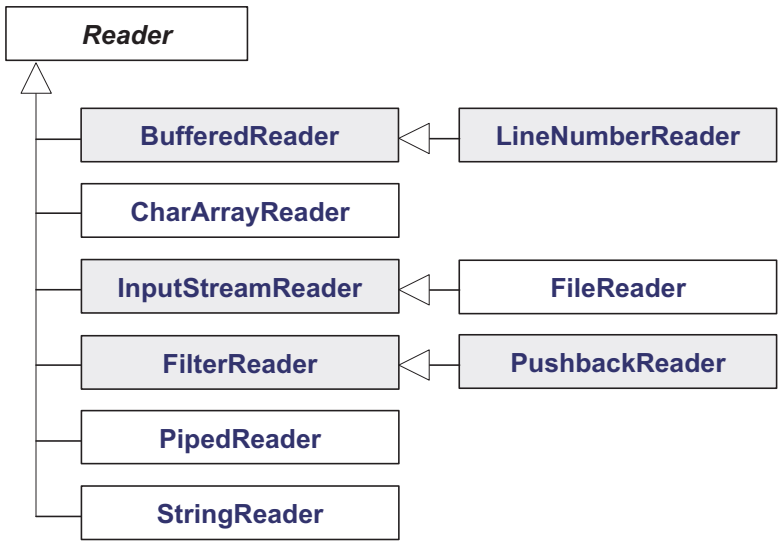

Abbildung 4.3: Reader-Klassen

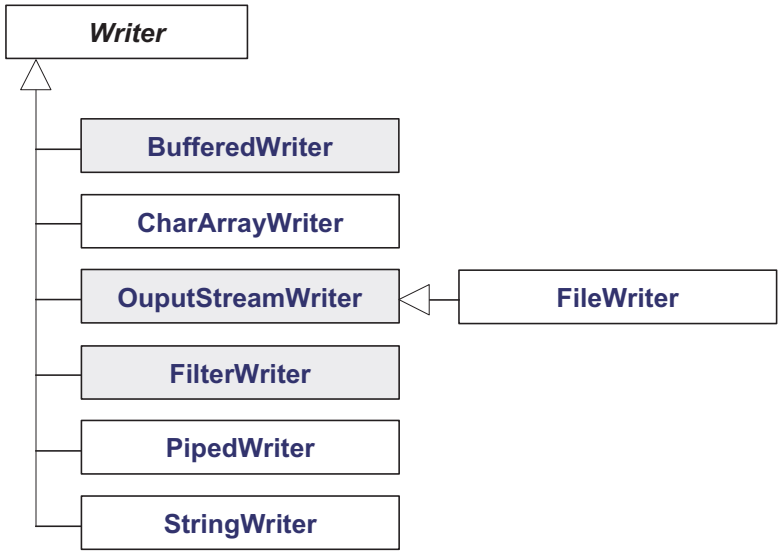

Abbildung 4.4: Writer-Klassen

Die Einsatzgebiete der einzelnen Klassen können in der API-Dokumentation nachgeschlagen werden. Teilweise sind aber auch die Klassennamen selbsterklärend.

4.3 Byte-Ströme

Byte-Ströme werden verwendet, um 8-Bit-Daten zu lesen oder zu schreiben. Dazu gehören auch Binärdaten wie Bilder und Töne. Alle anderen Klassen, die von den abstrakten Basisklassen `InputStream` oder `OutputStream` erben, verarbeiten Byte-Ströme. Davon abgeleitete Klassen implementieren spezialisierte Ströme, die sich in zwei Klassen einteilen lassen:

- Lesen von Datenquellen und Schreiben auf Datenzielen (in den folgenden Diagrammen blau beschriftet)
- Bearbeitung von Daten (in den folgenden Diagrammen grau hinterlegt)

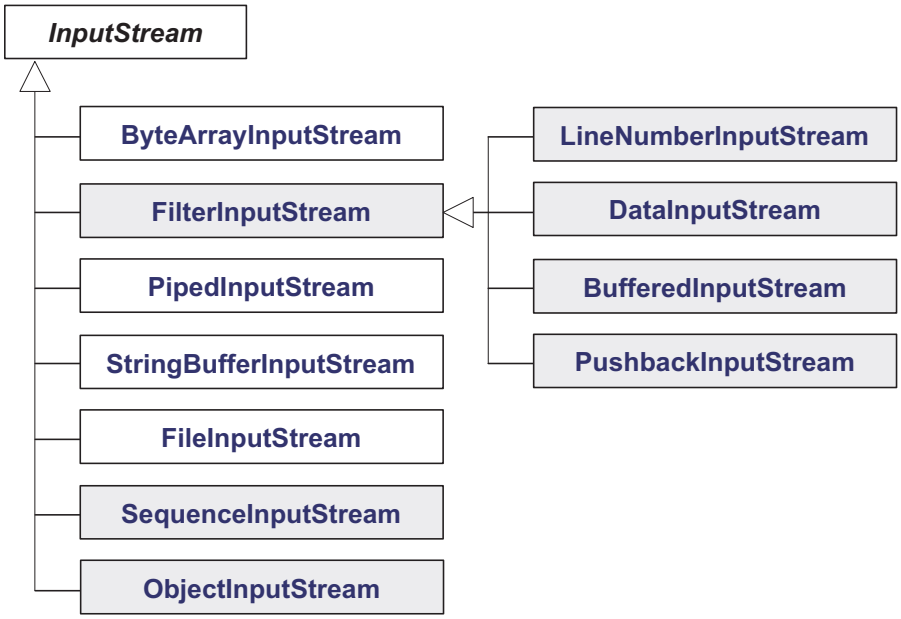

Abbildung 4.5: InputStream-Klassen

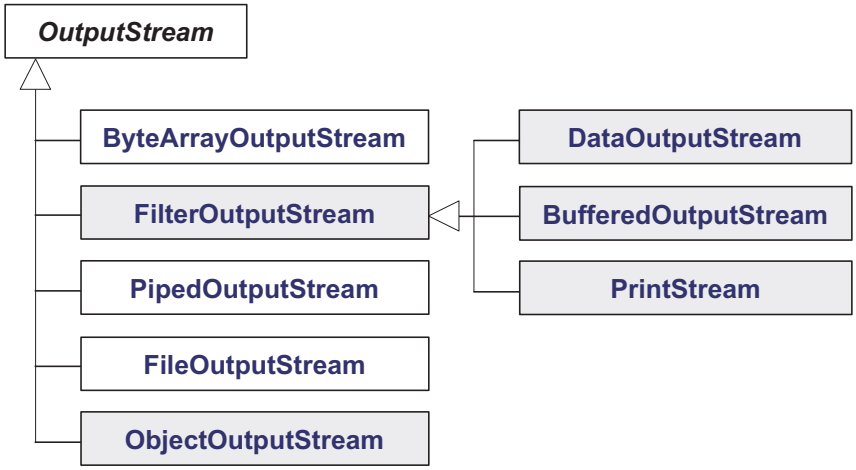

Abbildung 4.6: OutputStream-Klassen

4.4 Verwendung der Klassen

Prinzipiell lassen sich lesender und schreibender Zugriff unterscheiden.

4.4.1 Lesender Zugriff über die Basisklassen

`Reader` und `InputStream` definieren ähnliche Signaturen aber mit unterschiedlichen Datentypen:

- `Reader`:

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
```

- `InputStream`:

```
int read()
int read(byte cbuf[])
int read(byte cbuf[], int offset, int length)
```

`Reader` und `InputStream` besitzen Methoden, um die aktuelle Position im Stream zu markieren, Eingabedaten zu überspringen und die Leseposition zurückzusetzen.

4.4.2 Schreibender Zugriff über die Basisklassen

Die Basisklassen `Writer` und `OutputStream` sind sehr ähnlich aufgebaut:

- `Writer`:

```
void write(int c )
void write(char cbuf[])
void write(char cbuf[], int offset, int length)
```

- `OutputStream`:

```
void write(int c)
void write(byte cbuf[])
void write(byte cbuf[], int offset, int length)
```

4.4.3 Öffnen und Schließen eines Stroms

Alle Streamklassen öffnen den zugehörigen Datenstrom automatisch, wenn ein Objekt angelegt wird (bei Aufruf des Konstruktors). Ein Datenstrom kann explizit geschlossen werden (Methode `close`). Der Garbage-Collector schließt einen Strom automatisch, bevor das Objekt gelöscht wird. Darauf sollte man sich aber nicht verlassen, da dann eine Datei eventuell sehr lange geöffnet bleiben kann.

4.4.4 Prinzip der Datenzugriffe

Die folgenden beiden Abschnitte zeigen den prinzipiellen Datenzugriff über Methoden der Basisklassen.

4.4.4.1 Textdatei zeichenweise kopieren

Dieses Beispiel zeigt, wie der Zugriff auf Textdateien mit Hilfe der Basisklassenmethoden erfolgen kann:

```
public class Copy {
  public static void main(String[] args) throws IOException {
    FileReader in  = new FileReader("Eingabe.txt");
    FileWriter out = new FileWriter("Ausgabe.txt");
    int c;

    while ((c = in.read()) != -1) {
      out.write(c);
    }

    in.close();
    out.close();
  }
}
```

Anmerkungen:

- Das Ende der Datei wird hier beim Lesen dadurch erkannt, dass die `read`-Methode -1 als Ergebnis liefert.
- Da die meisten Dateisysteme auf einer 8-Bit-Darstelleung basieren, wird jedes Zeichen in ein Unicode-Zeichen umgewandelt, was ineffizient ist.
- Das Lesen und Schreiben erfolgt zeichenweise und damit ungepuffert, was ebenfalls ineffizient ist. Ein Beispiel für gepufferten Zugriff ist weiter unten im Text zu finden.

4.4.4.2 Binärdatei byteweise kopieren

Dieses Beispiel zeigt, wie der Zugriff auf Binärdateien mit Hilfe der Basisklassenmethoden erfolgen kann:

```
public class Copy {
  public static void main(String[] args) throws IOException {
    FileInputStream  in  = new FileInputStream("Eingabe.dat");
    FileOutputStream out = new FileOutputStream("Ausgabe.dat");
    int c;

    while ((c = in.read()) != -1) {
      out.write(c);
    }

    in.close();
    out.close();
  }
}
```

Auch hier erfolgt der Zugriff ungepuffert und somit ineffizient.

4.4.5 Filtern von Daten

Viele Ströme erwarten andere Ströme als Konstruktor-Parameter:

```
FileReader     in  = new FileReader("Eingabe.txt");
BufferedReader bin = new BufferedReader(in);
```

Das Codefragment öffnet einen `BufferedReader` auf dem Strom `in`, welcher selbst ein anderer Reader-Typ ist. Damit wird der Reader in einen anderen Reader „eingepackt“. Das Programm liest direkt von einem `BufferedReader`, der seinersets von dem `FileReader` `in` liest. Dieses geschieht hier aus zwei Gründen:

1. Das Programm kann die Methode `readLine` von `BufferedReader` verwenden, um ganze Zeilen einer Textdatei zu lesen.
2. Weiterhin liest `BufferedReader` größere Textblöcke und speichert sie intern (puffert sie). Der Zugriff wird somit effizienter.

Das Einpacken von `Reader` und `Writer` oder `InputStream` bzw. `OutputStream` ist eine sehr häufig eingesetzte Technik. Man spricht auch von Filter-Strömen, da die Klassen die Daten von einem Strom entgegennehmen, sie bearbeiten (filtern) und dann weitergeben.
Die folgenden Abschnitte zeigen einige Filter-Ströme in tyischen Einsatzszenarien.

4.4.5.1 Textdatei zeilenweise kopieren

Hier werden ein `BufferedReader` zum zeilenweisen Lesen und ein `PrintWriter` zum zeilenweisen Schreiben als Filter eingesetzt:

```
public class Copy {
  public static void main(String[] args) throws IOException {
    FileReader in  = new FileReader("Eingabe.txt");
    FileWriter out = new FileWriter("Ausgabe.txt");
    BufferedReader brIn  = new BufferedReader(in);
    PrintWriter    prOut = new PrintWriter(out);

    String line = brIn.readLine();

    while (line != null) {  // Abbruch: keine Zeile mehr
      prOut.println(line);
      line = brIn.readLine();
    }
    in.close();
    out.close();
  }
}
```

4.4.5.2 Binärdatei gepuffert kopieren

Hier wird eine Binärdatei mit einem `BufferedInputStream` gepuffert gelesen und anschließend mit einem `BufferedOutputStream` gepuffert geschrieben:

```
public class Copy {
  public static void main(String[] args) throws IOException {
    InputStream  in  = new FileInputStream("Eingabe.dat");
    OutputStream out = new FileOutputStream("Ausgabe.dat");

    in  = new BufferedInputStream(in);
    out = new BufferedOutputStream(out);

    int c;

    while ((c = in.read()) != -1) {
      out.write(c);
    }

    in.close();
    out.close();
  }
}
```

4.4.5.3 Serialisierung

Java erlaubt das Lesen und Schreiben kompletter Objekthierarchien durch die beiden Strom-Klassen `ObjectInputStream` und `ObjectOutputStream`. Objekte werden in einer bestimmten Art serialisiert, so dass sie später exakt wieder rekonstruiert werden können. Einsatzgebiete:

- Kommunikation mit RMI
- Leichtgewichts-Persistenz
- ...

Das folgende Beispiel schreibt zwei Objekte auf einen `ObjectOutputStream`:

```
public class SerialWriter {
  public static void main(String[] args) {
    try (FileOutputStream out =
                      new FileOutputStream("TheTime.dat");
         ObjectOutputStream str = new ObjectOutputStream(out)) {
      str.writeObject("Today");
      str.writeObject(new Date());
    }
    catch (IOException ex) {
      System.err.println(ex.getMessage());
    }
  }
}
```

Hier werden die im obigen Beispiel geschriebenen Objekte wieder in den Speicher zurückgelesen:

```
public class SerialReader {
  public static void main(String[] args) {
    try (FileInputStream in  =
                    new FileInputStream("TheTime.dat");
         ObjectInputStream str = new ObjectInputStream(in)) {
      String today = (String)str.readObject();
      Date date = (Date) str.readObject();
      System.out.println(today + ": " + date);
    }
    catch (IOException | ClassNotFoundException ex) {
      System.err.println(ex.getMessage());
    }
  }
}
```

Anmerkungen:

- Beim Lesen eines Objektes mit `readObject` können die folgenden Ausnahmen (Exceptions) auftreten:
  - `ClassNotFoundException`: Die Klassendatei für das zu lesende Objekt ist nicht verfügbar.
  - `InvalidClassException`: Die Klasse hat keinen Default-Konstruktor, oder der Klassencode wurde nach der Serialisierung verändert, oder die Klasse hat unbekannte Datentypen.
  - `StreamCorruptedException`: Kontrollinformationen in den serialisierten Daten wurden verändert.
  - `OptionalDataException`: Der Stream enthält primitive Datentypen statt Objekte.
- `readObject` liefert immer eine Referenz auf Object als Ergebnis zurück. Diese muss in den richtigen Typ umgewandelt werden.
- Durch Serialisierung lassen sich auch primitive Datentypen schreiben und lesen. Dazu existieren Methoden wie `writeInt`, `readInt` usw. für alle primitiven Datentypen.

Damit Objekte eigener Klassen serialisiert werden können, müssen sie die leere Schnittstelle `Serializable` implementieren. Dieses stellt nur einen Hinweis an die Strom-Klassen dar, dass Objekte der Klasse serialisiert werden dürfen (ein Flag). Weitere Bedingungen:

- Eine nicht-serialisierbare Vaterklasse der serialisierbaren Klasse muss einen parameterlosen Konstruktor besitzen.
- Es ist nicht notwendig, zusätzliche Methoden zur Serialisierung selbst zu schreiben.
- Die Methode `writeObject` aus `OjectOutputStream` schreibt alle Informationen, die zur Wiederherstellung des Objektes notwendig sind:
  - Klasse des Objektes
  - Klassensignatur
  - Werte aller nicht-transienten und nicht-statischen Attribute. Ein Attribut kann von der Serialisierung ausgeschlossen werden, indem es als `transient` deklariert wird.
    Beispiel: `private transient int cnt = 0;`
    Eine weitere Bedeutung hat `transient` nicht.
  - Es werden auch alle referenzierten Objekte geschrieben.

Der normale Serialisierungsmechanismus ist relativ langsam. Für spezielle Bedürfnisse kann er daher angepasst werden, was an dieser Stelle ab er nicht näher vertieft werden soll. Noch eine kleine Warnung: Wird eine Klasse verändert, nachdem bereits Objekte von ihr serialisiert wurden, so können diese Daten nicht mehr einfach eingelesen werden. Es reicht schon, eine Methode hinzuzufügen. Zur Lösung des Problems kann man die Berechnung einer eindeutigen ID, die die Version einer Klasse (Prüfsumme) beinhaltet, selbst kontrollieren.

4.4.6 Hinweise zum Umgang mit Dateien

Da Java eine plattformunabhängige Programmiersprache ist, sind im Umgang mit Dateinamen einige Punkte zu beachten. Hilfreich ist hier u.A. die Klasse `System` des Paketes `java.lang`. Sie bietet mit der Methode `getProperty("name")` viele Dienste, um unabhängig von der Plattform Verzeichnisdienste in Anspruch nehmen zu können.

- Benutzereinstellungen
  - `System.getProperty("user.home")`: Home-Verzeichnis des Benutzers:
  - `System.getProperty("user.dir")`: Aktuelles Arbeitsverzeichnis des Benutzers:
- Dateitrennzeichen sind auf verschiedenen Systemen unterschiedlich codiert: `\` unter Windows, `/` unter UNIX. Die korrekte Darstellung wird zur Laufzeit mit Hilfe der Methode `System.getProperty("file.separator")` ermittelt. Pfade dürfen aber auch mit einem der beiden Trennzeichen fest angegeben werden. Die API-Klassen passen das Zeichen der aktuellen Plattform automatisch an.
- Pfadtrennzeichen: Sind in einer Environment-Variablen mehrere Pfade abgelegt, so ist das Trennzeichen auch systemabhängig. Diese plattformspezifische Einstellung kann zur Laufzeit mit `System.getProperty("path.separator")` ermittelt werden.
- Zeilentrennzeichen: Die Zeilen einer Textdatei werden durch verschiedene Trennzeichen beendet: `\n` unter UNIX und Windows, `\r\n` unter DOS. Auch diese Einstellung kann zur Laufzeit mit `System.getProperty("line.separator")` ermittelt werden.

- Weitere Einstellungen: Java unterstützt weitere sogenannte Properties. Diese können mit `System.getProperties()` komplett gelesen werden. Nähere Informationen dazu sind in der API-Dokumentation der Klasse `java.lang.System` zu finden.

5

Abbildungsverzeichnis

6

Literaturverzeichnis

[Ull02] Ullenboom C.: Java ist auch eine Insel, Galileo Computing (kostenlos als HTML-Dokumente unter http://www.tutego.de/javabuch/)

[Ull03] Ullenboom C.: Java ist mehr als eine Insel, Galileo Computing

[RaScWi11] Ratz D., Scheffler J., Seese D., Wiesenberger J.: Grundkurs Programmieren in Java. Hanser Fachbuchverlag

[Kr09] Krüger G.: Handbuch der Java-Programmierung, Addison-Wesley (kostenlos als HTML-Dokumente unter http://www.javabuch.de)

[ORA01] Oracle: Java-Tutorial, http://download.oracle.com/javase/tutorial/

[ORA02] Oracle: JDK-Referenz http://www.oracle.com/technetwork/java/javase/documentation/index.html

[ORA03] Oracle: Code Conventions for the Java™ Programming Language http://www.oracle.com/technetwork/java/codeconv-138413.html

7

Stichwortverzeichnis

Symbole

A